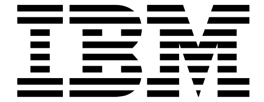
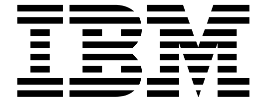
CICS Transaction Gateway for Multiplatforms
Version 9 Release 2



Developing Applications

SC34-7339-00

CICS Transaction Gateway for Multiplatforms
Version 9 Release 2



Developing Applications

SC34-7339-00

Note

Note: Before you use this information and the product it supports, read the information in Safety and environmental notices and Notices.

This edition applies to Version 9, Release 2 Modification 0 of CICS TG for Multiplatforms, program number 5724-I81 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1998, 2016.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this information vii

Part 1. Developing applications . . . 1

Chapter 1. Application programming interfaces 3

Chapter 2. Intercept plug-ins 5
CICS Intercept plug-in 5
  CICS Intercept plug-in development 6
  Java BasicCicsPlugin sample 6
Gateway Intercept plug-in 6

Chapter 3. Client applications 7
JEE applications 8

Chapter 4. External Call Interface (ECI) . 9
List CICS systems 9
The ECI request 9
  I/O parameters on ECI calls 9
  Program link calls 10
  Status information calls 12
  Retrieving replies from asynchronous ECI requests 12
Introduction to channels and containers 13
ECI and CICS transaction IDs 13
Timeout of the ECI request 14
  Request timeout 14
  Response timeout 14
Security in the ECI 15
IPIC support for ECI 15
ECI performance considerations when using COMMAREAs 15

Chapter 5. External Presentation Interface (EPI). 19
List CICS systems 19
EPI concepts 19
  Adding and deleting terminals 19
  Starting transactions 19
  Sending and receiving data 20
  Managing CICS conversations 20
Terminal characteristics 21
Timeout of the EPI request 22
Security in the EPI 22
  Specifying terminal sign-on capability 23
Automatic transaction initiation (ATI) 24
Restrictions on application design when using EPI 24
3270 data streams for the EPI 25
BMS map conversion utilities 25

Chapter 6. External Security Interface (ESI). 27
I/O parameters on ESI calls 27
Using ESI to manage passwords 28

Chapter 7. JSON web services 29
Concepts of RESTful JSON web services. 31
Creating a Request-Response JSON web service from high-level language structures 32
Creating a Request-Response JSON web service from JSON Schemas 33
Creating a RESTful JSON web service 34
The JSON web services assistant 36
  Creating a Request-Response WSBind file from a language structure 37
  Creating a Request-Response WSBind file from JSON Schemas 45
  Creating a RESTful WSBind file 51
  High-level language and JSON Schema mapping 58
JSON web service restrictions 90
Error responses from JSON web services 91

Chapter 8. Statistics APIs. 93
Statistical data overview 93
API and protocol version control 93
Statistics C API 95
  Calling the C API 95
  Statistics C API components 95
  Statistics C API program structure. 96
  C API data types 97
  Statistics C API trace levels. 100
  C API functions 101
  Correlating results and error checking 108
Statistics Java API 108

Chapter 9. Code page information . . 111

Chapter 10. Programming in Java. . . 113
Overview of the programming interface for Java 113
  Writing Java client applications 113
  SSL cipher suites in Java Client applications . . 114
  JavaGateway security. 115
Making ECI calls from a Java client program . . . 115
  Linking to a CICS server program 116
  Creating Java channels and containers for ECI calls 116
  Managing an LUW 117
  Retrieving replies from asynchronous requests 117
Making EPI calls from a Java Client program. . . 118
  EPI support classes 118
  EPIRequest 128
  EPI security 129
Making ESI calls from a Java client program . . . 129
Compiling and running a Java Client application 129

© Copyright IBM Corp. 1998, 2016

Setting stack and heap sizes 129
Setting up the CLASSPATH 130
Integration testing Java applications using a Gateway Intercept plug-in 130
The sample plug-in 131
Plug-in development 131
Enabling a Gateway intercept plug-in in a Java SE application 131
Problem determination for Java client programs 132
Tracing in Java client programs 132
Security for Java client programs 134
CICS Transaction Gateway security classes . . 134
Using a Java 2 Security Manager 135
Signing Applets and Web Start Applications . . . 136
Using the CICS TG OSGi bundle 137

Chapter 11. Programming using the JEE Connector Architecture 139
Overview of the JCA programming interface . . . 139
The Common Client Interface (CCI) 139
The programming interface model 139
Record objects 140
ECI resource adapter 140
EPI resource adapter 141
Managed and nonmanaged environments . . . 141
The Common Client Interface 141
Generic CCI Classes 141
CICS-specific classes 142
Using the ECI resource adapter 142
Using the ECI resource adapter with channels and containers 143
Connection to a CICS server using the ECI resource adapter 144
Linking to a program on a CICS server. . . . 145
ECI resource adapter CICS-specific records using the streamable interface 146
Transaction management 146
Samples 148
Using the EPI resource adapter 148
Connecting to a CICS server using the EPI resource adapter CCI 149
Starting a transaction. 149
Sending and receiving data. 150
Writing LogonLogoff classes 152
Samples 154
Using the resource adapters in a nonmanaged environment. 154
Creating the appropriate ConnectionFactory object 154
Saving and reusing connection factories . . . 155
Running the JEE resource adapters in a nonmanaged environment 156
Compiling applications 156
Compiling and running JEE components 156
Integration testing JEE components using an intercept plug-in 157
Security credentials and the CICS resource adapters 157
JEE tracing 157
Tracing issues relating to serialized interfaces and ConnectionFactory objects 158
Resource adapter samples 158
ECI COMMAREA sample 158
EPI sample 160
ECI channels and containers sample. 161
Assistance in coding CCI applications 162
JEE Connector Architecture API 162

Chapter 12. Programming in C and COBOL 165
Overview of the programming interfaces for C and COBOL 165
Making ECI V1 calls from C and COBOL programs 165
CICS_ExternalCall. 166
Program link calls. 166
Reply solicitation calls 167
Security credentials in ECI V1 168
Making ECI V2 and ESI V2 calls from C programs 168
Making ECI V2 calls 168
Making ESI V2 calls 169
Establishing a connection to a Gateway daemon 170
Program link calls. 171
Reply solicitation calls 173
Using channels and containers in ECI V2 applications 173
Tracing in ECI V2 and ESI V2 applications . . 174
Security credentials in ECI V2 175
Multithreaded ECI V2 and ESI V2 applications 175
Making EPI calls from C and COBOL programs 176
EPI versions. 177
EPI Initialization and termination 177
Adding a terminal to CICS 177
Deleting a terminal 177
Starting transactions 178
Sending and receiving data. 178
Managing pseudoconversations 178
Events and callbacks 178
Processing events 179
Automatic transaction initiation (ATI) 180
3270 data streams for the EPI 180
Making ESI V1 calls from C and COBOL programs 183
Verifying a password using ESI 183
Changing a password using ESI 184
Setting default security using ESI. 184
Compiling and linking C and COBOL applications 184
Windows 185
IBM AIX 185
Solaris. 186
Linux 187
HP-UX 188

Chapter 13. Programming in C++ . . . 189
Overview of the programming interface for C++ 189
Writing C++ Client applications 189
Making ECI calls from a C++ Client program . . 189
Linking to a CICS server program 189
Managing logical units of work 191
Retrieving replies from synchronous requests 192
Retrieving replies from asynchronous requests 193
Reply solicitation calls 194
ECI security 195

Finding potential servers 196
Monitoring server availability 196
C++ ECI classes 197
Making EPI calls from a C++ Client program. . . 197
Adding a terminal to CICS 198
EPI call synchronization types 198
Sending and receiving data. 200
Converting BMS maps and using the Map class 201
Support for Automatic Transaction Initiation (ATI) 204
EPI Security 205
C++ EPI classes 206
Compiling and linking a C++ application 206
Problem determination for C++ Client programs 207
Handling Exceptions 207

Chapter 14. Programming using COM 211
Overview of the programming interface for COM 211
Writing COM Client applications 211
Making ECI calls from a COM Client program . . 214
Linking to a CICS server program using Visual Basic 214
Linking to a CICS server program using VBScript 215
Managing an LUW 215
Retrieving replies from asynchronous requests 216
ECI security 217
ECI CICS Server Information and Connection Status 218
ECI COM classes 218
Making EPI calls from a COM Client Program . . 219
Adding a terminal to CICS 219
Sending and receiving data. 220
EPI call synchronization types 221
Converting BMS maps and using the Map class 222
Support for Automatic Transaction Initiation (ATI) 223
EPI Security 223
EPI CICS Server Information 224
EPI COM classes 224
Problem determination for COM Client programs 225
Handling exceptions 225

Chapter 15. Developing Microsoft .NET Framework-based applications . 227
Overview of the programming interface 227
Making ECI calls from Microsoft .NET Framework-based programs 227
Making ESI calls from Microsoft .NET Framework-based programs 228
Using channels and containers in Microsoft .NET Framework-based programs 229
Developing ECI and ESI applications based on the Microsoft .NET Framework. 230
Problem determination for Microsoft .NET Framework-based client programs 231
Tracing for Microsoft NET Framework-based client programs. 231

Chapter 16. Request monitoring exits 233
Java request monitoring exits 233
Correlation points available in the exits. . . . 236
Data available by FlowType and RequestEvent 236
ECI and EPI C exits 243
Loading the exits 243
Sample exits and interface definitions 244
Writing your own user exits 244
Diagnostic information 245
EPI user exits 246

Chapter 17. Creating a CICS request exit 249
Writing a CICS request exit. 249

Chapter 18. Sample programs 251
Sample CICS programs and maps 251
Java client samples 252
Compiled Java samples 252
Running the sample programs. 252
Connecting to CICS Transaction Gateway . . . 253
Java ECI base class samples 253
Java EPI base class samples 256
Java ESI base class samples. 256
Java EPI support class samples 256
JEE samples 257
JEE ECIDateTime sample 257
JEE EPIPlayScript sample 259
JEE EC03Channel sample 261
C remote client samples 262
C ctgecib1 sample 262
C ctgecib2 sample 263
C ctgecib3 sample 264
C ctgesib1 sample 264
C, C++ and COBOL local client samples 265
Building C sample programs 265
Building C++ sample programs 266
Building COBOL sample programs 266
C/C++ ECIB1 sample 267
C/C++ ECII1 sample 267
C/C++ EPIB1 sample. 267
C/C++ EPII1 sample 267
C/C++ EPIA1 sample 268
C/C++ ECIA1 sample 268
C/C++ ESIB1 sample. 268
ECI extensions that are environment-dependent 268
C# and Visual Basic samples based on Microsoft .NET Framework 272
C# and Visual Basic EciB1 sample based on Microsoft .NET Framework. 272
C# and Visual Basic EciB3 sample based on Microsoft .NET Framework. 272
C# and Visual Basic EsiB1 sample based on Microsoft .NET Framework. 273
User exit samples 273
Java security exit data compression samples . . 273
Java request monitoring exit samples 274
Java CICS request exit samples 275
C ECI and EPI user exit template samples. . . 276
C ECI and EPI user exit samples 276

Building C user exit samples 276
C and Java statistics API samples. 277
C ctgstat1 statistics API sample 277
Java Ctgstat1 statistics API sample 278
Java Ctgstat2 statistics recording sample 278

Part 2. Appendixes 279

Glossary 281

Related literature 303

Accessibility 305
Installation 305
Configuration Tool accessibility 305
Starting the Gateway daemon 305
cicsterm 306

Notices 309
Programming interface information 311
Trademarks 311
Terms and conditions for product documentation 311
IBM Online Privacy Statement. 312
Safety and environmental notices. 312
Trademarks 312

About this information

This information describes the planning, installation, configuration, and operation of the IBM® CICS® Transaction Gateway and the IBM CICS Transaction Gateway Desktop Edition products.

You should be familiar with the operating system on which CICS Transaction Gateway runs and also with Internet terminology.

This information includes trademarks including Java™, for more information about Trademarks, see the Trademark information at the back of this publication.

© Copyright IBM Corp. 1998, 2016

Part 1. Developing applications

This section contains information you need to make your applications work with CICS Transaction Gateway.

The CICS TG Software Development Kit (SDK) enables CICS TG application development on a different system to where the CICS TG product is installed. For more information, see .

© Copyright IBM Corp. 1998, 2016

Chapter 1. Application programming interfaces

The CICS Transaction Gateway supports the integration of CICS servers and client applications. There is a standard set of functions to allow user applications to call CICS programs, initiate CICS 3270 transactions or perform password expiry management (PEM).

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). CICS TG can also expose JSON web services which allow HTTP and HTTPS clients to call CICS programs.

There are also statistical data APIs, which enable a user application to collect statistical information about a running CICS Transaction Gateway.

Related information:

Chapter 4, “External Call Interface (ECI),” on page 9
The External Call Interface (ECI) enables a client application to call a CICS program synchronously or asynchronously. It enables the design of new applications to be optimized for client/server operation, with the business logic on the server and the presentation logic on the client.

Chapter 5, “External Presentation Interface (EPI),” on page 19
The External Presentation Interface (EPI) enables a user application to install a virtual IBM 3270 terminal into a CICS server. The EPI also enables a user application to delete a virtual IBM 3270 terminal from a CICS server.

Chapter 8, “Statistics APIs,” on page 93
The statistics APIs enable user applications to obtain runtime statistics on the Gateway daemon. To use the statistics APIs, the Gateway daemon must be configured with a statistics API protocol handler.

© Copyright IBM Corp. 1998, 2016

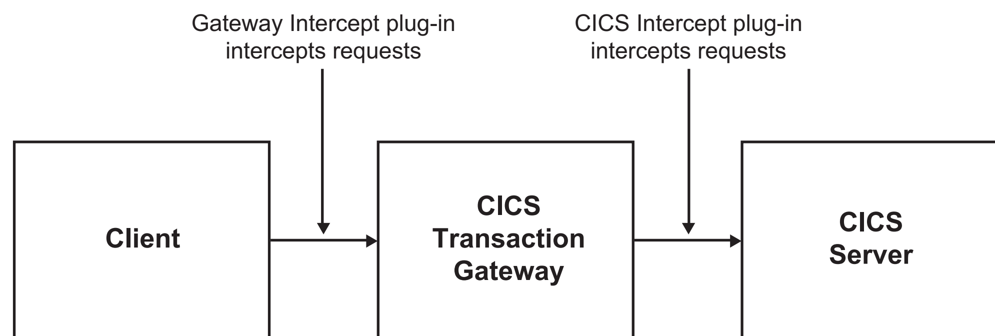
Chapter 2. Intercept plug-ins

You can use intercept plug-ins to test applications without requiring a running CICS server.

There are two intercept plug-ins:

Gateway Intercept plug-in
A Gateway Intercept plug-in intercepts requests before they reach the Gateway. It can be used to test Java applications without a running CICS server or Gateway daemon. .

CICS Intercept plug-in
A CICS Intercept plug-in intercepts requests between the Gateway and CICS servers. You can use a CICS Intercept plug-in and tools such as IBM Rational® Integration Tester to test applications without requiring a running CICS server.



CICS Intercept plug-in

You can use a CICS Intercept plug-in to intercept ECI, EPI, and ESI calls from Java, ECI V2, ESI V2, JSON web service and .NET Framework based applications.

These applications can be tested without the need for a running CICS server. Tools such as IBM Rational Integration Tester provide a CICS Intercept plug-in. Alternatively, you can develop your own by writing a Java program which implements the CicsIntercept interface.

You can use a plug-in to test your application by writing a Java program which implements the CicsIntercept interface, and enabling this plug-in for the Gateway daemon.

When enabled, a CICS Intercept plug-in is notified before a request is sent to CICS. Before a request is sent, the plug-in can change the properties of the request object. If `Return` is returned by the plug-in, the Gateway daemon will bypass calling CICS and return the request object as it stands. If `Continue` is returned by the plug-in, the Gateway daemon will continue the normal processing of the request to CICS.

© Copyright IBM Corp. 1998, 2016

CICS Intercept plug-in development

The API for implementing a Gateway intercept plug-in is provided in `ctgclient.jar`

Plug-in classes must implement the `com.ibm.ctg.client.CicsIntercept` interface. For more information, see Interface CicsIntercept

To compile your plug-in class, `ctgclient.jar` must be available on the class path.

Java BasicCicsPlugin sample

The `com.ibm.ctg.samples.intercept.BasicCicsPlugin` sample, which is written in Java, is provided with this feature.

This sample shows the basic use of the CICS Intercept plug-in. The sample program intercepts requests for the EC01 and EC03 CICS server programs and returns a simulated response. The sample can be used with all EciB1 and EciB3 samples in remote mode, and with the WSBind file from scenario SC11 for JSON web services. The BasicCicsPlugin sample is compiled into `ctgsamples.jar` along with all other Java samples.

To enable the BasicCicsPlugin sample on the Gateway daemon, set the **`cicsintercept`** value in the configuration file (`ctg.ini`) to `com.ibm.ctg.samples.intercept.BasicCicsPlugin`.

Gateway Intercept plug-in

A Gateway Intercept plug-in intercepts requests before they reach the Gateway.

A Gateway Intercept plug-in can be used to test Java applications without a running CICS server or Gateway daemon. For more information see "Integration testing Java applications using a Gateway Intercept plug-in" on page 130

Chapter 3. Client applications

CICS Transaction Gateway supports client applications running in local or remote mode topologies. Client applications enable access to CICS server transactions and programs from the host machine.

The following figure shows both Java and non-Java client applications running in both local and remote mode on a UNIX, Linux or Windows System.

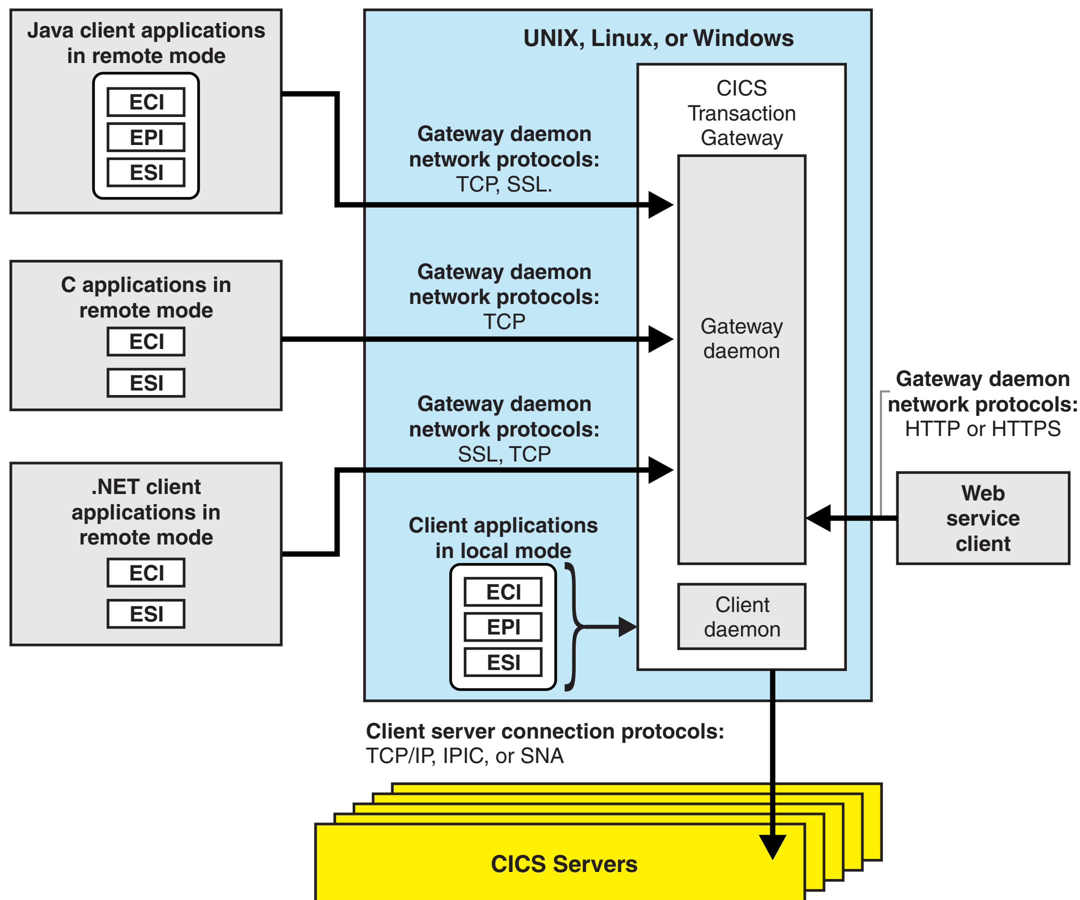


Figure 1. CICS Transaction Gateway for UNIX, Linux, or Windows

Note:

1. Java client applications use the Gateway classes to communicate with CICS servers.
2. JCA client applications use the JEE CICS resource adapters to communicate with CICS servers.
3. C client applications running in remote mode use the ECI V2 and ESI V2 C language bindings to communicate with CICS servers.
4. C client applications running in local mode use the ECI V1, ESI V1 and EPI C language bindings to communicate with CICS servers.

© Copyright IBM Corp. 1998, 2016

JEE applications

CICS Transaction Gateway implements the JCA by providing JEE resource adapters.

These resource adapters support the JEE Common Client Interface (CCI) defined by the JCA and are a middle-tier between JCA-compliant applications and CICS Transaction Gateway. The JEE application server can run locally on the same machine as CICS Transaction Gateway, or remotely.

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 JEE application server.

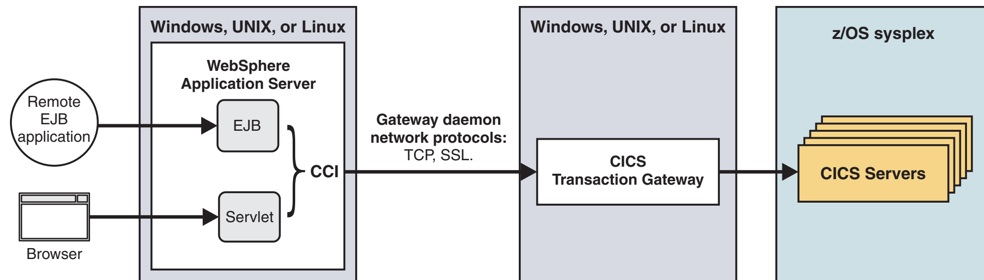


Figure 2. Topology with CICS Transaction Gateway and WebSphere Application Server in remote mode

Chapter 4. External Call Interface (ECI)

The External Call Interface (ECI) enables a client application to call a CICS program synchronously or asynchronously. It enables the design of new applications to be optimized for client/server operation, with the business logic on the server and the presentation logic on the client.

The external interfaces allow non-CICS applications to access and update CICS resources by calling CICS programs. When used in conjunction with CICS communication, the external interfaces enable non-CICS programs to access and update resources on any CICS server. This method of using the external interfaces supports such activities as the development of graphical user interface (GUI) front ends for CICS applications and it allows the integration of CICS servers and non-CICS servers.

The application can connect to several CICS servers at the same time and have several called CICS programs running concurrently. The CICS programs can transfer information using COMMAREAs or channels.

CICS programs that are invoked by an ECI request must follow the rules for distributed program link (DPL) requests. For information on DPL requests, refer to your CICS server documentation.

List CICS systems

To determine which CICS servers requests can be directed to, user applications can query the CICS Transaction Gateway for a list of CICS systems.

The query returns a list of the CICS servers that have been defined within the CICS Transaction Gateway. There is no guarantee that communication links exist between the CICS servers and the CICS Transaction Gateway or that any of the CICS servers are actually available.

The ECI request

An ECI request can be used to make program link calls, status information calls and reply solicitation calls.

I/O parameters on ECI calls

Input parameters passed to the CICS server with an ECI call, and output parameters returned to the user application following an ECI call.

Input parameters

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.

© Copyright IBM Corp. 1998, 2016

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

Message qualifier
: For reply solicitation calls, a token that identifies the asynchronous request.

Password
: The password or password phrase 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. This can be a logical CICS server or an actual CICS server name.

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.

Output parameters

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 an LUW.

Message qualifier
: For asynchronous calls, a token that identifies the asynchronous request and can be used to retrieve the response.

Program link calls

An ECI request to call a program on a CICS server results in a program link call to attach the CICS mirror transaction to run the server program.

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. This is referred to as SYNCONRETURN.

Extended

The user application controls whether recoverable resources are committed or rolled back. Multiple calls are possible, allowing a logical unit of work (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. An LUW is the processing that a CICS server program performs between sync points. A sync point is the point at which all changes to recoverable resources that were made by a task since its last sync point 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 sync point.

ECI-based communications between the CICS server and the CICS Transaction Gateway 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.

Only one transaction can be active at a time in a logical unit of work, so if you are using asynchronous requests with extended LUWs, there is a restriction that the reply from one ECI call must be retrieved before making a subsequent ECI call within the same LUW.

Managing logical units of work

On a successful return from the first of a sequence of extended ECI calls for an LUW, the user application is returned an 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 an LUW is outstanding (incomplete), the user application issues 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 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 you must commit or rollback the LUW. If you do not, the unit of work remains outstanding and prevents a normal shutdown of CICS Transaction Gateway.

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 as soon as 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.

Note: EXCI does not support asynchronous ECI request status link calls for `Immediate`, `Change`, and `Cancel` types.

Retrieving replies from asynchronous ECI requests

Callbacks and reply solicitation calls can be used to retrieve 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. Callbacks are not available for all APIs.

Reply solicitation calls

User applications that issue asynchronous calls can have several ECI requests outstanding at a time. Reply solicitation calls can be used by the calling application to retrieve the replies for each outstanding request. There are two types of reply solicitation call:

General
: Retrieves all replies 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. Depending on the API that the application uses, message qualifiers are either automatically generated or a they have to be manually assigned to each asynchronous call within a single application.

If no reply is available, reply solicitation calls can either wait for a reply or return control directly to the user application.

Introduction to channels and containers

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

Each container is a named block of binary (BIT) or character (CHAR) data that is not limited to 32 KB. Containers are grouped together in sets called channels.

The channel and 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. Any number of containers 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 consist of 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 keep track of the size of the data.
- CICS automatically destroys containers and their storage when they are no longer required.

When you are using channels and containers in preference to COMMAREAs, note that:

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

For more information about using channels and containers see the following topics:

- Using channels and containers in the JCA framework, see “Using the ECI resource adapter with channels and containers” on page 143.
- Using channels and containers with ECI calls for Java clients, see “Creating Java channels and containers for ECI calls” on page 116.
- Using channels and containers with ECI V2 calls for C clients, see “Using channels and containers in ECI V2 applications” on page 173
- Using channels and containers with ECI calls for NET Framework-based client applications, see “Using channels and containers in Microsoft .NET Framework-based programs” on page 229.

ECI and CICS transaction IDs

The transaction ID of the mirror transaction for an ECI call can be controlled through the parameters TPNName and 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 CPMI.

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

Table 1. Specifying TPNName and TranName

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

Timeout of the ECI request

An ECI timeout is the time that the CICS Transaction Gateway will wait for a response to an ECI request sent to a CICS server before returning a timeout error to the Client application.

An ECI timeout can occur either before or after the ECI request has been sent to the CICS server, so there are two timeout conditions, request timeout and response timeout.

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.

A response timeout can occur in the following situations:

- The call was intended to be the only call of a new LUW. The LUW was started, but CICS Transaction Gateway 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 CICS Transaction Gateway 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.

Note: If you use timeouts in your applications, you must ensure that the application specifically confirms if resources were committed or backed out following a timeout.

Security in the ECI

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

ECI security involves authentication and authorization. During authentication, checks are performed to ensure that the user ID and password or password phrase information associated with an ECI call are valid. During authorization, a check is performed on the CICS server to ensure that the authenticated user is allowed to access the requested resource.

The user application can set the user ID and password or password phrase on an ECI request for a conversation with a specific CICS server. These values override any default values set for the CICS server connection.

IPIC support for ECI

IPIC connections do not support ECI State calls. If you are using local mode, IPIC connections are not displayed in the list systems 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 list systems call.

IPIC does not support the following ECI calls:

- ECI_STATE_ASYNC
- ECI_STATE_ASYNC_JAVA
- ECI_STATE_CANCEL
- ECI_STATE_CHANGED
- ECI_STATE_IMMEDIATE
- ECI_STATE_SYNC
- ECI_STATE_SYNC_JAVA (deprecated)

ECI performance considerations when using COMMAREAs

The performance of ECI might be affected by the amount of data transmitted over the network in the COMMAREA between the client application and the CICS server.

To reduce the number of bytes transmitted over network protocols between the Gateway daemon and the CICS server the CICS Transaction Gateway removes trailing nulls from the COMMAREA before transmission and restores them again after transmission, this is referred to as null stripping. Null stripping is transparent to client application programs which always see the full-size COMMAREA.

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 CICS server also performs null stripping before transmitting the COMMAREA back over the network.

To reduce the number of bytes transmitted between a Client application and the Gateway daemon, functions are provided to set the length of data in the COMMAREA that is to flowed to the CICS server, COMMAREA outbound length, and to set the length of COMMAREA data returned from the Gateway daemon to the client application, COMMAREA inbound length.

For JEE applications:
- the outbound COMMAREA length is set automatically by the CICS Transaction Gateway to remove trailing nulls
- use the setReplyLength and getReplyLength methods of the ECIInteractionSpec for the inbound COMMAREA length

For Java Client applications use the following methods:
- setCommareaOutboundLength
- setCommareaInboundLength
- getInboundDataLength

For ECI v2 applications use the **CTG_ECI_PARMS** parameter block fields:
- commarea_outbound_length
- commarea_inbound_length

For NET Framework-based applications use the EciRequest class fields:
- CommareaInboundLength
- CommareaOutboundLength

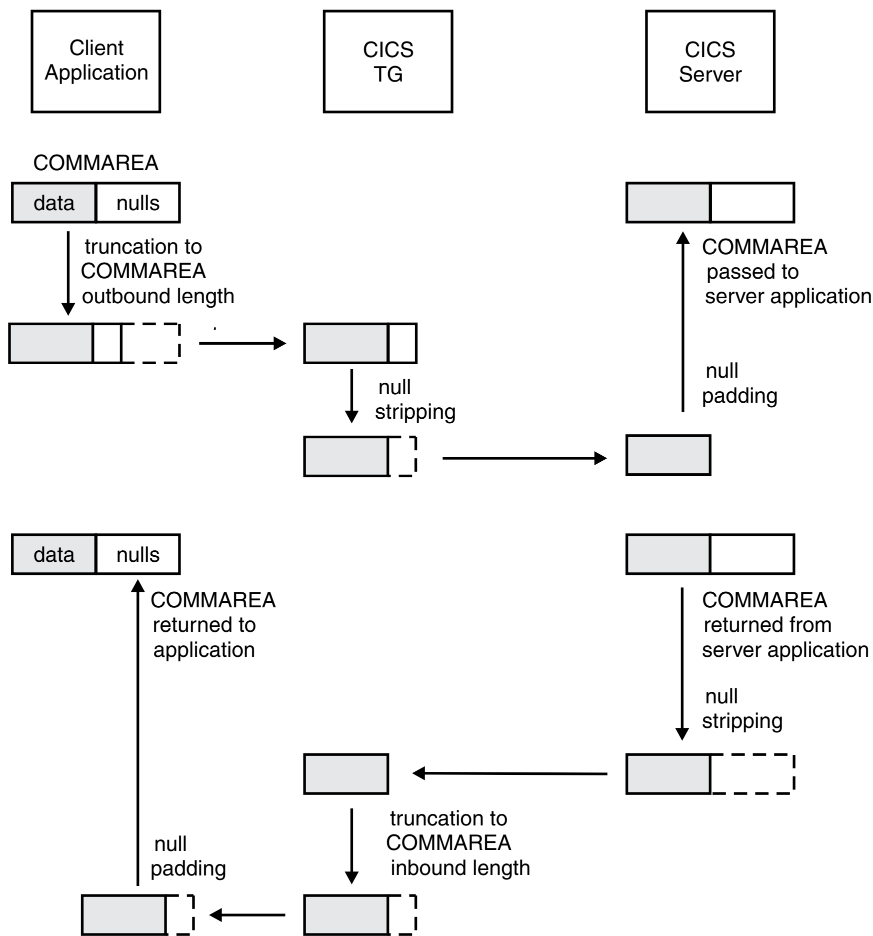


Figure 3. COMMAREA data flow optimizations using IPIC, TCP/IP or SNA

Chapter 5. External Presentation Interface (EPI)

The External Presentation Interface (EPI) enables a user application to install a virtual IBM 3270 terminal into a CICS server. The EPI also enables a user application to delete a virtual IBM 3270 terminal from a CICS server.

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 the CICS Transaction Server Application Programming Guide*CICS Transaction Server Application Programming Guide*.

List CICS systems

To determine which CICS servers requests can be directed to, user applications can query the CICS Transaction Gateway for a list of CICS systems.

The query returns a list of the CICS servers that have been defined within the CICS Transaction Gateway. There is no guarantee that communication links exist between the CICS servers and the CICS Transaction Gateway or that any of the CICS servers are actually available.

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 21.

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.

© Copyright IBM Corp. 1998, 2016

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 on SPARC 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 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 22 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 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 22 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 23.

Userid
The user ID 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 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 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 user ID and password might be required for each conversation that takes place between the CICS Transaction Gateway and the CICS server, depending on how the CICS Transaction Gateway and the CICS server have been configured.

EPI security involves:

Authentication
The CICS server checks that the user ID and password information associated with a terminal is valid. The frequency with which the user ID 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 user ID and password can be set at terminal or connection level. Both types can be set by the user application. If there are no user ID and password values for the terminal, the values for the connection are used. For information about how to set the connection user ID and password, refer to the information about Security in the *CICS Transaction Gateway: UNIX and Linux Administration* or the *CICS Transaction Gateway: Windows Administration*.

The requirement for a user ID 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. The following table shows the results of a request to override the default sign-on capability for different CICS servers.

Table 2. 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 user ID 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 user ID.

The user ID 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 user ID and password to be used with a particular CICS server. For details, refer to the information about Default

connection settings in the *CICS Transaction Gateway: UNIX and Linux Administration* or the *CICS Transaction Gateway: Windows Administration*.

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 user ID and password are determined by the user application and are embedded in the 3270 data. If the user ID is authenticated, subsequent transactions started at the terminal are executed in the CICS server with the authorities assigned to the authenticated user ID. Transactions started before a sign-on transaction has completed have the authorities granted to the default user ID defined for the CICS server. A check is also done against the user ID associated with the connection to see whether the CICS Transaction Gateway has authority to execute the transaction.

The user application can start a sign-off 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 each case, subsequent transactions started at the terminal are executed with the authorities assigned to the CICS server default user ID.

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

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 application can start a CICS server transaction on 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 list of actions that cannot be carried out when using an EPI user application.

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

- Use 14- and 16- bit addresses and structured fields, because the CICS Transaction Gateway supports 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 alternative 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

All data flows for the EPI are in ASCII format, and structured fields are not supported. The contents of the data buffer might be code-page converted if the buffer is passed between CICS systems, in which case the data must be limited to ASCII and EBCDIC characters.

The data streams implemented for the EPI follow those defined in the *3270 Data Stream Programmer's Reference*. 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

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 returns valid 3270 data to be converted for the transaction.

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.

Chapter 6. External Security Interface (ESI)

The External Security Interface (ESI) enables user applications to perform security-related tasks such as the viewing and updating of user IDs and passwords held by an external security manager (ESM), or the setting of default security credentials used on CICS server connections.

I/O parameters on ESI calls

Information about the input and output parameters on ESI calls.

Input parameters

New password
The new password or password phrase for the specified user.

Current password
The current password or password phrase for the specified user.

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

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

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

Output parameters

Expiry date
The date on which the password or password phrase will expire.

Expiry time
The time at which the password or password phrase will expire.

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

Last access date
The date on which the user ID was last accessed.

Last access time
The time at which the user ID was last accessed.

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

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

© Copyright IBM Corp. 1998, 2016

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:

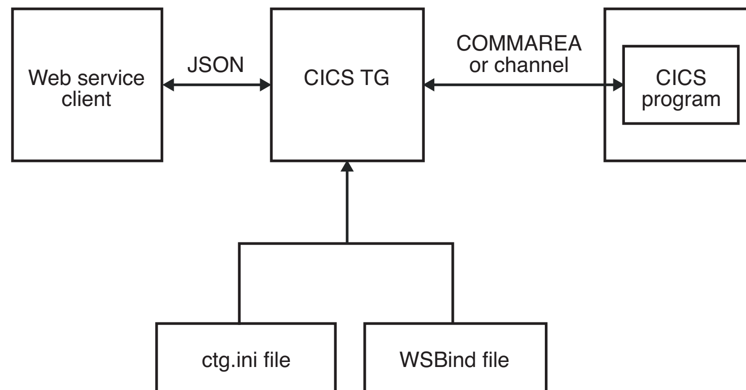
- Verify that a password matches the password or password phrase recorded by the CICS External Security Manager (ESM) for a specified user ID.
- Change the password or password phrase recorded by the CICS ESM for a specified user ID.
- Determine if a user ID is revoked, or a password or password phrase has expired.
- Obtain additional information about a verified user such as:
  - When the password or password phrase 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

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

Chapter 7. JSON web services

CICS TG can create web services that use JavaScript Object Notation (JSON) for HTTP request and response payloads.

CICS TG converts the JSON data into a channel or COMMAREA payload, which is then passed to a CICS program. CICS TG then converts the data returned from CICS back to JSON.



CICS TG supports two types of JSON web service, Request-Response and RESTful:

Request-Response

A Request-Response web service accepts a JSON object as input and produces another JSON object for output. The web service is implemented by a program in CICS, and CICS TG is responsible for transforming the incoming JSON object into application data and calling the program in CICS. The application returns output data back to CICS TG, and CICS TG transforms the output data into JSON data to return to the client.

In this scenario, the web service client must call the web service with the HTTP POST method.

A Request-Response mode JSON web service can be developed in either bottom-up mode or top-down mode.

In bottom-up mode, an existing CICS program is exposed as a JSON web service. The JSON web services assistant is used to generate new JSON Schemas that describe the input and output JSON data of the web service based on existing high-level language structures.

In top-down mode a new JSON web service can be developed to implement an interface that is described by existing JSON Schemas. The JSON web services assistant is used to generate new high-level language structures, that describe the CICS program payload based on the existing JSON Schemas.

© Copyright IBM Corp. 1998, 2016

The Request-Response pattern can be used to build JSON web services that target either COMMAREA or channel based CICS programs.

RESTful

A RESTful JSON web service implements the architectural principles of the REpresentational State Transfer (REST) design pattern. This design pattern is unlikely to be relevant for existing CICS programs, so is available only in top-down mode.

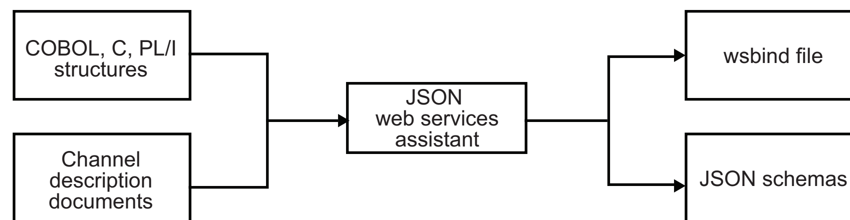
A JSON web services assistant is used to generate new high-level language structures that describe the CICS program payload based on the existing JSON Schema. A CICS program must be written to implement the service and it needs to behave differently depending on the HTTP method and URI that was used for the incoming request.

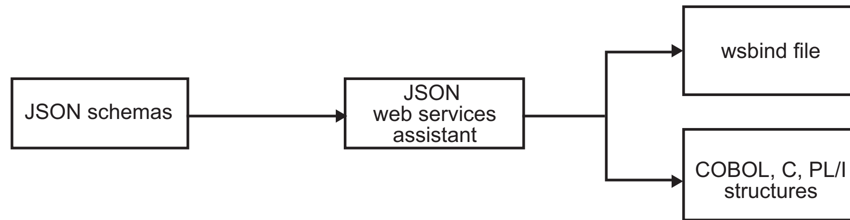
CICS TG implements a pure style of RESTful application, where the data format for POST (create) GET (inquire) and PUT (replace) are the same. RESTful JSON web service programs must use a channel-based program interface. Using a COMMAREA is not supported. This means that RESTful JSON web services can only be implemented when CICS TG is connected to CICS using the IPIC protocol.

For more information, see "Concepts of RESTful JSON web services" on page 31.

Creating a JSON web service

You can create a JSON web service from high-level language structures (bottom-up) or from JSON Schemas (top-down). The JSON web services assistant helps you create a JSON web service by generating a WSBind file, and JSON Schemas or high-level language structures. The JSON web services assistant is run using the executable script **ctgassist**. The assistant is located in the directory `<install_path>/bin` or in the CICS TG SDK directory `cicstgsdk/webservices/assistant`.





WSBind files

A web service binding (WSBind) file describes which CICS program is called and how the data is transformed between JSON data format and CICS program payload when a JSON web service is started.

JSON Schemas

JSON Schemas are used by JSON web services to describe the JSON data format for HTTP request and response payloads. For more information on the JSON Schema specification, see http://json-schema.org/. At the time of writing this is a draft specification which is making its way through the Internet Engineering Task Force (IETF) standardization process. The JSON web services assistant supplies a partial implementation of draft 4 of this emerging specification.

Concepts of RESTful JSON web services

The concepts behind RESTful web services and how they differ from Request-Response services.

RESTful web services

REpresentational State Transfer, or REST, is a design pattern for interacting with resources stored in a server. Each resource has an identity, a data type, and supports a set of actions. The RESTful design pattern is normally used in combination with HTTP. In this context, the resource's identity is its URI; the data type is its Media Type; and the actions are made up of the standard HTTP methods (GET, PUT, POST, and DELETE).

This style of service differs from Request-Response style web services as follows:

- Request-Response services interact with a program, whereas RESTful services typically interact with data (referred to as “resources”).
- Request-Response services involve application defined “operations”, but RESTful services avoid application-specific concepts and rely instead on using just the HTTP methods to specify the operation.
- Request-Response services have different data formats for each message, but RESTful services typically share a data format across different HTTP methods.

The four major HTTP methods define the four operations that are commonly implemented by RESTful Services. The HTTP POST method is used for creating a resource; GET is used to query it; PUT is used to change it; and DELETE is used to remove it. The most common RESTful architecture involves a shared data model that is used across these four operations. This data model defines the input to the

POST method (create), the output for the GET method (inquire) and the input to the PUT method (replace). This simple design pattern is popular within the RESTful community, but it is not the only RESTful design pattern. Some RESTful APIs are designed in other ways.

A fifth HTTP method, called “HEAD”, is sometimes supported by RESTful web services. This method is equivalent to GET, except that it returns only HTTP headers, and no body data. It can be used to test the existence of a resource without returning the resource data itself. Not all RESTful APIs support the use of the HEAD method.

Traditional CICS applications are unlikely to match the RESTful architectural pattern. Typical CICS applications implement multiple operations, each of which have data models for input and output formats. These existing operations are unlikely to map directly to the four HTTP methods. For this reason, the RESTful architectural pattern is primarily aimed at new applications in CICS. To expose existing CICS applications as RESTful web services, you might need to wrap them with a new interface that conforms to the RESTful principles.

The URI

The identity of a RESTful service is indicated by its URI. A URI can be made up of several components, including the host name, port number, the path, and an optional query string. The domain name and port number together target a configured HTTP or HTTPS protocol handler in CICS TG. The URI path is a qualifier, and might be sufficient to uniquely identify the service. However, many RESTful web services use an extra query string to identify the precise resource. Consider the following examples:

- `http://www.example.org:10000/JSONServices/AccountService`
- `https://www.example.org:10000/JSONServices?Service=Account`

In the first example, the URI path is `JSONServices/AccountService`. In the second example, the path is `JSONServices` and there is an extra query string of `Service=Account`. Both styles of URI are considered to be acceptable for a JSON web service implementation.

The URI used to start the web service is defined in the **`Uri`** parameter of a CICS TG web service definition or the WSBind file. This URI can contain a query string.

Creating a Request-Response JSON web service from high-level language structures

How to create a JSON web service, starting from high-level language structures in COBOL, C, or PL/I, or channel description documents.

Procedure

1. Create high-level language structures or channel description documents for the request and response data structure for the CICS program. For existing CICS programs, you may already have an appropriate file or you may need to create one to match the program specification.
2. Run the JSON web services assistant to create a WSBind file and JSON Schemas.
3. Configure CICS Transaction Gateway with an HTTP or HTTPS protocol handler and a JSON web service using the generated WSBind file.

4. Start CICS TG and test that you can invoke the JSON web service. Generated JSON Schemas can be used to develop a JSON web services client to invoke the web service.

Creating a channel description document

If the CICS program uses channels with multiple containers, create a channel description document that describes how each container appears in the JSON Schemas. Alternatively, you can use a single container in a channel and not create a channel description document. For more information, see "Creating a channel description document" on page 43.

The JSON assistant

Run the CICS TG JSON web services assistant by customizing the `CTGLS2JS.txt` sample parameter file using `MAPPING-MODE=LS2JS` and running **`ctgassist <parameter file>`**. The assistant generates a WSBind file and JSON schemas from the language structures. `CTGLS2JS.txt` contains the required parameters for the LS2JS mapping mode. For more information, see "Creating a Request-Response WSBind file from a language structure" on page 37.

The **`ctgassist`** program and samples are shipped with the CICS TG SDK in the `webservices` directory

Creating a Request-Response JSON web service from JSON Schemas

How to create a Request-Response JSON web service, starting from JSON Schemas.

Procedure

1. Create JSON Schemas to describe the format of the request and response JSON data.
2. Run the JSON web services assistant to create a WSBind file and high-level language structures.
3. Write a CICS program using the generated high-level language structures that implements the business logic for the JSON web service.
4. Configure CICS Transaction Gateway with a HTTP or HTTPS protocol handler and a JSON web service using the generated WSBind file.
5. Start CICS Transaction Gateway and test that you can invoke the JSON web service.

The JSON assistant

Run the CICS TG JSON web services assistant by customizing the `CTGJS2LS.txt` sample parameter file, or creating your own parameter file using `MAPPING-MODE=JS2LS` with `JSON-SCHEMA-REQUEST` and `JSON-SCHEMA-RESPONSE` and running **`ctgassist <parameter file>`**. The assistant generates WSBind file and high-level language structures from the JSON Schemas provided. `CTGJS2LS.txt` contains the required parameters for creating a Request-Response JSON web service WSBind file from JSON Schemas. Consider these options when you are creating a new application for web services:

- Which mechanism will CICS TG use to pass data to the CICS program? You can use a channel and pass the data in containers or use a COMMAREA. Specify them with the **`PGMINT`** parameter.

- Which language do you want to generate? The assistant can generate COBOL, C/C++, or PL/I language data structures. Specify the language with the **LANG** parameter.

For more information, see "Creating a Request-Response WSBind file from JSON Schemas" on page 45.

Writing the CICS program

If the program interface is `COMMAREA`, the CICS program can be written in the same way as any other COMMAREA-based program. The data that is passed in the COMMAREA conforms to the language structure generated by the JSON web services assistant.

If the program interface is `channel`, the data is passed in a single BIT container, which is named `DFHWS-DATA` by default. To use a container with a different name when you run the JSON web services assistant, specify the **CONTID** parameter. The data in the container conforms to the language structure generated by the JSON web services assistant.

The data is aligned differently depending on the value of TARGET-CICS-PLATFORM, for more information, see "Creating a Request-Response WSBind file from JSON Schemas" on page 45.

Creating a RESTful JSON web service

How to create a RESTful JSON web service.

Procedure

1. Create a JSON Schema to describe the format of the RESTful JSON data.
2. Run the JSON web services assistant to create a WSBind file and high-level language structure.
3. Write a CICS program using the generated high-level language structure that implements the business logic for the JSON web service.
4. Configure CICS Transaction Gateway with a HTTP or HTTPS protocol handler and a JSON web service using the generated WSBind file.
5. Start CICS Transaction Gateway and test that you can invoke the JSON web service.

The JSON assistant

Run the CICS TG JSON web services assistant by customizing `CTGJS2R.txt` sample parameter file, or create your own parameter file using `MAPPING-MODE=JS2LS` with `JSON-SCHEMA-RESTFUL` and running **`ctgassist <parameter file>`**. The assistant generates a WSBind file and high-level language structure from the JSON Schemas provided. Consider these options when you are creating a new application for web services:

- Which HTTP methods will the web service support? By default, the GET, POST, PUT, and DELETE methods are all enabled, but you might not want to support all of these methods, or you might want to also support the HEAD method. Specify which methods are enabled with the **HTTP-METHODS** parameter.
- Which language do you want to generate? The assistant can generate COBOL, C/C++, or PL/I language data structures. Specify the language with the **LANG** parameter.

For more information, see “Creating a RESTful WSBind file” on page 51.

Writing the CICS program

For all HTTP methods except DELETE, data is passed to and returned from the CICS program in a single BIT container, which is named `DFHWS-DATA` by default. To use a container with a different name, specify the **CONTID** parameter when you run the assistant. The data in the container conforms to the language structure generated by the assistant.

In addition to the data container, the following CHAR containers are passed to the CICS program:

Table 3. CHAR containers passed to the CICS program

Container name	Description
DFHHTTPMETHOD	The HTTP method that is used to call the web service, such as GET, POST, PUT, DELETE, or HEAD. The value is padded with spaces to a length of 8 characters.
DFHWS-URIMAPPATH	The URI property of the CICS TG web service definition that was matched.
DFHWS-URI	The path of the URI used to call the web service.
DFHWS-URI-QUERY	The query string of the URI used to call the web service.
DFHWS-URI-RESID	The resource ID of the URI used to call the web service. The resource ID is the portion of the URI that was matched by the wildcard character.

The CICS program can use the following CHAR containers to report errors to the client application by setting the HTTP status code, and to return non-JSON responses:

Table 4. CHAR containers received from the CICS program

Container name	Description
DFHHTTPSTATUS	Specifies the HTTP status code to return to the web service client. The content of the container must be the same as the initial status line of an HTTP response message, which has the following structure: `HTTP/1.1 nnn tttttttt` where `nnn` specifies the three-digit decimal HTTP status code and `tttttttt` specifies the human-readable status text that is associated with the status code.
DFHRESPONSE	Specifies the content of the HTTP response that is returned to the web service client.

Table 4. CHAR containers received from the CICS program (continued)

Container name	Description
DFHMEDIATYPE	Specifies the content type of the data in the DFHRESPONSE container. The content of the container must consist of a type and subtype that are separated by a slash character. For example: text/plain image/svg+xml

When the GET or HEAD method is used to call the web service, no data is passed to the CICS program. The program must update the language structure with data associated with the resource specified in the DFHWS-URI-RESID container, then write the language structure into the data container. To report an error, the program must use the DFHHTTPSTATUS container to set the HTTP status code.

When the POST or PUT methods are used to call the web service, data is passed to the CICS program in the data container. The program must read the language structure from the data container and use the contents to create a new resource (POST) or update an existing resource (PUT). To return a response to the client, such as the identifier or URI of the new resource, the program must use the DFHRESPONSE and DFHMEDIATYPE containers. To report an error, the program must use the DFHHTTPSTATUS container to set the HTTP status code. When the DELETE method is used to call the web service, the data container is not used. The program deletes the resource that is specified in the DFHWS-URI-RESID container. To report an error, the program uses the DFHHTTPSTATUS container to set the HTTP status code.

The data is aligned differently depending on the value of TARGET-CICS-PLATFORM, for more information, see "Creating a Request-Response WSBind file from JSON Schemas" on page 45.

The JSON web services assistant

The JSON web services assistant creates a mapping between JSON Schemas and language structures.

Before you create a JSON web service, you must have either:
- copybooks, header files and/or channel description documents which describe the interface to the CICS program, or
- JSON Schema files describing the interface to the JSON web service.

The JSON web services assistant is a supplied utility that helps you to create the necessary artifacts for a new JSON web service. These artifacts include WSBind files, language structures and JSON schemas.

A WSBind file describes which CICS program to call when a JSON web service is invoked and how JSON is converted to a channel or COMMAREA payload.

The JSON web services assistant is run using the executable script **ctgassist**. The assistant is located in the directory <install_path>/bin or in the CICS TG SDK directory cicstgsdk/webservices/assistant. **ctgassist** requires Java 7 or later to be available.

The JSON web services assistant can create a JSON Schema from a high-level language structure, or a high-level language structure from an existing JSON Schema. It supports COBOL, C/C++, and PL/I. It also generates information to enable runtime transformation of the JSON data to containers and COMMAREAs, and vice versa.

CICS TG is compatible with WSBind files generated by the CICS Transaction Server or IBM Rational Application Developer for IBM z Systems™ versions of the JSON web services assistant. See their documentation for guidance on how to use those interfaces. WSBind files that are generated by the CICS Transaction Gateway JSON web services assistant are compatible with CICS Transaction Server versions that support CICS JSON assistant MINIMUM-RUNTIME-LEVEL=4.0. However, if the CCSID parameter is specified with a non-EBCDIC value, then the WSBind file cannot be used in CICS Transaction Server.

Creating a Request-Response WSBind file from a language structure

The JSON web services assistant generates a WSBind file and JSON Schemas from high-level language data structures when using parameter `MAPPING-MODE=LS2JS`.

The executable script **`ctgassist`** is provided to run the JSON web services assistant. The script takes a single parameter which is the name of a file containing `<name>=<value>` pairs. A sample parameter file, `CTGLS2JS.txt` is provided in the `<install path>/samples/webservices` directory.

`CTGLS2JS.txt` contains the required parameters for creating a Request-Response JSON web service WSBind file from high-level language structures.

The temporary workspace

The JSON web services assistant requires a temporary workspace, so it uses the directory value that is specified in the *TMPDIR* environment variable. If *TMPDIR* is not specified the default `/tmp` is used. If *TMPDIR* is specified, it must be the location of a directory, and the user ID used to run the script must have read and write permission to this directory.

Note: The JSON web services assistant does not lock access to the files. Therefore, if two or more instances of JSON web services assistant run concurrently, and use the same temporary workspace files, nothing prevents one script from overwriting the workspace files while another script is using them, leading to unpredictable failures. You should devise a naming convention, and operating procedures, that avoid this situation.

Parameter use

- LS-REQUEST and LS-RESPONSE must be defined, unless REQUEST-CHANNEL and RESPONSE-CHANNEL are defined. These pairs are mutually exclusive.
- You can specify the input parameters in any order.
- Each parameter must start on a new line.
- There must be no blanks or other whitespace characters between `<name>` and `<value>`, only the `=` symbol.
- If a parameter is too long to fit on a single line, use a backslash (\) character at the end of the line to indicate that the parameter continues on the next line. Everything, including spaces before the backslash is considered part of the parameter. For example:

```
WSBIND=wsbinddir\
/app1
```

is equivalent to

```
WSBIND=wsbinddir/app1
```

Parameter descriptions

CCSID=*value*
Specifies the CCSID that is used at run time to encode character data in the application data structure. The default value is EBCDIC CCSID 037 when **TARGET-CICS-PLATFORM** is zOS, IBM-i, VSE, or not specified, otherwise it is ASCII CCSID 437.

If a value is specified, it must be supported by Java. For more information, see *IBM z/OS Unicode Services User's Guide and Reference* and *CICS Transaction Gateway: Programming Reference*.

If the CICS application program specified on the **PGMNAME** parameter is defined in the CICS conversion table DFHCNV, then CCSID should be set to the value specified on the **CLIENTCP** parameter in the DFHCNV entry.

CHAR-OCCURS={**STRING**|**ARRAY**}
Specifies how character arrays in the language structure are mapped. For example, PIC X OCCURS 20. This parameter is only for use by COBOL language.

ARRAY
Character arrays are mapped to a JSON array so that every character is mapped as an individual JSON element.

STRING
Character arrays are mapped to a JSON string so that the entire COBOL array is mapped as a single JSON element.

CHAR-USAGE=**NATIONAL**|**DBCS**
In COBOL, the national data type, PIC N, can be used for UTF-16 or DBCS data. This setting is controlled by the NSYMBOL compiler option. You must set the CHAR-USAGE parameter on the assistant to the same value as the NSYMBOL compiler option to ensure that the data is handled appropriately. This parameter is typically set to CHAR-USAGE=NATIONAL when you use UTF-16.

DBCS Data from PIC (n) fields is treated as UTF-16 encoded data.

NATIONAL
Data from PIC (n) fields is treated as DBCS encoded data.

CHAR-VARYING={**NO**|**NULL**|**COLLAPSE**|**BINARY**}
Specifies how character fields in the language structure are mapped. A character field in COBOL is a Picture clause of type X, for example PIC(X) 10; a character field in C/C++ is a character array. You can select these options:

NO Character fields are mapped to a JSON string and are processed as fixed-length fields. The maximum length of the data is equal to the length of the field.

This value does not apply to Enterprise and Other PL/I language structures.

NULL Character fields are mapped to a JSON string and are processed as null-terminated strings. CICS TG adds a terminating null character when transforming from a JSON message. The maximum length of the

character string is calculated as one character less than the length indicated in the language structure. NULL is the default value for the **CHAR-VARYING** parameter for C/C++.

This value does not apply to Enterprise and Other PL/I language structures.

COLLAPSE
Character fields are mapped to a JSON string. Trailing white space in the field is not included in the JSON message. The inbound JSON message is parsed to remove all leading, trailing, and embedded white space. For COBOL and PL/I, the default is `COLLAPSE`.

BINARY
Character fields are mapped to a JSON string that contains `base64` encoded data and are processed as fixed-length fields.

CONTID=*value*
In a service provider, specifies the name of the container that holds the top-level data structure that is used to represent a JSON message.

The length of the container that CICS TG passes to the target application program is the greater of the lengths of the request container and the response container.

DATA-TRUNCATION={**DISABLED**|**ENABLED**}
Specifies if variable length data is tolerated in a fixed-length field structure:

DISABLED
If the data is less than the fixed length that CICS TG is expecting, then CICS TG rejects the truncated data and issues an error message.

ENABLED
If the data is less than the fixed length that CICS TG is expecting, then CICS TG tolerates the truncated data and processes the missing data as null values.

DATETIME={**UNUSED**|**PACKED15**}
Specifies if potential ABSTIME fields in the high-level language structure are mapped as time stamps:

PACKED15
Packed decimal fields of length 15 (8 bytes) are treated as CICS ABSTIME fields, and mapped as timestamps.

UNUSED
Packed decimal fields of length 15 (8 bytes) are not treated as timestamps.

JSON-SCHEMA-REQUEST=*value*
This parameter is mandatory.

The value indicates the filename where the request JSON Schema is stored.

JSON-SCHEMA-RESPONSE=*value*
This parameter is mandatory.

The value indicates the filename where the response JSON Schema is stored.

LANG=**COBOL**
Specifies that the programming language of the high-level language structure is COBOL.

LANG=**PLI-ENTERPRISE**
Specifies that the programming language of the high-level language structure is Enterprise PL/I.

LANG=**PLI-OTHER**
Specifies that the programming language of the high-level language structure is a level of PL/I other than Enterprise PL/I.

LANG=**C**
Specifies that the programming language of the high-level language structure is C.

LANG=**CPP**
Specifies that the programming language of the high-level language structure is C++.

LOGFILE=*value*
The fully qualified path of the file into which the JSON web services assistant writes its activity log and trace information. The JSON web services assistant creates the file, but not the directory structure, if it does not exist.

Typically, you do not use this file, but it might be requested by the IBM service organization if you encounter problems with the JSON web services assistant.

LS-CODEPAGE=*value*
Specifies the code page that is used for the files specified in the **LS-REQUEST** and **LS-RESPONSE** parameters, where *value* is a CCSID number or a Java code page number. If this parameter is not specified, the current code page is used.

For example, **LS-CODEPAGE**=037.

LS-REQUEST=*value*
Specifies a filename that the JSON web services assistant uses to generate the names of the files that contain the high-level language structures for the web service request, which is the input data to the application program.

This is a mandatory parameter, unless you specify **REQUEST-CHANNEL** and **RESPONSE-CHANNEL**.
If you specify **LS-REQUEST**, you must also specify **LS-RESPONSE**. The value that is specified must be different to the value specified on the **LS-RESPONSE** parameter.

LS-RESPONSE=*value*
Specifies a filename that the JSON web services assistant uses to generate the names of the files that contain the high-level language structures for the web service response, which is the output data from the application program.

This is a mandatory parameter, unless you specify **REQUEST-CHANNEL** and **RESPONSE-CHANNEL**..
If you specify **LS-RESPONSE** you must also specify **LS-REQUEST**. The value that is specified must not be the same as the value specified on the **LS-REQUEST** parameter.

MAPPING-MODE=**LS2JS**
This parameter is mandatory.

Indicates mapping from language structure to JSON.

PGMINT={**CHANNEL**|**COMMAREA**}
For a service provider, specifies how CICS TG passes data to the target application program:

CHANNEL
CICS TG uses a channel interface to pass data to the target application program.

- The channel can contain multiple containers. Use the **REQUEST-CHANNEL** and **RESPONSE-CHANNEL** parameters. Do not specify **LS-REQUEST**, or **LS-RESPONSE**.

COMMAREA
CICS TG uses a communication area to pass data to the target application program.

When the target application program has processed the request, it must use the same mechanism to return the response. If the request was received in a communication area, then the response must be returned in the communication area; if the request was received in a container, the response must be returned in a container. The length of the communication area or container that CICS TG passes to the target application program is the greater of the lengths of the request communication area or container and the response communication area or container.

PGMNAME=*value*
Specifies the name of the CICS PROGRAM resource for the target application program that is exposed as a web service.

REQUEST-CHANNEL=*value*
This parameter is optional but if you do not specify **REQUEST-CHANNEL** and **RESPONSE-CHANNEL**, then you must specify **LS-REQUEST** and **LS_RESPONSE**. If you do specify **REQUEST-CHANNEL** and **RESPONSE-CHANNEL**, then you cannot specify **LS-REQUEST** and **LS_RESPONSE**.

Specifies the name and location of a channel description document. The channel description describes the containers that the web service provider application can use in its interface when it receives a JSON message from a web service requester. The channel description is a JSON document that must conform to the CICS supplied channel schema.

RESPONSE-CHANNEL=*value*
This parameter is optional but if you do not specify **REQUEST-CHANNEL** and **RESPONSE-CHANNEL**, then you must specify **LS-REQUEST** and **LS_RESPONSE**. If you do specify **REQUEST-CHANNEL** and **RESPONSE-CHANNEL**, then you cannot specify **LS-REQUEST** and **LS_RESPONSE**.

Specifies the name and location of a channel description document. The channel description describes the containers that the web service provider application can use in its interface when it sends a JSON response message to a web service requester. The channel description is a JSON document that must conform to the CICS supplied channel schema.

STRUCTURE=(*request*,*response*)
For C and C++ only, specifies the names of the high-level structures that are contained in the files that are specified in the **LS-REQUEST** and **LS-RESPONSE** parameters:

request
Specifies the name of the high-level structure that contains the request when the **LS-REQUEST** parameter is specified. The default value is **DFHREQUEST**.

If you specify a value, the file must contain a high-level structure with the name that you specify or a structure named **LS-RESPONSE** if you do not specify a name.

response
Specifies the name of the high-level structure that contains the response when the **LS-RESPONSE** parameter is specified. The default value is **DFHRESPONSE**.

If you specify a value, the file must contain a high-level structure with the name that you specify or a structure named **LS-RESPONSE** if you do not specify a name.

SYNCONRETURN={**NO**|**YES**}
Specifies whether the remote web service can issue a sync point.

NO The remote web service cannot issue a sync point. This value is the default. If the remote web service issues a sync point, it fails with an ADPL abend.

YES The remote web service can issue a sync point. If you select YES, the remote task is committed as a separate unit of work when control returns from the remote web service. If the remote web service updates a recoverable resource and a failure occurs after it returns, the update to that resource cannot be backed out.

TARGET-CICS-PLATFORM={**zOS**|**AIX**|**HP-UX**|**Solaris**|**IBM-i**|**VSE**|**LinuxI**|**Windows**}
Specifies the platform for the CICS server that receives requests from this web service.

zOS CICS Transaction Server for IBM z/OS

AIX IBM TXSeries® for Multiplatforms on IBM AIX®

HP-UX
IBM TXSeries for Multiplatforms on HP-UX

Solaris
IBM TXSeries for Multiplatforms on Solaris

IBM-i CICS Transaction Server for i

VSE CICS Transaction Server for VSE/ESA

LinuxI
IBM TXSeries for Multiplatforms on Intel Linux

Windows
IBM TXSeries for Multiplatforms on Microsoft Windows

For IBM z/OS, data is aligned on the IBM z/OS default boundaries. For all other platforms data is aligned on natural boundaries:

Table 5. Natural boundaries

Data type	Storage (bytes)	Alignment (bytes)
Short	2	2
Int	4	4
Long	8	8
Float	4	4
Double	8	8

TRANSACTION=*name*
Specifies the 1-4 character name of a transaction identifier that is passed in the EIBTRNID field of the exec interface block (EIB). If the web service is configured with the **defaultmirror** property set to No, the value of this parameter also

specifies the name of the mirror transaction which is attached when the CICS program is called. The value of this parameter can be overridden by the **transactionid** property in the CICS TG configuration file.

If you specify **TRANSACTION** you must also specify **URI**.

Acceptable characters:
A-Z a-z 0-9 $ @ # _ < >

URI=*value*
Specifies the relative URI that a client uses to access the web service. The value of this parameter can be overridden by the **uri** property in the CICS TG configuration file.

The URI consists of a path and an optional query string. If the URI path ends with *, the web service matches any request URI that starts with the specified path. The trailing * character is not considered part of the URI to match. If the URI contains a query string, the web service matches requests for the URI that contains all of the specified query string parameters in any order.

USERID=*id*
Specifies a 1-8 character user ID. The mirror transaction is attached under this user ID when the CICS program is called if the web service client does not provide user credentials.

Acceptable characters:
A-Z a-z 0-9 $ @ #

If you specify **USERID** you must also specify **URI**.

WSBIND=*value*
The fully qualified path of the web service binding file. The JSON web services assistant creates the file, but not the directory structure, if it does not exist. The file extension is `.wsbind`.

Creating a channel description document

Create a channel description document when your service provider application uses a channel interface with many containers.

About this task

Use an XML editor to create the channel description document. The schema for the channel description is called `channel.xsd` and is installed in the `<install_path>/docs` directory.

Procedure

1. Create an XML document with a <channel> element and the CICS channel namespace:
```
<channel name="myChannel" xmlns="http://www.ibm.com/xmlns/prod/CICS/channel">
</channel>
```
2. Add a <container> element for every container that the application program interface uses on the channel. You must use name, type and use attributes to describe each container. The following example shows six containers with different attribute values:

```
<container name="cont1" type="char" use="required"/>
<container name="cont2" type="char" use="optional"/>
<container name="cont3" type="bit" use="required"/>
<container name="cont4" type="bit" use="optional"/>
<container name="cont5" type="bit" use="required">
    <structure location="/u/userid/code/member.copybook"/>
</container>
<container name="cont6" type="bit" use="optional">
    <structure location="/u/userid/code/member2.copybook"/>
</container>
```

The structure element indicates that the content is defined in a language structure located in a partitioned data set member.

3. Save the XML document on the local file system.

Channel schema

The channel description document must conform to the following schema:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.ibm.com/xmlns/prod/CICS/channel"
   xmlns:tns="http://www.ibm.com/xmlns/prod/CICS/channel" elementFormDefault="qualified">
   <element name="channel"> 1
      <complexType>
         <sequence>
            <element name="container" maxOccurs="unbounded" "unbounded" minOccurs="0"> 2
               <complexType>
                  <sequence>
                     <element name="structure" minOccurs="0"> 3
                       <complexType>
                           <attribute name="location" type="string" use="required"/>
                           <attribute name="structure" type="string" use="optional"/>
                       </complexType>
                     </element>
                  </sequence>
                  <attribute name="name" type="tns:name16Type" use="required"/>
                  <attribute name="type" type="tns:typeType" use="required"/>
                  <attribute name="use" type="tns:useType" use="required"/>
               </complexType>
             </element>
          </sequence>
          <attribute name="name" type="tns:name16Type" use="optional" />
       </complexType>
   </element>
  <simpleType name="name16Type">
      <restriction base="string">
         <maxLength value="16"/>
      </restriction>
  </simpleType>
  <simpleType name="typeType">
      <restriction base="string">
        <enumeration value="char"/>
        <enumeration value="bit"/>
      </restriction>
  </simpleType>
  <simpleType name="useType">
     <restriction base="string">
        <enumeration value="required"/>
        <enumeration value="optional"/>
     </restriction>
  </simpleType>
</schema>
```

Note:

1 This element represents a CICS channel.

2 This element represents a CICS container within the channel.

3 A structure can only be used with `bit` mode containers. The `location` attribute indicates the location of a file that maps the contents of container. The `structure` attribute may be used in C and C++ to indicate the name of structure.

Creating a Request-Response WSBind file from JSON Schemas

The JSON web services assistant generates a Request-Response WSBind file and high-level language data structures from JSON Schemas when using parameter **MAPPING-MODE=JS2LS** with `JSON-SCHEMA-REQUEST` and `JSON-SCHEMA-RESPONSE`.

The executable script **ctgassist** is provided to run the JSON web services assistant. The script takes a single parameter which is the name of a file containing `<name>=<value>` pairs. There is a sample parameter file: `CTGJS2LS.txt` in the `<install path>/samples/webservices` directory.

The temporary workspace

The JSON web services assistant requires a temporary workspace, so it uses the directory value that is specified in the *TMPDIR* environment variable. If *TMPDIR* is not specified the default `/tmp` is used. If *TMPDIR* is specified, it must be the location of a directory and the user ID used to run the script must have read and write permission to this directory.

Note: The JSON web services assistant does not lock access to the IBM z/OS UNIX System Services files or the data set members. Therefore, if two or more instances of JSON web services assistant, run concurrently, and use the same temporary workspace files, nothing prevents one script from overwriting the workspace files while another script is using them, leading to unpredictable failures. You should devise a naming convention, and operating procedures, that avoid this situation.

Parameter use

- You can specify the input parameters in any order.
- Each parameter must start on a new line.
- There must be no blanks or other whitespace characters between `<name>` and `<value>`, only the = symbol.
- If a parameter is too long to fit on a single line, use a backslash (\) character at the end of the line to indicate that the parameter continues on the next line. Everything, including spaces before the backslash is considered part of the parameter. For example:

  ```
  WSBIND=wsbinddir\
  /app1
  ```

  is equivalent to

  ```
  WSBIND=wsbinddir/app1
  ```

Parameter descriptions

CCSID=*value*
: Specifies the CCSID that is used at run time to encode character data in the

application data structure. The default value is EBCDIC CCSID 037 when **TARGET-CICS-PLATFORM** is zOS, IBM-i, VSE, or not specified, otherwise it is ASCII CCSID 437.

If a value is specified, it must be supported by Java. For more information, see *IBM z/OS Unicode Services User's Guide and Reference* and *CICS Transaction Gateway: Programming Reference*.

If the CICS application program specified on the **PGMNAME** parameter is defined in the CICS conversion table DFHCNV, then set CCSID to the value specified on the **CLIENTCP** parameter in the DFHCNV entry.

CHAR-MULTIPLIER={**1**|*value*}
Specifies the number of bytes to allow for each character. The *value* of this parameter can be a positive integer in the range of 1 - 2,147,483,647. All nonnumeric character-based mappings, are subject to this multiplier. Binary, numeric, zoned, and packed decimal fields are not subject to this multiplier.

This parameter can be useful if, for example, you are planning to use DBCS characters where you might opt for a multiplier of 3 to allow space for potential shift-out and shift-in characters around every double-byte character at run time.

When you set **CCSID=1200** (indicating UTF-16), the only valid values for **CHAR-MULTIPLIER** are 2 or 4. When you use UTF-16, the default value is 2. Use **CHAR-MULTIPLIER=2** when you expect application data to contain characters that require 1 UTF-16 encoding unit. Use **CHAR-MULTIPLIER=4** when you expect application data to contain characters that require 2 UTF-16 encoding units.

Note: Setting **CHAR-MULTIPLIER** to 1 does not preclude the use of DBCS characters, and setting it to 2 does not preclude the use of UTF-16 surrogate pairs. However, if wide characters are routinely used then some valid values will not fit into the allocated field. If a larger **CHAR-MULTIPLIER** value is used, it can be possible to store more characters in the allocated field than are valid in the XML. Care must be taken to conform to the appropriate range restrictions.

CHAR-VARYING={**NO**|**NULL**|**YES**}
Specifies how variable-length character data is mapped. Variable-length binary data types are always mapped to either a container or a varying structure. If you do not specify this parameter, the default mapping depends on the language specified. You can select these options:

NO Variable-length character data is mapped as fixed-length strings.

NULL Variable-length character data is mapped to null-terminated strings.

YES Variable-length character data is mapped to a CHAR VARYING data type in PL/I. In the COBOL, C, and C++ languages, variable-length character data is mapped to an equivalent representation that comprises two related elements: data-length and the data.

CHAR-VARYING-LIMIT={**32767**|*value*}
Specifies the maximum size of binary data and variable-length character data that is mapped to the language structure. If the character or binary data is larger than the value specified in this parameter, it is mapped to a container and the container name is used in the generated language structure. The value can range from 0 to the default 32,767 bytes.

CONTID=*value*
In a service provider, specifies the name of the container that holds the top-level data structure that is used to represent a JSON message.

The length of the container that CICS TG passes to the target application program is the greater of the lengths of the request container and the response container.

DATA-TRUNCATION={**DISABLED**|**ENABLED**}
Specifies if variable length data is tolerated in a fixed-length field structure:

DISABLED
If the data is less than the fixed length that CICS TG is expecting, then CICS TG rejects the truncated data and issues an error message.

ENABLED
If the data is less than the fixed length that CICS TG is expecting, then CICS TG tolerates the truncated data and processes the missing data as null values.

DATETIME={**PACKED15**|**STRING**}
Specifies how JSON date-time elements are mapped to the language structure.

PACKED15
The default is that any JSON date-time element is processed as a time stamp and is mapped to CICS ABSTIME format.

STRING
The JSON date-time element is processed as text.

DEFAULT-CHAR-MAXLENGTH={**255**|*value*}
Specifies the default array length of character data in characters for mappings where no length is implied in the web service description document. The value of this parameter can be a positive integer in the range of 1 - 2,147,483,647.

INLINE-MAXOCCURS-LIMIT={**1**|*value*}
Specifies whether inline variable repeating content is used based on the `maxItems` JSON Schema keyword. Variably repeating content that is mapped inline is placed in the current container with the rest of the generated language structure. The variably repeating content is stored in two parts, as a counter that stores the number of occurrences of the data and as an array that stores each occurrence of the data. The alternative mapping for variably repeating content is container-based mapping, which stores the number of occurrences of the data and the name of the container where the data is placed. Storing the data in a separate container has performance implications that might make inline mapping preferable.

The value of **INLINE-MAXOCCURS-LIMIT** can be a positive integer in the range of 0 - 32,767. A value of `0` indicates that inline mapping is not used. A value of `1` ensures that optional elements are mapped inline. If the *value* of the `maxOccurs` attribute is greater than the *value* of **INLINE-MAXOCCURS-LIMIT**, container-based mapping is used; otherwise inline mapping is used.

When you decide whether you want variably repeating lists to be mapped inline, consider the length of a single item of recurring data. If few instances of long length occur, container-based mapping is preferable; if many instances of short length occur, inline mapping is preferable.

JSON-SCHEMA-REQUEST=*value*
This is a mandatory parameter.

The value indicates the fully qualified path name for the file where the request JSON Schema is stored.

JSON-SCHEMA-RESPONSE=*value*
This is a mandatory parameter.

The value indicates the fully qualified path name for the file where the response JSON Schema is stored.

LANG=**COBOL**
: Specifies that the programming language of the high-level language structure is COBOL.

LANG=**PLI-ENTERPRISE**
: Specifies that the programming language of the high-level language structure is Enterprise PL/I.

LANG=**PLI-OTHER**
: Specifies that the programming language of the high-level language structure is a level of PL/I other than Enterprise PL/I.

LANG=**C**
: Specifies that the programming language of the high-level language structure is C.

LANG=**CPP**
: Specifies that the programming language of the high-level language structure is C++.

LOGFILE=*value*
: The fully qualified path of the file into which the JSON web services assistant writes its activity log and trace information. If it does not exist, the JSON web services assistant creates the file, but not the directory structure.

  Typically, you do not use this file, but it might be requested by the IBM service organization if you encounter problems with JS2LS.

LS-CODEPAGE=*value*
: Specifies the code page that is used in the files that are specified in the **LS-REQUEST** and **LS-RESPONSE** parameters, where *value* is a CCSID number or a Java code page number. If this parameter is not specified, the current code page is used. For example, **LS-CODEPAGE**=037.

LS-REQUEST=*value*
: Specifies the fully qualified path name of the file that contains the high-level language structures for the web service request that is generated from the JSON Schema.

  This is a mandatory parameter.

  If you specify **LS-REQUEST** you must also specify **LS-RESPONSE**. The value that is specified must not be the same as the value specified on the **LS-RESPONSE** parameter.

LS-RESPONSE=*value*
: Specifies the fully qualified path name of the file that contains the high-level language structures for the web service response that is generated from the JSON Schema.

  This parameter is mandatory. If you specify **LS-RESPONSE** you must also specify **LS-REQUEST**.

MAPPING-MODE=**JS2LS**
: This is a mandatory parameter.

  Indicates mapping from JSON to the language structure.

MAPPING-OVERRIDES=**{UNDERSCORES-AS-HYPHENS|LESS-DUP-NAMES}**
: Specifies whether the default behavior is overridden for the specified mapping level when you generate language structures.

LESS-DUP-NAMES
: This parameter generates non-structural structure field names with `_value` at the end of the name to enable direct referencing to the field. For example, in the following PL/I language structure, when MAPPING-OVERRIDES=LESS-DUP-NAMES is specified, level 12 field streetName is suffixed with `_value`:

```
09 streetName,
    12 streetName                   CHAR(255) VARYING
   UNALIGNED,
    12 filler                       BIT (7),
    12 attr_nil_streetName_value    BIT (1),
```

The resulting structure is as follows:

```
09 streetName,
    12 streetName_value             CHAR(255) VARYING
   UNALIGNED,
    12 filler                       BIT (7),
    12 attr_nil_streetName_value    BIT (1),
```

UNDERSCORES-AS-HYPHENS
: For COBOL only. This parameter converts any underscores in the JSON Schema to hyphens, rather than the character X, to improve the readability of the generated COBOL language structures. If any field name clashes occur, the fields are numbered to ensure that they are unique. For more information, see "JSON Schema to COBOL mapping" on page 63.

NAME-TRUNCATION={RIGHT|LEFT}
: Specifies whether JSON names are truncated from the left or the right. The JSON web services assistant truncates JSON names to the appropriate length for the high-level language specified; by default names are truncated from the right.

PGMINT={**CHANNEL**|**COMMAREA**}
: For a service provider, specifies how CICS TG passes data to the target application program:

CHANNEL
: CICS TG uses a channel interface to pass data to the target application program.

COMMAREA
: CICS TG uses a communication area to pass data to the target application program.

When the target application program has processed the request, it must use the same mechanism to return the response. If the request was received in a communication area, then the response must be returned in the communication area; if the request was received in a container, the response must be returned in a container. The length of the communication area or container that CICS TG passes to the target application program is the greater of the lengths of the request communication area or container and the response communication area or container.

PGMNAME=*value*
: Specifies the name of a CICS PROGRAM resource.

When JS2LS is used to generate a web service binding file that is used in a service provider, you must supply this parameter. It specifies the resource name of the application program that is exposed as a web service.

STRUCTURE=(*request*,*response*)
For C and C++ only, specifies how the names of the request and response structures are generated.

The generated request and response structures are given names of *request01* and *response01*.
If one or both names are omitted, *request* is the value that is specified for the filename in **LS-REQUEST**, and *response* is the value that is specified for the filename in **LS-RESPONSE**.

SYNCONRETURN={**NO**|**YES**}
Specifies whether the remote web service can issue a sync point.

NO The remote web service cannot issue a sync point. This value is the default. If the remote web service issues a sync point, it fails with an ADPL abend.

YES The remote web service can issue a sync point. If you select YES, the remote task is committed as a separate unit of work when control returns from the remote web service. If the remote web service updates a recoverable resource and a failure occurs after it returns, the update to that resource cannot be backed out.

TARGET-CICS-PLATFORM={**zOS**|**AIX**|**HP-UX**|**Solaris**|**IBM-i**|**VSE**|**LinuxI**|**Windows**}
Specifies the platform for the CICS server that receives requests from this web service.

zOS CICS Transaction Server for IBM z/OS

AIX IBM TXSeries for Multiplatforms on IBM AIX

HP-UX
IBM TXSeries for Multiplatforms on HP-UX

Solaris
IBM TXSeries for Multiplatforms on Solaris

IBM-i CICS Transaction Server for i

VSE CICS Transaction Server for VSE/ESA

LinuxI
IBM TXSeries for Multiplatforms on Intel Linux

Windows
IBM TXSeries for Multiplatforms on Microsoft Windows

For IBM z/OS, data is aligned on the IBM z/OS default boundaries. For all other platforms data is aligned on natural boundaries:

Table 6. Natural boundaries

Data type	Storage (bytes)	Alignment (bytes)
Short	2	2
Int	4	4
Long	8	8
Float	4	4
Double	8	8

TRANSACTION=*name*
Specifies the 1-4 character name of a transaction identifier that is passed in the EIBTRNID field of the exec interface block (EIB). If the web service is configured

with the **defaultmirror** property set to `No`, the value of this parameter also specifies the name of the mirror transaction which is attached when the CICS program is called. The value of this parameter can be overridden by the **transactionid** property in the CICS TG configuration file.

If you specify **TRANSACTION** you must also specify **URI**.

Acceptable characters:
`A-Z a-z 0-9 $ @ # _ < >`

URI=*value*
Specifies the relative URI that a client uses to access the web service. The value of this parameter can be overridden by the **uri** property in the CICS TG configuration file.

The URI consists of a path and an optional query string. If the URI path ends with *, the web service matches any request URI that starts with the specified path. The trailing * character is not considered part of the URI to match. If the URI contains a query string, the web service matches requests for the URI that contains all of the specified query string parameters in any order.

USERID=*id*
Specifies a 1-8 character user ID. The mirror transaction is attached under this user ID when the CICS program is called if the web service client does not provide user credentials.

If you specify **USERID** you must also specify **URI**.

Acceptable characters:
`A-Z a-z 0-9 $ @ #`

WIDE-COMP3={**YES**|**NO**}
For COBOL only. Controls the maximum size of the packed decimal variable length in the COBOL language structure.

NO JS2LS limits the packed decimal variable length to 18 when you generate the COBOL language structure type COMP-3. If the packed decimal size is greater than 18, message DFHPI9022W is issued to indicate that the specified type is being restricted to a total of 18 digits.

YES JS2LS supports the maximum size of 31 when you generate the COBOL language structure type COMP-3.

WSBIND=*value*
The fully qualified path of the web service binding file. The JSON web services assistant creates the file, but not the directory structure, if it does not exist. The file extension is `.wsbind`.

Creating a RESTful WSBind file

The JSON web services assistant generates a RESTful WSBind file and high-level language data structures from a JSON Schema when using parameter **MAPPING-MODE=JS2LS** with `JSON-SCHEMA-RESTFUL`.

The executable script **ctgassist** is provided to run the JSON web services assistant. The script takes a single parameter which is the name of a file containing `<name>=<value>` pairs. A sample parameter file: `CTGJS2R.txt` is provided in the `<install path>/samples/webservices` directory.

The temporary workspace

The JSON web services assistant requires a temporary workspace, so it uses the directory value that is specified in the *TMPDIR* environment variable. If *TMPDIR* is not specified the default `/tmp` is used. If *TMPDIR* is specified, it must be the location of a directory, and the user ID used to run the script must have read and write permission to this directory.

Note: The JSON web services assistant does not lock access to the files. Therefore, if two or more instances of JSON web services assistant, run concurrently, and use the same temporary workspace files, nothing prevents one script from overwriting the workspace files while another script is using them, leading to unpredictable failures. You should devise a naming convention, and operating procedures, that avoid this situation.

Parameter use

- You can specify the input parameters in any order.
- Each parameter must start on a new line.
- There must be no blanks or other whitespace characters between `<name>` and `<value>`, only the `=` symbol.
- If a parameter is too long to fit on a single line, use a backslash (\) character at the end of the line to indicate that the parameter continues on the next line. Everything, including spaces before the backslash is considered part of the parameter. For example:

  ```
  WSBIND=wsbinddir\
  /app1
  ```

  is equivalent to

  ```
  WSBIND=wsbinddir/app1
  ```

Parameter descriptions

CCSID=*value*

Specifies the CCSID that is used at run time to encode character data in the application data structure. The default value is EBCDIC CCSID 037 when **TARGET-CICS-PLATFORM** is zOS, IBM-i, VSE, or not specified, otherwise it is ASCII CCSID 437.

If a value is specified, it must be supported by Java. For more information, see *IBM z/OS Unicode Services User's Guide and Reference* and *CICS Transaction Gateway: Programming Reference*.

If the CICS application program specified on the **PGMNAME** parameter is defined in the CICS conversion table `DFHCNV`, then CCSID should be set to the value specified on the **CLIENTCP** parameter in the `DFHCNV` entry.

CHAR-MULTIPLIER={**1**|*value*}

Specifies the number of bytes to allow for each character. The *value* of this parameter can be a positive integer in the range of 1 - 2,147,483,647. All nonnumeric character-based mappings, are subject to this multiplier. Binary, numeric, zoned, and packed decimal fields are not subject to this multiplier.

This parameter can be useful if, for example, you are planning to use DBCS characters where you might opt for a multiplier of 3 to allow space for potential shift-out and shift-in characters around every double-byte character at run time.

When you set **CCSID=1200** (indicating UTF-16), the only valid values for **CHAR-MULTIPLIER** are 2 or 4. When you use UTF-16, the default value is 2. Use **CHAR-MULTIPLIER=2** when you expect application data to contain characters that require 1 UTF-16 encoding unit. Use **CHAR-MULTIPLIER=4** when you expect application data to contain characters that require 2 UTF-16 encoding units.

Note: Setting **CHAR-MULTIPLIER** to 1 does not preclude the use of DBCS characters, and setting it to 2 does not preclude the use of UTF-16 surrogate pairs. However, if wide characters are routinely used then some valid values will not fit into the allocated field. If a larger **CHAR-MULTIPLIER** value is used, it can be possible to store more characters in the allocated field than are valid in the XML. Care must be taken to conform to the appropriate range restrictions.

CHAR-VARYING=NO|NULL|YES

Specifies how variable-length character data is mapped. Variable-length binary data types are always mapped to either a container or a varying structure. If you do not specify this parameter, the default mapping depends on the language specified. You can select these options:

NO Variable-length character data is mapped as fixed-length strings.

NULL Variable-length character data is mapped to null-terminated strings.

YES Variable-length character data is mapped to a CHAR VARYING data type in PL/I. In the COBOL, C, and C++ languages, variable-length character data is mapped to an equivalent representation that comprises two related elements: data-length and the data.

CHAR-VARYING-LIMIT=32767|*value*

Specifies the maximum size of binary data and variable-length character data that is mapped to the language structure. If the character or binary data is larger than the value specified in this parameter, it is mapped to a container and the container name is used in the generated language structure. The value can range from 0 to the default 32,767 bytes.

CONTID=*value*

In a service provider, specifies the name of the container that holds the top-level data structure that is used to represent a JSON message.

The length of the container that CICS TG passes to the target application program is the greater of the lengths of the request container and the response container.

DATA-TRUNCATION={DISABLED|ENABLED}

Specifies if variable length data is tolerated in a fixed-length field structure:

DISABLED

If the data is less than the fixed length that CICS TG is expecting, then CICS TG rejects the truncated data and issues an error message.

ENABLED

If the data is less than the fixed length that CICS TG is expecting, then CICS TG tolerates the truncated data and processes the missing data as null values.

DATETIME=PACKED15|STRING

Specifies how JSON date-time elements are mapped to the language structure.

PACKED15

The default is that any JSON date-time element is processed as a time stamp and is mapped to CICS ABSTIME format.

STRING
The JSON date-time element is processed as text.

DEFAULT-CHAR-MAXLENGTH=**255**|*value*
Specifies the default array length of character data in characters for mappings where no length is implied in the web service description document. The value of this parameter can be a positive integer in the range of 1 - 2,147,483,647.

HTTP-METHODS={*GET*|*POST*|*PUT*|*DELETE*|*HEAD*},{*GET*|*POST*|*PUT*|*DELETE*|*HEAD*},*
This is an optional parameter.

The default value is for GET, POST, PUT, and DELETE to be set, which tells JS2R that the application supports the four main RESTful operations.
If a value is provided, JS2R builds a WSBind file in which only the explicitly specified HTTP methods are accepted.
If an application wants to implement the HEAD method, it must deliberately opt-in to doing so. By default JS2R assumes that the application does not support incoming HTTP HEAD methods.
If a JSON client attempts to use an unsupported HTTP method, CICS TG responds with an HTTP 405 response.

INLINE-MAXOCCURS-LIMIT=**1**|*value*
Specifies whether inline variable repeating content is used based on the `maxItems` JSON Schema keyword. Variably repeating content that is mapped inline is placed in the current container with the rest of the generated language structure. The variably repeating content is stored in two parts, as a counter that stores the number of occurrences of the data and as an array that stores each occurrence of the data. The alternative mapping for variably repeating content is container-based mapping, which stores the number of occurrences of the data and the name of the container where the data is placed. Storing the data in a separate container has performance implications that might make inline mapping preferable.

The value of **INLINE-MAXOCCURS-LIMIT** can be a positive integer in the range of 0 - 32,767. A value of `0` indicates that inline mapping is not used. A value of `1` ensures that optional elements are mapped inline. If the *value* of the `maxOccurs` attribute is greater than the *value* of **INLINE-MAXOCCURS-LIMIT**, container-based mapping is used; otherwise inline mapping is used.

When you decide whether you want variably repeating lists to be mapped inline, consider the length of a single item of recurring data. If few instances of long length occur, container-based mapping is preferable; if many instances of short length occur, inline mapping is preferable.

JSON-SCHEMA-RESTFUL=*value*
This parameter is mandatory.

The value specifies the fully qualified path name for the file where the response JSON Schema is stored.

LANG=**COBOL**
Specifies that the programming language of the high-level language structure is COBOL.

LANG=**PLI-ENTERPRISE**
Specifies that the programming language of the high-level language structure is Enterprise PL/I.

LANG=**PLI-OTHER**
Specifies that the programming language of the high-level language structure is a level of PL/I other than Enterprise PL/I.

LANG=C
: Specifies that the programming language of the high-level language structure is C.

LANG=CPP
: Specifies that the programming language of the high-level language structure is C++.

LOGFILE=*value*
: The fully qualified path of the file into which JSON web services assistant writes its activity log and trace information. The JSON web services assistant creates the file, but not the directory structure, if it does not exist.

  Typically, you do not use this file, but it might be requested by the IBM service organization if you encounter problems with the JSON web services assistant.

LS-CODEPAGE=*value*
: Specifies the code page that is used in the files that are specified in the **LS-RESTFUL** parameter, where *value* is a CCSID number or a Java code page number. If this parameter is not specified, the current code page is used. For example, you might specify **LS-CODEPAGE**=037.

LS-RESTFUL
: Specifies the fully qualified path name for the file to contain the generated high-level language structures. For a service provider, the web service request is the input to the application program.

MAPPING-MODE=JS2LS
: This parameter is mandatory.

  Indicates mapping from JSON to the language structure.

MAPPING-OVERRIDES={UNDERSCORES-AS-HYPHENS|LESS-DUP-NAMES}
: Specifies whether the default behavior is overridden for the specified mapping level when the assistant generates language structures.

  LESS-DUP-NAMES
  : This parameter generates non-structural structure field names with `_value` at the end of the name to enable direct referencing to the field. For example, in the following PL/I language structure, when MAPPING-OVERRIDES=LESS-DUP-NAMES is specified, level 12 field streetName is suffixed with `_value`:

```
09 streetName,
    12 streetName                   CHAR(255) VARYING
   UNALIGNED,
    12 filler                       BIT (7),
    12 attr_nil_streetName_value    BIT (1),
```

  The resulting structure is as follows:

```
09 streetName,
    12 streetName_value             CHAR(255) VARYING
   UNALIGNED,
    12 filler                       BIT (7),
    12 attr_nil_streetName_value    BIT (1),
```

  UNDERSCORES-AS-HYPHENS
  : For COBOL only. This parameter converts any underscores in the JSON Schema to hyphens, rather than the character X, to improve the readability of the generated COBOL language structures. If any field name clashes occur, the fields are numbered to ensure that they are unique. For more information, see "JSON Schema to COBOL mapping" on page 63.

NAME-TRUNCATION={RIGHT|LEFT}
: Specifies whether JSON names are truncated from the left or the right. The web services assistant truncates JSON names to the appropriate length for the high-level language specified; by default names are truncated from the right.

PGMINT=CHANNEL|COMMAREA
: For a service provider, specifies how CICS TG passes data to the target application program:

CHANNEL
: CICS TG uses a channel interface to pass data to the target application program.

COMMAREA
: CICS TG uses a communication area to pass data to the target application program.

When the target application program has processed the request, it must use the same mechanism to return the response. If the request was received in a communication area, then the response must be returned in the communication area; if the request was received in a container, the response must be returned in a container. The length of the communication area or container that CICS TG passes to the target application program is the greater of the lengths of the request communication area or container and the response communication area or container.

PGMNAME=*value*
: Specifies the name of a CICS PROGRAM resource.

When JS2R is used to generate a web service binding file that is used in a service provider, you must supply this parameter. It specifies the resource name of the application program that is exposed as a web service.

STRUCTURE=*data*
: For C and C++ only, specifies how the name of the structure is generated.

The generated structure is given the name of *name01*.
If the name is omitted, the structure has the same name as the file generated from the **LS_RESTFUL** parameter that you specify.

SYNCONRETURN={NO|YES}
: Specifies whether the remote web service can issue a sync point.

NO
: The remote web service cannot issue a sync point. This value is the default. If the remote web service issues a sync point, it fails with an ADPL abend.

YES
: The remote web service can issue a sync point. If you select YES, the remote task is committed as a separate unit of work when control returns from the remote web service. If the remote web service updates a recoverable resource and a failure occurs after it returns, the update to that resource cannot be backed out.

TARGET-CICS-PLATFORM={zOS|AIX|HP-UX|Solaris|IBM-i|VSE|LinuxI|Windows}
: Specifies the platform for the CICS server that receives requests from this web service.

zOS
: CICS Transaction Server for IBM z/OS

AIX
: IBM TXSeries for Multiplatforms on IBM AIX

HP-UX
: IBM TXSeries for Multiplatforms on HP-UX

Solaris
: IBM TXSeries for Multiplatforms on Solaris

IBM-i CICS Transaction Server for i

VSE CICS Transaction Server for VSE/ESA

LinuxI
: IBM TXSeries for Multiplatforms on Intel Linux

Windows
: IBM TXSeries for Multiplatforms on Microsoft Windows

For IBM z/OS, data is aligned on the IBM z/OS default boundaries. For all other platforms data is aligned on natural boundaries:

Table 7. Natural boundaries

Data type	Storage (bytes)	Alignment (bytes)
Short	2	2
Int	4	4
Long	8	8
Float	4	4
Double	8	8

TRANSACTION=*name*
: Specifies the 1-4 character name of a transaction identifier that is passed in the `EIBTRNID` field of the exec interface block (EIB). If the web service is configured with the **defaultmirror** property set to `No`, the value of this parameter also specifies the name of the mirror transaction, which is attached when the CICS program is called. The value of this parameter can be overridden by the **transactionid** property in the CICS TG configuration file.

If you specify **TRANSACTION** you must also specify **URI**.

Acceptable characters:
`A-Z a-z 0-9 $ @ # _ < >`

URI=*value*
: Specifies the relative URI that a client uses to access the web service. The value of this parameter can be overridden by the **uri** property in the CICS TG configuration file.

The URI consists of a path and an optional query string. If the URI path ends with *, the web service matches any request URI that starts with the specified path. The trailing * character is not considered part of the URI to match. If the URI contains a query string, the web service matches requests for the URI that contains all of the specified query string parameters in any order.

USERID=*id*
: Specifies a 1-8 character user ID. The mirror transaction is attached under this user ID when the CICS program is called if the web service client does not provide user credentials.

If you specify **USERID** you must also specify **URI**.

Acceptable characters:
`A-Z a-z 0-9 $ @ #`

WIDE-COMP3=**YES**|**NO**
: For COBOL only. Controls the maximum size of the packed decimal variable length in the COBOL language structure.

NO JS2R limits the packed decimal variable length to 18 when it generates the COBOL language structure type `COMP-3`. If the packed decimal size is greater than 18, message DFHPI9022W is issued to indicate that the specified type is being restricted to a total of 18 digits.

YES JS2R supports the maximum size of 31 when it generates the COBOL language structure type COMP-3.

WSBIND=*value*
: The fully qualified path of the web service binding file. The JSON web services assistant creates the file, but not the directory structure, if it does not exist. The file extension defaults to `.wsbind`.

High-level language and JSON Schema mapping

Use the JSON web services assistant to generate mappings between high-level language structures and JSON Schemas.

Sample parameter files `CTGLS2JS.txt`, `CTGJS2LS.txt`, and `CTGJS2R.txt` are available to configure the JSON web services assistant.

- `CTGLS2JS.txt` maps high-level language structures to JSON Schemas for request-response services.
- `CTGJS2LS.txt` maps JSON Schemas to high-level language structures for request-response services.
- `CTGJS2R.txt` maps JSON Schemas to high-level language structures for RESTful services.

The mappings are not symmetrical:

- If you process a language data structure with the JSON web services assistant using `MAPPING-MODE=LS2JS`, and then process the resulting JSON Schema with the JSON web services assistant using `MAPPING-MODE=JS2LS`, do not expect the final data structure to be the same as the original.
- If you process a JSON Schema with the JSON web services assistant using `MAPPING-MODE=JS2LS`, and then process the resulting language structure with the JSON web services assistant using `MAPPING-MODE=LS2JS`, do not expect the final JSON Schema to be the same as the original.
- In some cases, `CTGJS2LS.txt` generates language structures that are not supported by `CTGLS2JS.txt`.

You must code high-level language structures that are processed by the JSON web services assistant according to the rules of the language, as implemented in the language compilers that CICS supports.

COBOL to JSON Schema mapping

The JSON web services assistant with parameters `MAPPING-MODE=LS2JS` and `LANG=COBOL` will map between COBOL data structures and JSON Schema definitions.

COBOL names are converted to JSON names according to the following rules:

- Duplicate names are made unique by the addition of one or more numeric digits.

For example, two instances of `year` become `year` and `year1`.

- Hyphens are replaced by underscores. Strings of contiguous hyphens are replaced by contiguous underscores.

  For example, `current-user--id` becomes `current_user__id`.
- Segments of names that are delimited by hyphens and that contain only uppercase characters are converted to lowercase.

  For example, `CA-REQUEST-ID` becomes `ca_request_id`.
- A leading underscore is added to names that start with a numeric character.

  For example, `9A-REQUEST-ID` becomes `_9a_request_id`.

The JSON web services assistant maps COBOL data description elements to schema elements according to the following table. COBOL data description elements that are not shown in the table are not supported by LS2JS. The following restrictions also apply:

- Data description items with level numbers of 66 and 77 are not supported. Data description items with a level number of 88 are ignored.
- The following clauses on data description entries are not supported:

  REDEFINES

  RENAMES; that is level 66

  DATE FORMAT
- The following clauses on data description items are ignored:

  BLANK WHEN ZERO

  JUSTIFIED

  VALUE
- The SIGN clauses SIGN TRAILING and SIGN LEADING are supported.
- SEPARATE CHARACTER is supported for both SIGN TRAILING and SIGN LEADING clauses.
- The following phrases on the USAGE clause are not supported:

  OBJECT REFERENCE

  POINTER

  FUNCTION-POINTER

  PROCEDURE-POINTER
- The following phrases on the USAGE clause are supported:

  COMPUTATIONAL-1

  COMPUTATIONAL-2
- The only PICTURE characters that are supported for DISPLAY and COMPUTATIONAL-5 data description items are 9, S, and Z.
- The PICTURE characters that are supported for PACKED-DECIMAL data description items are 9, S, V, and Z.
- The only PICTURE characters that are supported for edited numeric data description items are 9 and Z.

COBOL data description	JSON Schema definition

`PIC X(`*n*`)`
`PIC A(`*n*`)`
`PIC G(`*n*`) DISPLAY-1`
`PIC N(`*n*`)`
	`"type":"string",`
	`"maxLength":`*n*

COBOL data description	JSON Schema definition

`PIC S9 DISPLAY`
`PIC S99 DISPLAY`
`PIC S999 DISPLAY`
`PIC S9999 DISPLAY`
`PIC S9(n) DISPLAY`
`PIC S9(n) COMP`
`PIC S9(n) COMP-4`
`PIC S9(n) COMP-5`
`PIC S9(n) BINARY`
	`"type":"integer",`
	`"minimum":- (n + 1),`
	`"maximum":n`
	
	where *n* is the maximum value that can be represented by the pattern of '9' characters.

`PIC 9 DISPLAY`
`PIC 99 DISPLAY`
`PIC 999 DISPLAY`
`PIC 9999 DISPLAY`
`PIC 9(n) DISPLAY`
`PIC 9(n) COMP`
`PIC 9(n) COMP-4`
`PIC 9(n) COMP-5`
`PIC 9(n) BINARY`
	`"type":"integer",`
	`"minimum":0,`
	`"maximum":n`
	
	where *n* is the maximum value that can be represented by the pattern of '9' characters.

`PIC S9(m)V9(n) COMP-3`	`"type":"number",`
	`"description":"decimal",`
	`"minimum": x,`
	`"maximum": y,`
	`"multipleOf": z`
	
	where:
	
	x is the minimum value that can be represented by the pattern of '9' characters.
	
	y is the maximum value that can be represented by the pattern of '9' characters.
	
	z is the smallest unit available = $1 / 10^n$

`PIC 9(m)V9(n) COMP-3`	`"type":"number",`
	`"description":"decimal",`
	`"minimum": 0,`
	`"maximum": y,`
	`"multipleOf": z`
	
	where:
	
	y is the maximum value that can be represented by the pattern of '9' characters.
	
	z is the smallest unit available = $1 / 10^n$

`PIC S9(m) COMP-3`
 	
Supported when `DATETIME=PACKED15`
	`"type":"string",`
	`"format":"date-time"`
	
	The format of the time stamp is defined by RFC3339.

COBOL data description	JSON Schema definition

`PIC S9(m)V9(n) DISPLAY`	`"type":"number",`
	`"description":"decimal",`
	`"minimum": x,`
	`"maximum": y,`
	`"multipleOf": z`
	
	where:
	
	x is the minimum value that can be represented by the pattern of '9' characters.
	
	y is the maximum value that can be represented by the pattern of '9' characters.
	
	z is the smallest unit available = 1 / 10^n

`COMP-1` **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not exactly the same as the IEEE-754-1985 representation used for JSON. Some values might not convert exactly from one representation to the other. Some extremely large or small values might not be valid for float data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of COMP-1 data types with fixed precision alternatives.	`"type":"number",` `"description":"float"`
`COMP-2` **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not exactly the same as the IEEE-754-1985 representation used for JSON. Some values might not convert exactly from one representation to the other. Some extremely large or small values might not be valid for double data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of COMP-2 data types with fixed precision alternatives.	`"type":"number",` `"description":"double"`

COBOL data description	JSON Schema definition

data description OCCURS *n* TIMES	For primitives:
	"type":"array"
	"maxItems":*n*
	"minItems":*n*
	"items":{
	   "type":"object",
	   "properties":{
	      name:{
	         data description JSON
	      }
	   }
	   "required":[
	      name
	   ]
	}
	
	For data items:
	"type":"array"
	"maxItems":*n*
	"minItems":*n*
	"items":{
	  data description JSON
	}
	
	Where *data description JSON* is the JSON Schema representation of the COBOL *data description* and *name* is the name of the COBOL *data description*.

data description OCCURS *n* TO *m* TIMES DEPENDING ON *t*	*"field-name"*:{
	            "type":*"array"*,
	                  "maxItems":*m*
	                  "minItems":*n*
	                  "items":{
	                    ...
	                  }
	}
	
	The content of the array item depends on the datatype used.

COBOL data description	JSON Schema definition

PIC X OCCURS *n* TIMES
PIC A OCCURS *n* TIMES
PIC G DISPLAY-1 OCCURS *n* TIMES
PIC N OCCURS *n* TIMES
	When **CHAR-OCCURS**=STRING:
	"field-name":{
	"type":*"string"*,
	"maxLength":*n*
	}
	This is a string.
	When **CHAR-OCCURS**=ARRAY:
	"field-name":{
	"maxItems":*m*,
	"minItems":*n*,
	"items":{
	"type":*"object"*,
	"properties":{
	"field-name":{
	"type":*"string"*,
	"maxLength":1
	}
	},
	"required":[
	"field-name"
	]
	}
	}
	This is an array of single characters.

PIC X OCCURS *n* TO *m* TIMES
DEPENDING ON *t*
PIC A OCCURS *n* TO *m* TIMES
DEPENDING ON *t*
PIC G DISPLAY-1 OCCURS *n* TO *m* TIMES
DEPENDING ON *t*
PIC N OCCURS *n* TO *m* TIMES
DEPENDING ON *t*
	When **CHAR-OCCURS**=STRING:
	"field-name":{
	"type":*"string"*,
	"maxLength":*m*
	"minLength":*n*
	}

PIC N(*n*) USAGE NATIONAL When **CHAR-USAGE**=NATIONAL: PIC N(*n*)	"type":"string", "maxLength":*n* At run time, CICS TG populates the application data structure field with UTF-16 data.

Related reference:

"JSON Schema to COBOL mapping"
The JSON web services assistant with parameters MAPPING-MODE=JS2LS and LANG=COBOL will map between JSON Schema and COBOL data structures.

JSON Schema to COBOL mapping

The JSON web services assistant with parameters MAPPING-MODE=JS2LS and LANG=COBOL will map between JSON Schema and COBOL data structures.

The JSON web services assistant generates unique, valid field names for COBOL variables from the schema element names by using the following rules:

1. COBOL reserved words are prefixed with 'X'.

   For example, DISPLAY becomes XDISPLAY.
2. Characters other than A-Z, a-z, 0-9, or hyphen are replaced with 'X'.

   For example, monthly_total becomes monthlyXtotal.

3. If the last character is a hyphen, it is replaced with 'X'.

   For example, ca-request- becomes ca-requestX.

4. Duplicate names in the same scope are made unique by the addition of one or two numeric digits to the second and subsequent instances of the name.

   For example, three instances of year become year, year1, and year2.

5. A JSON Schema specifies that a variable has varying cardinality if it has a "type" value of "array", and the keywords "minItems" and "maxItems" are omitted or have different values. If the schema specifies that the variable has varying cardinality, then field names are created with suffixes of "_cont" and "_num".

   For more information, see "Variable arrays of elements with MAPPING-MODE=JS2LS" on page 85.

6. A JSON Schema specifies that a variable is optional if it does not appear in the "required" keyword array that is associated with the enclosing JSON Schema "object" type. For optional fields, an additional field is generated with a suffix of _num added to the element name. At run time this is zero to indicate the value was absent from the JSON data, and non-zero if the value was present in the JSON data.

7. Field names are limited to 28 characters. If a generated name, including the prefix and suffix, exceeds this length, the element name is truncated.

JS2LS maps schema types to COBOL data description elements according to the following table. Note that if the **CHAR-VARYING** parameter is set to YES, then variable-length character data is mapped to two related elements: a length field and a data field. For example:

```
"textString": {
   "type":"string",
   "maxLength":10000,
   "minLength":1
}
```

maps to:

```
15 textString-length    PIC S9999 COMP-5 SYNC
15 textString           PIC X(10000)
```

JSON Schema keyword	COBOL data description
All of: "type":"null" "type": [] "enum": [] "allOf" "anyOf" "noneOf" "not" "definitions"	Not supported.
"$schema":"http://json-schema.org/draft-04/schema#"	This keyword is ignored, but it is assumed to be compatible with the draft 04 JSON Schema specification.
"title": "same text" "description": "more text"	These keywords are ignored.
"format": "<predefined values>"	The "format" keyword is used to modify either the generated structure or runtime value. See the information later in this table for the supported use of "format".

JSON Schema keyword	COBOL data description
"type": "array", "items": {<JSON Sub-schema>}, "additionalItems": false, "maxItems": *m*, "minItems": *n*	The only form of JSON array currently supported is a repeated number of same type values. The <JSON Sub-schema> must define a supported "type", but that "type" cannot be "array". This is a restriction on the language structure generated. "additionalItems" is assumed to be false, and no other value is supported. If both "minItems" and "maxItems" are present, and they are equal, then the array is treated as fixed cardinality, otherwise it is treated as varying cardinality. See “Variable arrays of elements with MAPPING-MODE=JS2LS” on page 85.
"type": "array", "uniqueItems": true	"uniqueItems" is not supported with JSON arrays.
"type": "object", "additionalProperties": false, "properties": { ["<element name>": {<JSON Sub-schema>} [,]]* } "required": [["<element name>" [,]]*]	The only form of JSON object that is currently supported is a fixed set of named elements. This generates a structure (or sub-structure) that uses the element names. "additionalProperties" is assumed to be false, and no other value is supported. Any element in the "properties" object is considered"optional" if it is not in the "required" array or if no "required" array exists. An "optional" element is given a variable ordinality of zero to X; where X is either 1 or the maximum number of items in the array, where that item is defined as an array. See “Variable arrays of elements with MAPPING-MODE=JS2LS” on page 85.
"type": "object", "maxProperties": m, "minProperties": n, "patternProperties": {}, "dependencies":	None of these keywords are supported with JSON objects.
"type":"string" "maxLength":*m* "pattern": "regular expression>", "minLength": *l*	PIC X(*z*) where the value of *z* is based on *m*, but dependent on the settings of the *CHAR-VARYING* parameter. *m* is based on the "maxLength" keyword and treated as fixed length string. "pattern" and "minLength" restrictions are passed through to the language structure only as a comment.

JSON Schema keyword	COBOL data description
"type":"string" "maxLength":*m*	When CCSID=1200: PIC N(*z*) USAGE NATIONAL where the value of *z* is based on *m*, but dependent on the settings of the **CHAR-VARYING** parameter. *m* is based on the "maxLength" keyword and treated as fixed length string.

"*name*":{
"type":"string",
"format":"date-time"
}
	PIC S9(15) COMP-3
	
	All supported when DATETIME=PACKED15
	
	Note that "maxLength" and "minLength" are not supported for this format.

"*name*":{
"type":"string",
"format":"uri"
}
	PIC X(*m*)
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string.
	
	When **CCSID=1200**:
	
	PIC N(m) USAGE NATIONAL
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string.

"*name*":{
"type":"string",
"format":"base64Binary"
}
	PIC X(*m*)
	
	where *m* is based on the "maxLength" keyword

"*name*":{
"type":"string",
"format":"hexBinary"
}
	PIC X(*m*)
	
	where *m* is based on the "maxLength" keyword

"*name*":{
"type":"string",
"format":"<predefined>"
}
	PIC X(*m*)
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string, and where <predefined> is one of: *email*, *hostname*, *ipv4*, or *ipv6*. A relevant "pattern" is used and passed to the comment.
	
	When **CCSID=1200**:
	
	PIC N(*m*) USAGE NATIONAL
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string, and where <predefined> is one of: *email*, *hostname*, *ipv4*, or *ipv6*. A relevant "pattern" is used and passed to the comment.

JSON Schema keyword	COBOL data description
"type":"boolean"	PIC X DISPLAY The value x'00' implies false, x'01' implies true.
"type": "integer", "maxExclusive": true, "minExclusive": true, "multipleOf": n	"maxExclusive" and "minExclusive" restrictions are passed to the language structure only as a comment. "multipleOf" is ignored.
"type"="integer", minimum=0, maximum=255	PIC 9(*z*) COMP-5 SYNC or PIC 9(*z*) DISPLAY where $10^{(z-1)} < m <= 10^z$
"type":"integer", minimum:-128, maximum:127	PIC S9(*z*) COMP-5 SYNC or PIC S9(*z*) DISPLAY where $10^{(z-1)} < m <= 10^z$
"type":"integer", minimum:0, maximum;*m*	PIC 9(*z*) COMP-5 SYNC or PIC 9(*z*) DISPLAY where $10^{(z-1)} < m <= 10^z$
"type":"integer", minimum:-*m*, maximum:*m*-1	PIC S9(*z*) COMP-5 SYNC or PIC S9(*z*) DISPLAY where $10^{(z-1)} < m <= 10^z$
"type": "number", "maximum": m, "minimum": n, "maxExclusive": true, "minExclusive": true, "multipleOf": n	"maximum", "minimum", "maxExclusive" and "minExclusive" restrictions are passed to the language structure only as a comment. "multipleOf" is ignored.
"type":"number" "format":"decimal"	PIC 9(*p*)V9(*n*) COMP-3 where *p* and *n* are default values.

JSON Schema keyword	COBOL data description
`"type":"number"` `"format":"float"`	• `COMP-1` **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not exactly the same as the IEEE-754-1985 representation used for JSON. Some values might not convert exactly from one representation to the other. Some extremely large or small values might not be valid for float data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of COMP-1 data types with fixed precision alternatives.
`"type":"number"` `"format":"double"`	• `COMP-2` **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not exactly the same as the IEEE-754-1985 representation used for JSON. Some values might not convert exactly from one representation to the other. Some extremely large or small values might not be valid for double data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of COMP-2 data types with fixed precision alternatives.

Note: CICS TG cannot transform integer values greater than the maximum value for a signed long (2^63 - 1) unless they are enclosed within quotes.

Note: Minimum and maximum values specified in the schema for numeric types are used only to map to a COBOL datatype. Data is not validated against these values at run time.

Some of the schema types that are shown in the table map to a COBOL format of COMP-5 SYNC or of DISPLAY, depending on the values (if any) that are specified in the `minimum` and `maximum` keywords:

- For signed types (`short`, `int`, and `long`), DISPLAY is used when the following are specified:

  ```
  "maximum":a
  "minimum":-a
  ```

  where *a* is a string of '9's.
- For unsigned types (`unsignedShort`, `unsignedInt`, and `unsignedLong`), DISPLAY is used when the following are specified:

  ```
  "maximum":a
  "minimum":0
  ```

  where *a* is a string of '9's.

When any other value is specified, or no value is specified, COMP-5 SYNC is used.

Related reference:

"COBOL to JSON Schema mapping" on page 58
The JSON web services assistant with parameters MAPPING-MODE=LS2JS and LANG=COBOL will map between COBOL data structures and JSON Schema definitions.

C and C++ to JSON Schema mapping

The JSON web services assistant with parameters MAPPING-MODE=LS2JS and LANG=C or LANG=CPP will map between C or C++ data types and JSON Schema definitions.

C and C++ names are converted to JSON names according to the following rules:

1. Characters that are not valid in JSON property names are replaced with 'X'.

   For example, monthly-total becomes monthlyXtotal.
2. Duplicate names are made unique by the addition of one or more numeric digits.

   For example, two instances of year become year and year1.

LS2JS maps C and C++ data types to schema elements according to the following table. C and C++ types that are not shown in the table are not supported by LS2JS. The _Packed qualifier is supported for structures. These restrictions apply:

- Header files must contain a top level struct instance.
- You cannot declare a structure type that contains itself as a member.
- The following C and C++ data types are not supported:
  - decimal
  - long double
  - wchar_t (C++ only)
- The following are ignored if they are present in the header file.

  Storage class specifiers:
  - auto
  - register
  - static
  - extern
  - mutable

  Qualifiers
  - const
  - volatile
  - _Export (C++ only)

  Function specifiers
  - inline (C++ only)
  - virtual (C++ only)

  Initial values
- The header file must not contain these items:
  - Unions
  - Class declarations
  - Enumeration data types
  - Pointer type variables
  - Template declarations
  - Predefined macros; that is, macros with names that start and end with two underscore characters (__)
  - The line continuation sequence (a \ symbol that is immediately followed by a newline character)

Prototype function declarators

Preprocessor directives

Bit fields

The `__cdecl` (or `_cdecl`) keyword (C++ only)

- The application programmer must use a 32-bit compiler to ensure that an `int` maps to 4 bytes.
- The following C++ reserved keywords are not supported:

`explicit`

`using`

`namespace`

`typename`

`typeid`

C and C++ data type	**Schema simpleType**
`char[`*z*`]`	`"type":"string"` `"maxlength":`*z*
`char16_t[`*n*`]`	`"type":"string"` `"maxlength":`*n* At run time, CICS TG populates the application data structure field with UTF-16 data.

`char[8]`
 	
Supported when `DATETIME=PACKED15`
	`"type":"string"`
	`"format":"date-time"`
	
	The format of the timestamp is defined by RFC3339.

`char`
`short`
`int`
`long`
`long long`
	`"type":"integer",`
	`"minimum":- (`*n* `+ 1),`
	`"maximum":`*n*
	
	where *n* is the maximum value that can be represented by the primitive.

`unsigned char`
`unsigned short`
`unsigned int`
`unsigned long`
`unsigned long long`
	`"type":"integer",`
	`"minimum":0,`
	`"maximum":`*n*
	
	where *n* is the maximum value that can be represented by the primitive.

`bool` (C++ only)	`"type":"boolean"`

`float`	`"type":"number",`
	`"description":"float"`
	
	Note: The IBM Hexadecimal Floating Point (HFP) data representation is not exactly the same as the IEEE-754-1985 representation used for JSON. Some values may not convert exactly from one representation to the other. Some extremely large or small values might not be valid for float data types. Some values may lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of float data types with fixed precision alternatives.

C and C++ data type	Schema simpleType
double	"type":"number", "description":"double" **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not exactly the same as the IEEE-754-1985 representation used for JSON. Some values may not convert exactly from one representation to the other. Some extremely large or small values might not be valid for double data types. Some values may lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of double data types with fixed precision alternatives.

type name[*n*]	For primitives:
	"type":"array"
	"maxItems":*n*
	"minItems":*n*
	"items":{
	  "type":"object",
	  "properties":{
	    name:{
	      type JSON
	    }
	  }
	  "required":[
	    name
	  ]
	}
	
	For structs:
	"type":"array"
	"maxItems":*n*
	"minItems":*n*
	"items":{
	  type JSON
	}
	
	Where *type JSON* is the JSON Schema representation of the C or C++ type.

Related reference:

"JSON Schema to C and C++ mapping"
The JSON web services assistant with parameters `MAPPING-MODE=JS2LS` and `LANG=C` or `LANG=CPP` will map between the JSON Schemas and C or C++ data types.

JSON Schema to C and C++ mapping

The JSON web services assistant with parameters `MAPPING-MODE=JS2LS` and `LANG=C` or `LANG=CPP` will map between the JSON Schemas and C or C++ data types.

The JSON web services assistant generates unique, valid field names for C and C++ variables from the schema element names using the following rules:

1. Characters other than A-Z, a-z, 0-9, or _ are replaced with '`X`'.

   For example, `monthly-total` becomes `monthlyXtotal`.
2. If the first character is not an alphabetic character, it is replaced by a leading '`X`'.

   For example, `_monthlysummary` becomes `Xmonthlysummary`.
3. Duplicate names in the same scope are made unique by the addition of one or two numeric digits to the second and subsequent instances of the name.

   For example, three instances of `year` become `year`, `year1`, and `year2`.
4. A JSON Schema specifies that a variable has varying cardinality if it has a `"type"` value of `"array"`, and the keywords `"minItems"` and `"maxItems"` are

omitted or have different values. If the schema specifies that the variable has varying cardinality, then field names are created with suffixes of "_cont" and "_num".

For more information, see "Variable arrays of elements with MAPPING-MODE=JS2LS" on page 85.

5. A JSON Schema specifies that a variable is optional if it does not appear in the "required" keyword array that is associated with the enclosing JSON Schema "object" type. For optional fields, an additional field is generated with a suffix of _num added to the element name. At run time this is zero to indicate the value was absent from the JSON data, and non-zero if the value was present in the JSON data.
6. Field names are limited to 50 characters. If a generated name, including any prefix and suffix, exceeds this length, the element name is truncated.

JS2LS and JS2R map JSON Schema type values to C and C++ data types according to the following table. The following rules also apply:

JSON Schema keyword	C and C++ data type
All of: "type":"null" "type": [] "enum": [] "allOf" "anyOf" "noneOf" "not" "definitions"	Not supported
"$schema":"http://json-schema.org/draft-04/schema#"	This keyword is ignored, but it is assumed to be compatible with the draft 04 JSON Schema specification.
"title": "same text" "description": "more text"	These keywords are ignored.
"format": "<predefined values>"	The "format" keyword is used to modify either the generated structure or runtime value. See the following information for the supported use of "format".
"type": "array", "items": {<JSON Sub-schema>}, "additionalItems": false, "maxItems": *m*, "minItems": *n*	The only form of JSON array currently supported is a repeated number of same type values. The <JSON Sub-schema> must define a supported "type", but that "type" cannot be "array". This is a restriction on the language structure generated. "additionalItems" is assumed to be false, and no other value is supported. If both "minItems" and "maxItems" are present, and they are equal, then the array is treated as fixed cardinality, otherwise it is treated as varying cardinality. See "Variable arrays of elements with MAPPING-MODE=JS2LS" on page 85.

JSON Schema keyword	C and C++ data type
"type": "array", "uniqueItems": true	"uniqueItems" is not supported with JSON arrays. The <JSON Sub-schema> must define a supported "type", but that "type" cannot be "array". This is a restriction on the language structure generated.
"type": "object", "additionalProperties": false, "properties": { ["<element name>": {<JSON Sub-schema>} [,]]* } "required": [["<element name>" [,]]*]	The only form of JSON object currently supported is a fixed set of named elements. This will generate a structure (or sub-structure) using the element names. "additionalProperties" is assumed to be false, and no other value is supported. Any element in the "properties" object is considered"optional" if it is not in the "required" array or if no "required" array exists. An "optional" element is given a variable ordinality of zero to X; where X is either 1 or the maximum number of items in the array, where that item is defined as an array. See "Variable arrays of elements with MAPPING-MODE=JS2LS" on page 85.
"type": "object", "maxProperties": m, "minProperties": n, "patternProperties": {}, "dependencies":	None of these keywords are supported with JSON objects.
"type":"string" "maxLength":*m* "pattern": "regular expression>", "minLength": *l*	char[*z*] where the value of *z* is based on *m*, but dependent on the settings of the **CHAR-VARYING** parameter. *m* is based on the "maxLength" keyword and treated as fixed length string. "pattern" and "minLength" restrictions are passed through to the language structure only as a comment.
"type":"string" "maxLength":*m*	When CCSID=1200: char16_t[*z*] where the value of *z* is based on *m*, but dependent on the settings of the **CHAR-VARYING** parameter. *m* is based on the "maxLength" keyword and treated as fixed length string.

"*name*":{
"type":"string",
"format":"date-time"
}
	char[*8*]
	
	All supported when DATETIME=PACKED15

JSON Schema keyword	C and C++ data type

"*name*":{
"type":"string",
"format":"uri"
}
	char[*m*]
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string.
	
	When **CCSID=1200**:
	
	char16_t[*m*]
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string.

"*name*":{
"type":"string",
"format":"base64Binary"
}
	char[*m*]
	
	where *m* is based on the "maxLength" keyword.

"*name*":{
"type":"string",
"format":"hexBinary"
}
	char[*m*]
	
	where *m* is based on the "maxLength" keyword.

"*name*":{
"type":"string",
"format":"<predefined>"
}
	char[*m*]
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string, and where <predefined> is one of: *email*, *hostname*, *ipv4* or *ipv6*. A relevant "pattern" is used and passed to the comment.
	
	When **CCSID=1200**:
	
	char16_t[*m*]
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string, and where <predefined> is one of: *email*, *hostname*, *ipv4*, or *ipv6*. A relevant "pattern" is used and passed to the comment.

"type":"boolean"	bool (C++ only) short (C only)
"type": "integer", "maxExclusive": true, "minExclusive": true, "multipleOf": n	"maxExclusive" and "minExclusive" restrictions are passed to the language structure only as a comment. "multipleOf" is ignored.
"type":"integer", minimum:-128, maximum:127	signed char
"type":"integer", minimum:0, maximum:255	unsigned char
"type":"integer", minimum:-32768, maximum:32767	short

JSON Schema keyword	C and C++ data type

`"type":"integer",`
`minimum:0,`
`maximum:65535`
	`unsigned short`

`"type":"integer",`
`minimum:-2147483648,`
`maximum:2147483647`
	`int`

`"type":"integer",`
`minimum:0,`
`maximum:4294967295`
	`unsigned int`

`"type":"integer",`
`minimum:-9223372036854775808,`
`maximum:9223372036854775807`
	`long long`

`"type":"integer",`
`minimum:0,`
`maximum:18446744073709551615`
	`unsigned long long`

`"type": "number",`
`"maximum": m,`
`"minimum": n,`
`"maxExclusive": true,`
`"minExclusive": true,`
`"multipleOf": n`
	`"maximum"`, `"minimum"`, `"maxExclusive"` and `"minExclusive"` restrictions are passed to the language structure only as a comment.
	
	`"multipleOf"` is ignored.

`"type":"number"` `"format":"float"`	• `float(*)` **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not exactly the same as the IEEE-754-1985 representation used for JSON. Some values may not convert exactly from one representation to the other. Some extremely large or small values might not be valid for float data types. Some values may lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of float data types with fixed precision alternatives.
`"type":"number"` `"format":"double"`	• `double(*)` **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not exactly the same as the IEEE-754-1985 representation used for JSON. Some values may not convert exactly from one representation to the other. Some extremely large or small values might not be valid for double data types. Some values may lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of double data types with fixed precision alternatives.

Note: CICS cannot transform integer values greater than the maximum value for a signed long (2^63 - 1) unless they are enclosed within quotes.

Note: Minimum and maximum values specified in the schema for numeric types are used only to map to a C or C++ datatype. Data is not validated against these values at run time.

Related reference:

"C and C++ to JSON Schema mapping" on page 69
The JSON web services assistant with parameters `MAPPING-MODE=LS2JS` and `LANG=C` or `LANG=CPP` will map between C or C++ data types and JSON Schema definitions.

PL/I to JSON Schema mapping

The JSON web services assistant with parameters `MAPPING-MODE=LS2JS` and `LANG=PLI-ENTERPRISE` or `LANG=PLI-OTHER` will map between PL/I data structures and JSON Schema definitions. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: `PLI-ENTERPRISE`, and `PLI-OTHER`.

PL/I names are converted to JSON names according to the following rules:

1. Characters that are not valid in JSON property names are replaced with 'x'.

   For example, `monthly$total` becomes `monthlyxtotal`.
2. Duplicate names are made unique by the addition of one or more numeric digits.

   For example, two instances of `year` become `year` and `year1`.

LS2JS maps PL/I data types to schema elements according to the following table. PL/I types that are not shown in the table are not supported by LS2JS. The following restrictions also apply:

- Data items with the COMPLEX attribute are not supported.
- Data items with the FLOAT attribute are supported. Enterprise PL/I FLOAT IEEE is not supported.
- VARYING and VARYINGZ pure DBCS strings are supported.
- Data items that are specified as DECIMAL(p,q) are supported only when $p \geq q$.
- Data items that are specified as BINARY(p,q) are supported only when $q = 0$.
- If the PRECISION attribute is specified for a data item, it is ignored.
- PICTURE strings are not supported.
- ORDINAL data items are treated as FIXED BINARY(7) data types.

LS2JS does not fully implement the padding algorithms of PL/I; therefore, you must declare padding bytes explicitly in your data structure. LS2JS issues a message if it detects that padding bytes are missing. Each top-level structure must start on a doubleword boundary and each byte in the structure must be mapped to the correct boundary. Consider this code fragment:

```
3 FIELD1 FIXED BINARY(7),
3 FIELD2 FIXED BINARY(31),
3 FIELD3 FIXED BINARY(63);
```

In this example:
- FIELD1 is 1 byte long and can be aligned on any boundary.
- FIELD2 is 4 bytes long and must be aligned on a fullword boundary.
- FIELD3 is 8 bytes long and must be aligned on a doubleword boundary.

The Enterprise PL/I compiler aligns the fields in the following order:

1. FIELD3 is aligned first because it has the strongest boundary requirements.
2. FIELD2 is aligned at the fullword boundary immediately before FIELD3.

3. FIELD1 is aligned at the byte boundary immediately before FIELD3.

Finally, so that the entire structure is aligned at a fullword boundary, the compiler inserts three padding bytes immediately before FIELD1.

Because LS2JS does not insert equivalent padding bytes, you must declare them explicitly before the structure is processed by LS2JS. For example:

```
3 PAD1   FIXED BINARY(7),
3 PAD2   FIXED BINARY(7),
3 PAD3   FIXED BINARY(7),
3 FIELD1 FIXED BINARY(7),
3 FIELD2 FIXED BINARY(31),
3 FIELD3 FIXED BINARY(63);
```

Alternatively, you can change the structure to declare all the fields as unaligned and recompile the application that uses the structure. For more information about PL/I structural memory alignment requirements, see *Enterprise PL/I Language Reference*.

PL/I data description	JSON Schema definition

`FIXED BINARY (n)`	`"type":"integer",`
	`"minimum":- (n + 1),`
	`"maximum":n`
	
	where *n* is the maximum value that can be represented by the primitive.

`UNSIGNED FIXED BINARY(n)`
Restriction: Enterprise PL/I only.
	`"type":"integer",`
	`"minimum":0,`
	`"maximum":n`
	
	where *n* is the maximum value that can be represented by the primitive.

`FIXED DECIMAL(n,m)`	`"type":"number",`
	`"description":"decimal",`
	`"multipleOf":x,`
	`"maximum":y,`
	`"minimum":-z`
	
	where:
	
	x is the smallest unit available = 1 / 10^{m}.
	
	y is the maximum value that can be represented by the combination of *n* and *m*.
	
	z is the maximum value that can be represented by the combination of *n* and *m*.

`FIXED DECIMAL(15)`
 	
Supported when **DATETIME=PACKED15**.
	`"type":"string",`
	`"format":"date-time"`
	
	The format of the time stamp is defined by RFC3339.

`BIT(n)`
 	
where *n* is a multiple of 8. Other values are not supported.
	`"type":"string"`
	`"maxLength":m`
	
	where `m = n`/8.

PL/I data description	JSON Schema definition
CHARACTER(*n*) VARYING and VARYINGZ are also supported. **Restriction:** VARYINGZ is supported only by Enterprise PL/I.	"type":"string" "maxLength":*n*
GRAPHIC(*n*) VARYING and VARYINGZ are also supported. **Restriction:** VARYINGZ is supported only by Enterprise PL/I.	"type":"string" "maxLength":*n*
WIDECHAR(*n*) **Restriction:** Enterprise PL/I only.	"type":"string" "maxLength":*n* CICS TG populates the application data structure field with UTF-16 data: <xsd:simpleType> <xsd:restriction base="xsd:string"> <xsd:maxlength value="n"/> <xsd:whiteSpace value="preserve"/> </xsd:restriction> </xsd:simpleType>
ORDINAL **Restriction:** Enterprise PL/I only.	"type":"integer", "minimum":0, "maximum":255
BINARY FLOAT(*n*) where *n* <= 21 **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not the same as the IEEE-754-1985 representation that is used for JSON. Some values might not convert exactly from one representation to the other. Some large or small values might not be valid for float data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of BINARY FLOAT data types with fixed precision alternatives.	"type":"number", "description":"float"

PL/I data description	JSON Schema definition
`BINARY FLOAT(n)` where 21 < n <= 53 Values greater than 53 are not supported. **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not the same as the IEEE-754-1985 representation that is used for JSON. Some values might not convert exactly from one representation to the other. Some large or small values might not be valid for float data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of BINARY FLOAT data types with fixed precision alternatives.	`"type":"number",` `"description":"double"`
`DECIMAL FLOAT(n)`where n <= 6 **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not the same as the IEEE-754-1985 representation that is used for JSON. Some values might not convert exactly from one representation to the other. Some large or small values might not be valid for float data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of DECIMAL FLOAT data types with fixed precision alternatives.	`"type":"number",` `"description":"float"`
`DECIMAL FLOAT(n)`where 6 < n <= 16 Values greater than 16 are not supported. **Note:** The IBM Hexadecimal Floating Point (HFP) data representation is not the same as the IEEE-754-1985 representation that is used for JSON. Some values might not convert exactly from one representation to the other. Some large or small values might not be valid for float data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of DECIMAL FLOAT data types with fixed precision alternatives.	`"type":"number",` `"description":"double"`

<table>
<tr><th>PL/I data description</th><th>JSON Schema definition</th></tr>
<tr><td>name (n) data description</td><td>For primitives:
<pre>
"type":"array"
"maxItems":n
"minItems":n
"items":{
   "type":"object",
   "properties":{
      name:{
         data description JSON
      }
   }
   "required":[
     name
   ]
}
</pre>
For data declarations:
<pre>
"type":"array"
"maxItems":n
"minItems":n
"items":{
   data description JSON
}
</pre>
Where data description JSON is the JSON Schema representation of the PL/I data description.</td></tr>
</table>

Related reference:

"JSON Schema to PL/I mapping"
The JSON web services assistant with parameters `MAPPING-MODE=JS2LS` and `LANG=PLI-ENTERPRISE` or `LANG=PLI-OTHER` will map between JSON Schemas and PL/I data structures. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: `PLI-ENTERPRISE` and `PLI-OTHER`.

JSON Schema to PL/I mapping

The JSON web services assistant with parameters `MAPPING-MODE=JS2LS` and `LANG=PLI-ENTERPRISE` or `LANG=PLI-OTHER` will map between JSON Schemas and PL/I data structures. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: `PLI-ENTERPRISE` and `PLI-OTHER`.

The JSON web services assistant generates unique and valid names for PL/I variables from the schema element names using the following rules:

1. Characters other than A-Z, a-z, 0-9, @, #, or $ are replaced with `'X'`.

   For example, `monthly_total` becomes `monthlyXtotal`.
2. Duplicate names in the same scope are made unique by the addition of one or two numeric digits to the second and subsequent instances of the name.

   For example, three instances of `year` become `year`, `year1`, and `year2`.
3. A JSON Schema specifies that a variable has varying cardinality if it has a `"type"` value of `"array"`, and the keywords `"minItems"` and `"maxItems"` are omitted or have different values. If the schema specifies that the variable has varying cardinality, then field names are created with suffixes of `"_cont"` and `"_num"`.

   For more information, see "Variable arrays of elements with MAPPING-MODE=JS2LS" on page 85.

4. A JSON Schema specifies that a variable is optional if it does not appear in the "required" keyword array that is associated with the enclosing JSON Schema "object" type. For optional fields, an additional field is generated with a suffix of _num added to the element name. At run time this is zero to indicate the value was absent from the JSON data, and non-zero if the value was present in the JSON data.
5. Field names are limited to 31 characters. If a generated name, including any prefix and suffix, exceeds this length, the element name is truncated.

The total length of the resulting name is 31 characters or less.

JS2LS and JS2R map schema type values to PL/I data types according to the following table. Also note the following points:

JSON Schema keyword	PL/I data description
All of: "type":"null" "type": [] "enum": [] "allOf" "anyOf" "noneOf" "not" "definitions"	Not supported
"$schema":"http://json-schema.org/draft-04/schema#"	This keyword is ignored, but it is assumed to be compatible with the draft 04 JSON Schema specification.
"title": "same text" "description": "more text"	These keywords are ignored.
"format": "<predefined values>"	The "format" keyword is used to modify either the generated structure or run-time value. See the following information for the supported use of "format".
"type": "array", "items": {<JSON Sub-schema>}, "additionalItems": false, "maxItems": *m*, "minItems": *n*	The only form of JSON array currently supported is a repeated number of same type values. The <JSON Sub-schema> must define a supported "type", but that "type" cannot be "array". This is a restriction on the language structure generated. "additionalItems" is assumed to be false, and no other value is supported. If both "minItems" and "maxItems" are present, and they are equal, then the array is treated as fixed cardinality, otherwise it is treated as varying cardinality. See "Variable arrays of elements with MAPPING-MODE=JS2LS" on page 85.
"type": "array", "uniqueItems": true	"uniqueItems" is not supported with JSON arrays. The <JSON Sub-schema> must define a supported "type", but that "type" cannot be "array". This is a restriction on the language structure generated.

JSON Schema keyword	PL/I data description
"type": "object", "additionalProperties": false, "properties": { ["<element name>": {<JSON Sub-schema>} [,]]* } "required": [["<element name>" [,]]*]	The only form of JSON object currently supported is a fixed set of named elements. This will generate a structure (or sub-structure) using the element names. "additionalProperties" is assumed to be false, and no other value is supported. Any element in the "properties" object is considered"optional" if it is not in the "required" array or if no "required" array exists. An "optional" element is given a variable ordinality of zero to X; where X is either 1 or the maximum number of items in the array, where that item is defined as an array. See "Variable arrays of elements with MAPPING-MODE=JS2LS" on page 85.
"type": "object", "maxProperties": m, "minProperties": n, "patternProperties": {}, "dependencies":	None of these keywords are supported with JSON objects.
"type":"string" "maxLength":*m* "pattern": "regular expression>", "minLength": *l*	char[*z*] where the value of *z* is based on *m*, but dependent on the settings of the **CHAR-VARYING** parameter. *m* is based on the "maxLength" keyword and treated as fixed length string. "pattern" and "minLength" restrictions are passed through to the language structure only as a comment.
"type":"string" "maxLength":*m*	When CCSID=1200: WIDECHAR(*z*) where the value of *z* is based on *m*, but dependent on the settings of the **CHAR-VARYING** parameter. *m* is based on the "maxLength" keyword and treated as fixed length string.

"*name*":{
  "type":"string",
  "format":"date-time"
}
	FIXED DECIMAL (15,0)
	
	All supported when DATETIME=PACKED15

JSON Schema keyword	PL/I data description

"*name*":{
"type":"string",
"format":"uri"
}
	CHAR(*m*)
	
	where m is based on the "maxLength" keyword and treated as fixed length string.
	
	When **CCSID=1200**:
	
	WIDECHAR(*m*)
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string.

"*name*":{
"type":"string",
"format":"base64Binary"
}
	CHAR(*m*)
	
	where m is based on the "maxLength" keyword.

"*name*":{
"type":"string",
"format":"hexBinary"
}
	CHAR(*m*)
	
	where m is based on the "maxLength" keyword.

"*name*":{
"type":"string",
"format":"<predefined>"
}
	CHAR(*m*) where *m* is based on the "maxLength" keyword and treated as a fixed length string, and <predefined> is one of: *email*, *hostname*, *ipv4* or *ipv6*. A relevant "pattern" is passed to the comment.
	
	When **CCSID=1200**:
	
	WIDECHAR(*m*)
	
	where *m* is based on the "maxLength" keyword and treated as fixed length string, and where <predefined> is one of: *email*, *hostname*, *ipv4*, or *ipv6*. A relevant "pattern" is used and passed to the comment.

"type":"boolean"	**Enterprise PL/I** BIT(7) BIT(1) **Other PL/I** BIT(7) BIT(1) where BIT(7) is provided for alignment and BIT(1) contains the Boolean mapped value.
"type": "integer", "maxExclusive": true, "minExclusive": true, "multipleOf": n	"maxExclusive" and "minExclusive" restrictions are passed to the language structure only as a comment. "multipleOf" is ignored.

JSON Schema keyword	PL/I data description

`"type":"integer",`
`minimum:-128,`
`maximum:127`
	Enterprise PL/I
	`SIGNED FIXED BINARY (7)`
	Other PL/I
	`FIXED BINARY (7)`

`"type":"integer",`
`minimum:0,`
`maximum:255`
	Enterprise PL/I
	`UNSIGNED FIXED BINARY (8)`
	Other PL/I
	`FIXED BINARY (8)`

`"type":"integer",`
`minimum:-32768,`
`maximum:32767`
	Enterprise PL/I
	`SIGNED FIXED BINARY (15)`
	Other PL/I
	`FIXED BINARY (15)`

`"type":"integer",`
`minimum:0,`
`maximum:65535`
	Enterprise PL/I
	`UNSIGNED FIXED BINARY (16)`
	Other PL/I
	`FIXED BINARY (16)`

`"type":"integer",`
`minimum:-2147483648,`
`maximum:2147483647`
	Enterprise PL/I
	`SIGNED FIXED BINARY (31)`
	Other PL/I
	`FIXED BINARY (31)`

`"type":"integer",`
`minimum:0,`
`maximum:4294967295`
	Enterprise PL/I
	`CHAR(`*y*`)`
	where *y* is a fixed length that is less than 16 MB.
	Other PL/I
	`BIT(64)`

`"type":"integer",`
`minimum:-9223372036854775808,`
`maximum:9223372036854775807`
	Enterprise PL/I
	`CHAR(`*y*`)`
	where *y* is a fixed length that is less than 16 MB.
	Other PL/I
	`BIT(64)`

`"type":"integer",`
`minimum:0,`
`maximum:18446744073709551615`
	Enterprise PL/I
	`CHAR(`*y*`)`
	where *y* is a fixed length that is less than 16 MB.
	Other PL/I
	`BIT(64)`

`"type":"number"` `"description":"decimal"`	`FIXED DECIMAL(`*n*`,`*m*`)`

JSON Schema keyword	PL/I data description

`"type":"number"`
`"description":"float"`
	Enterprise PL/I
	`DECIMAL FLOAT(6) HEXADEC`
	Other PL/I
	`DECIMAL FLOAT(6)`
	Note: The IBM Hexadecimal Floating Point (HFP) data representation is not the same as the IEEE-754-1985 representation that is used for JSON. Some values might not convert exactly from one representation to the other. Some large or small values might not be valid for float data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of DECIMAL FLOAT data types with fixed precision alternatives.

`"type":"number"`
`"description":"double"`
	Enterprise PL/I
	`DECIMAL FLOAT(16) HEXADEC`
	Other PL/I
	`DECIMAL FLOAT(16)`
	Note: The IBM Hexadecimal Floating Point (HFP) data representation is not the same as the IEEE-754-1985 representation that is used for JSON. Some values might not convert exactly from one representation to the other. Some large or small values might not be valid for float data types. Some values might lose precision when converted to or from HFP representation. If precise conversions are important, consider replacing use of DECIMAL FLOAT data types with fixed precision alternatives.

Note: CICS TG cannot transform integer values greater than the maximum value for a signed long (2^63 - 1) unless they are enclosed within quotes.

Note: Minimum and maximum values specified in the schema for numeric types are used only to map to a PL/I datatype. Data is not validated against these values at run time.

Related reference:

"PL/I to JSON Schema mapping" on page 76
The JSON web services assistant with parameters `MAPPING-MODE=LS2JS` and `LANG=PLI-ENTERPRISE` or `LANG=PLI-OTHER` will map between PL/I data structures and JSON Schema definitions. Because the Enterprise PL/I compiler and older PL/I compilers differ, two language options are supported: `PLI-ENTERPRISE`, and `PLI-OTHER`.

Variable arrays of elements with MAPPING-MODE=JS2LS

JSON can contain arrays of varying numbers of elements.

An array with a varying number of elements is represented in the JSON Schema by using the `minItems` and `maxItems` keywords in the schema with `"type"` value of `"array"`:

- The `minItems` keyword specifies the minimum number of times that the element can occur. It can have a value of 0 or any positive integer. It defaults to the value 0.
- The `maxItems` keyword specifies the maximum number of times that the element can occur. It can have a value of any positive integer greater than or equal to the value of the `minItems` keyword.
- If the `maxItems` keyword is missing it means the array is unbounded.

An optional field can be denoted by a variable array of `"maxItems":1`. For example, an optional 8 byte string called `"component"`:

```
"properties":{
   "component": {
      "type":"array",
      "maxItems":1,
      "items": {
         "type": "string",
         "maxLength": 8
      }
   }
},
"required": ["component"]
```

The same effect can be produced by not including the field name in the `"required"` keyword value:

```
"properties":{
   "component": {
      "type": "string",
      "maxLength": 8
   }
}
```

In general, JSON Schemas that contain varying numbers of elements do not map efficiently into a single high-level language data structure. To handle these cases, CICS TG uses a series of connected data structures that are passed to the application program in a series of containers. These structures are used as input and output from the application:

- When CICS TG transforms JSON data to application data, it populates these structures with the application data.
- When CICS TG transforms the application data to JSON data, it reads the application data in the structures that have been populated by the application.

The following examples illustrate the format of these data structures. These examples use an array of simple 8 byte fields. However, the model supports arrays of complex data types and arrays of data types that contain other arrays.

Example 1. Fixed number of elements

This example illustrates an element that occurs exactly three times:

```
"properties":{
   "component": {
      "type": "array",
      "maxItems": 3,
      "minItems": 3,
      "items": {
         "type": "string",
```

```
         "maxLength": 8
      }
   }
},
"required": ["component"]
```

In this example, the number of times that the element occurs is known in advance, so it can be represented as a fixed-length array in a simple COBOL declaration, or the equivalent in other languages.

```
05 component PIC X(8) OCCURS 3 TIMES.
```

Example 2. Variable number of elements

A JSON array containing a variable number of elements can be mapped using container-based or inline mapping. This example illustrates a mandatory element that can occur from one to five times:

```
"properties":{
   "component": {
      "type": "array",
      "maxItems": 5,
      "minItems": 1,
      "items": {
         "type": "string",
         "maxLength": 8
      }
   }
},
"required": ["component"]
```

If container-based mapping is used, the main generated data structure contains a declaration of two fields. At runtime the JSON data is transformed to binary data, the first field `component-num` contains the number of times that the element appears in the JSON data, and the second field, `component-cont`, contains the name of a container:

```
05 component-num PIC S9(9) COMP-5.
05 component-cont PIC X(16).
```

A second data structure contains the declaration of the element itself:

```
01 <LS-REQUEST>01-component.
02 component PIC X(8).
```

To process the data structure the CICS application must examine the value of `component-num`, which, in this example, will contain a value in the range 1 to 5, to find how many times the element occurs. The element contents are in the container named in `component-cont`; the container holds an array of elements, where each element is mapped by the `<LS-REQUEST>01-component` data structure.

If `minItems="0"`, or is missing, and `maxItems="1"`, the element is optional. In this case, to process the data structure the CICS application must again examine the value of `component-num`:

If it is zero, the JSON data has no component element and the contents of `component-cont` is undefined.

If it is one, the component element is in the container named in `component-cont`.

Note: If the JSON data consists of a single recurring element, the JSON web services assistant with `MAPPING-MODE=JS2LS` generates two language structures. The

main language structure contains the number of elements in the array and the name of a container which holds the array of elements. The second language structure maps a single instance of the recurring element.

You can use the **INLINE-MAXOCCURS-LIMIT** parameter in the JSON web services assistant. The **INLINE-MAXOCCURS-LIMIT** parameter specifies the way that varying numbers of elements are handled. The mapping options for varying numbers of elements are container-based mapping or inline mapping. The value of this parameter can be a positive integer in the range 0 - 32767:

- The default value of **INLINE-MAXOCCURS-LIMIT** is 1, which ensures that optional elements are mapped inline.
- A value of 0 for the **INLINE-MAXOCCURS-LIMIT** parameter prevents inline mapping.
- If maxItems is less than or equal to the value of **INLINE-MAXOCCURS-LIMIT**, inline mapping is used.
- If maxItems is greater than the value of **INLINE-MAXOCCURS-LIMIT**, container-based mapping is used.

Mapping varying numbers of elements inline results in the generation of both an array and a counter. The component-num field indicates how many instances of the element are present, and these are pointed to by the array. When **INLINE-MAXOCCURS-LIMIT** is less than or equal to 5, the generated data structure is like this:

```
05 component-num PIC S9(9) COMP-5 SYNC.
05 component OCCURS 5 PIC X(8).
```

The first field, component-num, is identical to the output for container-based mapping. The second field contains an array of length 5 which is large enough to contain the maximum number of elements that can be generated.

Container-based mapping stores the number of occurrences of the element and the name of the container where the data is placed, in the main language structure. Inline mapping stores all the data in the current container. Storing the data in the current container will generally improve performance and make inline mapping preferable.

Example 3. Nested variable arrays

Complex JSON Schemas can contain variable recurring elements, which in turn contain variable recurring elements. In this case, the structure described extends beyond the two levels described in the examples.

This example illustrates an optional element called "component2" that is nested in a mandatory element called "component1", where the mandatory element can occur from one to five times:

```
"properties":{
   "component1": {
      "type": "array",
      "maxItems": 5,
      "minItems": 1,
      "items": {
         "type": "object",
         "properties":{
            "component2":{
               "type": "string",
               "maxLength": 8
            }
         },
```

```
      "required": ["component2"]
      }
   },
"required": ["component1"]
```

The top-level data structure is exactly the same as in the previous examples:

```
05 component1-num PIC S9(9) COMP-5
05 component1-cont PIC X(16)
```

However, the second data structure contains these elements:

```
01 <LS-REQUEST>01-component1
  02 component2-num PIC S9(9) COMP-5
  02 component2-cont PIC X(16)
```

A third-level structure contains these elements:

```
01 <LS-REQUEST>01-component2
  02 component2 PIC X(8)
```

The number of occurrences of the outermost element "component1" is in component1-num.

The container named in component1-cont contains an array with that number of instances of the second data structure <LS-REQUEST>01-component1.

Each instance of component2-cont names a different container, each of which contains the data structure mapped by the third-level structure <LS-REQUEST>01-component2.

To illustrate this structure, consider the fragment of JSON data that matches the example:

```
{"component1":
   [
      {
         "component2": "string1"
      },
      {
         "component2": "string2"
      },
   ]
}
```

"component1" occurs three times. The first two contain an instance of "component2"; the third instance does not.

In the top-level data structure, component1-num contains a value of 3. The container named in component1-cont has three instances of <LS-REQUEST>01-component1:

1. In the first, component2-num has a value of 1, and the container named in component2-cont holds *string1*.
2. In the second, component2-num has a value of 1, and the container named in component2-cont holds *string2*.
3. In the third, component2-num has a value of 0, and the contents of component2-cont are undefined.

In this instance, the complete data structure is represented by four containers in all:

- The root data structure in container <LS-REQUEST>01-DATA.
- The container named in component1-cont.
- Two containers named in the first two instances of component2-cont.

Optional structures and the `required` keyword

Data structures are defined by the JSON Schema "type" of "object". The schemas relate field names to individual types using the object provided by the "properties" keyword. The requirement for these fields to be part of the JSON data described by the JSON Schema is controlled by the array provided by the "required" keyword. This array lists all the field names that must be present in the JSON data. Optional fields are therefore represented by their absence from this array, or as the array is not allowed to be empty, the absence of the "required" keyword. In this case, all fields are optional.

Optional fields are treated as a variable array of 0 or 1 elements. This adds an additional field with the suffix "-num" appended to the element name. If the total length is more than 28 characters, the element name is truncated. At run time this will be non-zero to indicate the value was present in the JSON data and zero if it was not.

This example shows two fields, one required called "required-structure" and the other one optional called "optional-structure" :

```
{
   "type": "object",
   "properties":  {
      "required-structure": {
         "type":"string",
         "maxLength": 8
      },
      "optional-structure": {
         "type":"string",
         "maxLength": 8
      }
   },
   "required": [
      "required-structure"
   ]
}
```

The generated COBOL structure shows both fields, but the second is preceded by "optional-structure-num" which is an integer count of the elements, with 0 representing none and 1 that it is present. The value is set to indicate whether the "optional-structure" contains valid data or not.

```
03 OutputData.
          06 required-structure          PIC X(8).
          06 optional-structure-num      PIC S9(9) COMP-5 SYNC.
          06 optional-structure          PIC X(8).
```

JSON web service restrictions

Use this reference material to understand capabilities that are not supported by JSON web services.

The following capabilities are not supported:

- Use of namespaces in JSON data (Badgerfish or Mapped conventions) is not supported.
- JSON payloads sent to CICS TG must be encoded in UTF-8. No other encoding is supported. Similarly, JSON sent by CICS TG is always encoded in UTF-8.
- If a JSON payload is missing some of its required content when CICS TG transforms it, the equivalent fields within the data structures are not initialized when passed to the application program.

- CICS TG cannot transform integer values greater than the maximum value for a signed long (2^63 - 1) unless they are enclosed within quotes.
- Use of simple data types or arrays is not supported at the root of a JSON Schema. The JSON Schema is required to describe a JSON Object, though the JSON Object can be composed of simple data types and arrays.

Note: Supported characters for JSON property names are: A-Z a-z _ : for the first character and A-Z a-z 0-9 _ : . - for all subsequent characters.

Error responses from JSON web services

If an error occurs during the execution of a JSON web services request, the HTTP status code is set to a value that indicates the type of error.

Details of the error are returned in a JSON object with the following format:

```
{
   "Fault":
   {
      "detail":
      {
         <additional detail fields>
      },
      "faultstring": "<Error text>"
   }
}
```

The `detail` object can contain the following fields:

- `Description`: A description of the error suitable for display to users.
- `CICSServer`: The CICS server that the web service request was sent to.
- `AbendCode`: The abend code if an abend occurs in the CICS program that implements the web service.
- `ExceptionMessage`: The exception message if an internal exception occurs within CICS TG.

A CICS program that uses the channel interface to implement a RESTful web service, can indicate its own error conditions by setting the HTTP status code in the `DFHHTTPSTATUS` container along with a custom error response. For more information, see "Creating a RESTful JSON web service" on page 34.

Chapter 8. Statistics APIs

The statistics APIs enable user applications to obtain runtime statistics on the Gateway daemon. To use the statistics APIs, the Gateway daemon must be configured with a statistics API protocol handler.

TCP/IP is used to connect the application to the Gateway daemon. The application can be local or remote to the Gateway daemon.

Statistical data overview

The statistics APIs provide a single-threaded or multithreaded user application access to statistical data from one or more running Gateway daemons. TCP/IP connects the application to the Gateway daemon. The application can be local or remote to the Gateway daemon.

API functions

The APIs provide functions to:
- Connect to specific Gateway daemons.
- Disconnect from specific Gateway daemons.
- 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

API and protocol version control

The API version represents the programming interface available from the ctgstats runtime library. The protocol version represents the set of responses that may be returned by a connected Gateway daemon in response to a statistics API function call. Comparison of compile time versus runtime values can be made to establish compatibility.

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`

© Copyright IBM Corp. 1998, 2016

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 API version that it was compiled with by using the compile-time string `CTG_STAT_API_VERSION`, defined in `ctgstats.h`.
- Find which API version is used at run time by a CICS Transaction Gateway daemon, or Java statistics API by using the "getStatsAPIVersion" on page 106 function.
- Find the protocol version that it was compiled with by using the compile-time string `CTG_STAT_PROTOCOL_VER`, defined in `ctgstdat.h`.
- Find which protocol version is used at run time by a CICS Transaction Gateway daemon, by using the "openGatewayConnection" on page 101 or "openRemoteGatewayConnection" on page 101 function.

API version

The major version number, first digit, of the statistics API version must match between the application at compile time and ctgstats runtime library.

For example; if `CTG_STAT_API_VERSION` is 1_0_0_0 and the runtime function *getStatsAPIVersion* returns 1_1_0_0 then the major version (1_x_x_x) matches. Therefore the application is guaranteed to be runtime compatible with at least those functions available for version 1_1_0_0.

If the major version numbers differ, runtime compatibility is not guaranteed and API calls might fail.

Assuming that the major version number matches, then the minor version number (second digit) of the statistics API version at application compile time must be the lower than or equal to the ctgstats runtime library.

For example; if `CTG_STAT_API_VERSION` is 1_0_0_0 and the runtime function *getStatsAPIVersion* returns 1_1_0_0 then the major version (1_x_x_x) matches, and the minor version (x_0_x_x) used by the application is lower than the runtime library. Therefore, the application is guaranteed to be runtime compatible because it can only use those functions that are available at runtime version 1_0_0_0.

If the minor version number. second digit, of the statistics API version at application compile time is greater than the `ctgstats` runtime library, then some functions available at compile time will not be available at run time. The 3rd and 4th digits are reserved for IBM service and maintenance usage.

Protocol version

The protocol version adheres to similar rules between compile time and run time as the API Version. However, the protocol version represents the interface between the compiled statistics application and the Gateway daemon connected at run time.

The major version number, first digit, of the protocol version must match between the application at compile time and the connected Gateway daemon.

Assuming that the major version number matches, then the minor version number, second digit, of the statistics API application at application compile time, must be

the greater than or equal to the minor version number returned by the connected CICS Transaction Gateway daemon upon connection. If the minor version number is lower than that of the connected Gateway daemon, then the statistics API application might be unable to interpret all responses from function calls.

Statistics C API

The statistics C API enables a C client application to request statistics from a Gateway daemon that has configured and enabled a statistics protocol handler. The CICS TG Statistics C API supports 32-bit applications.

Calling the C API

This section explains how applications call API functions.

Applications call C API functions defined in "C language header files," 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 C API does not provide logging messages. Runtime operation of the C API functions can be monitored using trace facilities. Statistics C API tracing can be enabled programatically with data written to `stderr`, or a specified file. C API errors are reported to the calling application using an integer result code.

Statistics C API components

The statistics C 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 C API function calls and data types required to use the C 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 C API runtime DLL is provided for each of the supported CICS Transaction Gateway hardware platforms. It is supplied as a platform-specific DLL or shared library. It must be available during the run time of the statistical user application.

File names and locations

The runtime DLL and header files are installed by the installer. The details of the files are provided in the following table.

Table 8. 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
IBM AIX	DLL	libctgstats.a	lib
HP-UX on Itanium	DLL	libctgstats.so	lib
Linux on Intel	DLL	libctgstats.so	lib
Linux on POWER®	DLL	libctgstats.so	lib
Linux on IBM z Systems	DLL	libctgstats.so	lib
Solaris on SPARC	DLL	libctgstats.so	lib
UNIX and Linux	Sample Makefile	samp.mak	samples/c/stats
Windows	DLL	ctgstats.dll	lib
Windows	Export symbols	ctgstats.lib	lib
Windows	Sample Makefile	ctgstat1mak.cmd	samples/c/stats

For information about supported compilers, see the information about development environments in the *CICS Transaction Gateway: UNIX and Linux Administration* or the *CICS Transaction Gateway: Windows Administration*.

Windows platform
: At compile time, applications that use the statistics C API need access to the C 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.

Sample code

A sample file `ctgstat1.c` is supplied. This provides a simple example for using the statistics C API. For further details, see “C and Java statistics API samples” on page 277.

Statistics C API program structure

Outline of a basic statistics C API program.

A basic statistics C 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
```

C API data types

Data types defined and used by the statistics API.

This information describes the main data types used by the statistics C 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 C API calls that retrieve statistical data must include the Gateway token as a parameter.

The statistics C 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 C API programs, or threads, at the same time.

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

A statistical C API program can hold multiple Gateway tokens, but can only use them on the thread that called the “openGatewayConnection” on page 101 or “openRemoteGatewayConnection” on page 101 to retrieve the token.

A Gateway token type (`CTG_GatewayToken_t`) is defined in the “C language header files” on page 95.

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 C 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 C API function produces a data result set, the function "getIdQuery" on page 104 can be used to obtain a query string suitable for input to another C 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 C API function call.

If a statistics C 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 can 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 104 or "getNextStat" on page 105 to manipulate the result set data.

An "ID data" on page 100 type is populated by the "getFirstId" on page 104 and "getNextId" on page 105 functions. A "Statistical data" on page 100 type is populated by the "getFirstStat" on page 105 and "getNextStat" on page 105 functions. The data types are used to access the data in the result sets, as described in "Correlating results and error checking" on page 108.

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

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

- The associated Gateway daemon connection is closed using the "closeGatewayConnection" on page 102 function.

or

- The function "closeAllGatewayConnections" on page 102 is called.

The result set token returned by the "copyResultSet" on page 105 function is *not* 'owned' by the C API. The token can only be freed using the "freeResultSet" on page 106 function.

Result set tokens 'owned' by the C API cannot be 'freed' using the “freeResultSet” on page 106 function. The tokens must be freed using the “closeGatewayConnection” on page 102 or “closeAllGatewayConnections” on page 102 functions.

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

Working with multiple result sets:

Working with multiple result sets requires special attention.

Calling a statistics C API function produces a result set token. This token identifies a result set owned by the statistics C 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 C 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 C API-owned result sets can only be used by the thread which created them. To create a result set token usable by any thread, call the “copyResultSet” on page 105 function.

For example, an application using the same gateway token to make two separate C 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 C API function call.

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

Use the “copyResultSet” on page 105 function to make a copy of a result set before it is overwritten by another C API call. When the application finishes using the copied result set, free the storage using the “freeResultSet” on page 106 function.

Example

In the following example code, two statistics C 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 C 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 C API call is still required later in the application, take a copy of the result set by calling the “copyResultSet” on page 105 function.

ID data

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

The data type CTG_IdData_t is defined in the "C language header files" on page 95. The data provides a name for individual results within statistical groups or statistics.

Individual results can be accessed using the "getFirstId" on page 104 and "getNextId" on page 105 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 108.

Statistical data

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

The data type CTG_StatData_t is defined in the "C language header files" on page 95. The statistical data represents individual statistics, or name-value pairs.

Individual results can be accessed using the "getFirstStat" on page 105 and "getNextStat" on page 105 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 108.

Statistics C API trace levels

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

Trace levels

The CICS Transaction Gateway statistics C 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 9. Statistics C 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 107 to choose a different trace destination.

C API functions

The statistics C API functions.

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 105 function.

For details of the return codes provided by the C API functions, see `ctgstats.h` in the “C language header files” on page 95, or see the Statistics APIs.

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 C 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 C 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 C API calls.

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

The user application must check that the version of the statistics C 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 95 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 C API protocol version.

Detail

This function is called with:

- A character pointer for the host name. This is a null terminated string containing the IP address or host name 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 C 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 C API calls.

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

The user application must check that the version of the statistics C 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 95 section. The major version number is the first digit in the compile-time string.

closeGatewayConnection:

This function closes a connection to a local 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 C API, including any open Gateway daemon connections.

Detail

An application can use this function as part of a typical 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 C API does not own or maintain a record of copied result sets.

ID functions

This information describes the ID functions provided in the statistics C 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 104 and "getNextId" on page 105, or used to generate a query string with "getIdQuery" on page 104.

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 can be parsed with functions "getFirstId" on page 104 and "getNextId" on page 105, or used to generate a query string with "getIdQuery" on page 104.

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 can be parsed with functions "getFirstId" on page 104 and "getNextId" on page 105, or used to generate a query string with "getIdQuery" on page 104.

Retrieving statistical data functions

This information describes the data retrieval functions provided in the statistics C 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 can be parsed with functions "getFirstStat" on page 105 and "getNextStat" on page 105, or used to generate a query string with "getIdQuery" on page 104.

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” and “getNextId” on page 105, or used to generate a query string with “getIdQuery.”

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 105 and “getNextStat” on page 105, or used to generate a query string with “getIdQuery.”

Result set functions

This information describes the result set functions provided in the statistics C 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 C API. The C API owns the storage for the character array, and the C API program does 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 100.

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 100.

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 100.

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 100.

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 C API calls on a result set. This is useful because some C 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 104 work with C 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.

Use this function only for result sets created using the “copyResultSet” on page 105 function. If the result set is owned by the statistics C API, an attempt to free the result set returns an error.

Utility functions

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

getStatsAPIVersion:

This function provides version information about the statistics C 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 93. The C API owns the storage for the character array, and the C API program does not free this storage.

getAPITraceLevel:

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

The levels are defined in the “C language header files” on page 95. 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 C API trace levels" on page 100.

setAPITraceLevel:

This function sets the trace level of the statistics C API.

Detail

This function takes an `int` value. The function sets the trace level of the C 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 95. 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 C API trace levels" on page 100.

setAPITraceFile:

This function sets the destination for statistics C 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 typically used for debug purposes.

Related reference:

"Statistics C API trace levels" on page 100
The CICS Transaction Gateway statistics C API provides several levels of diagnostic trace information.

dumpState:

This function outputs internal information about the C API.

Detail

This function writes internal information about the C 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 C API function call can be correlated back to the original query string data.

ID or statistical results within a result set from an C 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 104 or "getNextId" on page 105, 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 105 or "getNextStat" on page 105, 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.

Statistics Java API

The statistics Java API enables a Java-based client application to request statistics.

Calling the Java API

Applications can collect statistics from a Gateway daemon using the Java classes in the com.ibm.ctg.client.stats package. The classes are supplied in a the ctgstats.jar and can be used with Gateway daemons from V7.1 onwards. A sample file Ctgstat1.java is supplied that provides a simple example for using the Java statistics API.

Packaging restrictions with ctgstats.jar

If an application needs to use classes from both the com.ibm.ctg.client.stats package provided by ctgstats.jar and another API package supplied in ctgclient.jar, both jar files must be on the class path and must be from the same product version and release. The implication is that such an application can only connect to a Gateway daemon at the same version or higher for non-statistical requests.

The `ctgstats.jar` file can be used in isolation for standalone monitoring applications. `ctgstats.jar` is compatible with `ctgclient.jar` provided both jar files are from the same version of CICS TG. Mixing `ctgstats.jar` and `ctgclient.jar` that are from different versions of CICS TG is not supported.

Sample code

A sample file `Ctgstat1.java` is supplied that provides a simple example for using the statistics API.

Java API classes

The Java API classes are responsible for connecting and making statistical requests to a statistics port provided by the Gateway daemon. The constructors allow the destination to be supplied by the application.

The statistic resource groups are available through the *getResourceGroupIds* method. An *IdResultSet* object is returned that contains a collection of *IdData* objects that hold the names of the resource groups. You can iterate over the *IdResultSet* to search the resource groups available.

If the names of the available statistics are required use the *getStatIds* method. This method returns an *IdResultSet*, functioning the same as *getResourceGroupIds*.

You can retrieve actual statistic values using the *getStats* method. This method returns a *StatResultSet* object that contains a collection of *StatData* objects. These *StatData* objects contain both the statistic names, and their current values. You can iterate over the *StatResultSet* to search the statistics available from the request.

If a result set returned has the return code set you can map this to the reason using the *getReturnString* method of the *ResponseData* class.

Tracing

You can enable statistics API tracing programmatically using the Java tracing options, see "Tracing in Java client programs" on page 132. Java API errors are reported to the calling application.

Related information:

Package com.ibm.ctg.client.stats

Chapter 9. Code page information

When using the application programming interfaces of the CICS Transaction Gateway to start 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. For COMMAREA data, you can do this conversion by using 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 use the 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 see the Javadoc for `com.ibm.ctg.client.CicsCpRequest`.

© Copyright IBM Corp. 1998, 2016

Chapter 10. 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 following list of classes are the basic classes provided with the CICS Transaction Gateway. For a full description of all the classes and methods discussed in this section, see the Javadoc supplied with the CICS Transaction Gateway.

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 send requests 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.

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. 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.

When the JavaGateway is open, the Java client application can flow requests to the CICS server using the flow method of the JavaGateway. When there are no more requests for the CICS Transaction Gateway, the Java client application closes the JavaGateway object.

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.

© Copyright IBM Corp. 1998, 2016

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 on Itanium 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

Java applications that connect to the Gateway daemon using TCP or SSL can specify a local port number to use for the connection. Set the **`JavaGateway.PROP_CLIENT_PORT`** property in a `java.util.Properties` object to the numeric value of the port to use, then pass the Properties object to the JavaGateway constructor. If the local port is already in use the connection will fail, so a different property value must be used for each concurrent TCP or SSL connection.

SSL cipher suites in Java Client applications

Cipher suites define the key exchange, data encryption, and hash algorithms used for an SSL session between a client and server.

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. CICS Transaction Gateway uses cipher suites provided by the Java runtime environment for the SSL protocol. The cipher suites available to be used are dependant on the Java version. See the documentation supplied with your Java runtime environment for valid cipher suites.

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);
```

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 ECI 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 CICS Transaction Gateway. The following table shows Java objects corresponding to the ECI terms described in “I/O parameters on ECI calls” on page 9.

Table 10. ECI terms and corresponding Java objects

ECI term	Java object.field or object.method()
Abend code	ECIRequest.Abend_Code
Channel	ECIRequest.setChannel(channel) See “Introduction to channels and containers” on page 13.
COMMAREA	ECIRequest.Commarea See “ECI performance considerations when using COMMAREAs” on page 15.
ECI timeout	ECIRequest.setECITimeout(short) See “Timeout of the ECI request” on page 14.
LUW control	ECIRequest.Extend_Mode See “Program link calls” on page 10.
LUW identifier	ECIRequest.Luw_Token See “Managing logical units of work” on page 11.
Message qualifier	ECIRequest.getMessageQualifier() and ECIRequest.setMessageQualifier() See “Retrieving replies from asynchronous ECI requests” on page 12.
Password or password phrase	ECIRequest.Password See “Security in the ECI” on page 15.
Program name	ECIRequest.Program
Server name	ECIRequest.Server

Table 10. ECI terms and corresponding Java objects (continued)

ECI term	Java object.field or object.method()
TPNName	ECIRequest.Call_Type = ECI_SYNC_TPN or ECI_ASYNC_TPN and ECIRequest.Transid See "ECI and CICS transaction IDs" on page 13.
TranName	ECIRequest.Call_Type = ECI_SYNC or ECI_ASYNC and ECIRequest.Transid See "ECI and CICS transaction IDs" on page 13.
User ID	ECIRequest.Userid See "Security in the ECI" on page 15.

Linking to a CICS server program

A link to a CICS program is made using an ECIRequest constructor 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 program link calls by setting the **Call_Type** field to `ECI_SYNC` or `ECI_SYNC_TPN`. You can create ECI requests for asynchronous program link calls by setting the **Call_Type** field to `ECI_ASYNC` or `ECI_ASYNC_TPN`. The `ECI_SYNC` and `ECI_ASYNC` call types cause the **Transid** field to be used as **TranName**, and the `ECI_SYNC_TPN` and `ECI_ASYNC_TPN` call types cause the **Transid** field to be used as **TPNName**.

Creating Java channels and containers for ECI calls

You can use channels and containers when you connect to CICS using the IPIC protocol. You must construct a 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);
   jgateway.flow(eciReq);
   ```
4. When the request is complete, you can retrieve the contents of the containers in the channel by interpreting the type. For example:

```
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();
   }
}
```

Managing an LUW

Set the extend mode to ECI_EXTENDED if the ECI call is part of an extended LUW. If the call is the last, or only call for the LUW, the extend mode must be ECI_NO_EXTEND, ECI_COMMIT or ECI_BACKOUT.

Retrieving replies from asynchronous requests

Replies to asynchronous requests can be retrieved by using callbacks or reply solicitation calls.

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

You can retrieve asynchronous replies using message qualifiers and reply solicitation calls.

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.

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 must 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

IPIC connections do not support asynchronous requests using message qualifiers from Java clients. Java clients that perform asynchronous requests using IPIC connections must use callbacks.

Making EPI 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. Table 11 shows Java objects corresponding to the EPI terms described in the following table.

Table 11. 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
User ID	Terminal:Userid	EPIRequest.userid

EPI support classes

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

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

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 126.

In the examples in this section, 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 5, "External Presentation Interface (EPI)," on page 19.

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 and 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 121 and "Synchronization and sessions" on page 123 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 extended terminal:

There are two ways to construct an extended terminal: using the default constructor and 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 23)
- `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”).

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 by using one of the following connect methods.

Installing a terminal on CICS:

There are three connect methods that you can use to install a terminal on CICS.

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 123.

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

Synchronous and asynchronous events, such as Read Timeout events, and terminal connections to CICS 3270 screens.

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. Call the **endReason** and **endReasonString** methods for more information.

Terminal.discon
: Indicates that the terminal has been deleted. Call 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 calls 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," 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 the following diagram:

Figure 4. 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 start 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 123, 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

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.

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 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 -cp <install path>/classes/ctgclient.jar 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 features of the classes generated by using the BMS Conversion Utility, and using the generated Map class.

- 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.

Using the EPIRequest class

It is recommended that you use the EPI support classes or the JEE 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 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 ESI calls from a Java client program

Use the ESIRequest base class for password management.

The following table shows Java objects corresponding to the ESI terms listed in "I/O parameters on ESI calls" on page 27.

Table 12. 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

Issues to consider when compiling and running a Java client application include performance, the Java class path and whether or not you are running a Web browser on the same machine as CICS Transaction Gateway.

Setting stack and heap sizes

There are several memory allocation 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. See your Java documentation for more information.

For information about setting the Java heap size for the Gateway daemon, see *Setting Gateway daemon JVM options*.

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.

Integration testing Java applications using a Gateway Intercept plug-in

You can use integration test tools to intercept ECI, EPI, and ESI calls from Java Standard Edition and Java Enterprise Edition applications. This feature allows such applications to be tested without requiring a running CICS server using tools such as IBM Rational Integration Tester.

Overview

You can use a plug-in to test your application by writing a Java program which implements the GatewayIntercept interface, and enabling this plug-in for your application

When enabled, a Gateway intercept plug-in is notified before a connection to a Gateway daemon is opened or closed, and before a request is sent to the Gateway daemon.

- Before a connection is opened, the plug-in can change the properties of the connection object, and can prevent the connection from opening.
- Before a request is sent, the plug-in can change the properties of the request object, and return the request to the application without being sent.
- Before a connection is closed, the plug-in can prevent the connection from closing.

This feature allows various scenarios to be simulated for application testing.

Related tasks:

"Integration testing JEE components using an intercept plug-in" on page 157
You can use integration test tools to intercept ECI and EPI calls from Java Enterprise Edition components. This feature allows such components to be tested

without requiring a running CICS Transaction Gateway or CICS server.

The sample plug-in

A sample plug-in, which is written in Java, is provided with this feature.

The plug-in is used with the `EciB1` Java sample included with CICS TG. When the plug-in is active, `EciB1` can be run successfully without requiring a CICS TG or a CICS server to be running. The `EciB1` Java sample is provided compiled into `ctgsamples.jar` along with all other samples.

To use the `EciB1` sample plug-in, issue the following commands:

- On Linux, UNIX and IBM z/OS enter the command on one line:

```
java -cp ctgclient.jar:ctgsamples.jar
  -Dgateway.intercept.plugin=com.ibm.ctg.samples.intercept.BasicPlugin
  com.ibm.ctg.samples.eci.EciB1 cicstg.host.name
```

- On Windows:

```
java -cp ctgclient.jar;ctgsamples.jar
  -Dgateway.intercept.plugin=com.ibm.ctg.samples.intercept.BasicPlugin
  com.ibm.ctg.samples.eci.EciB1 cicstg.host.name
```

Plug-in development

The API for implementing a Gateway intercept plug-in is provided in `ctgclient.jar`.

Plug-in classes must implement the com.ibm.ctg.client.GatewayIntercept interface.

To compile your plug-in class, `ctgclient.jar` must be available on the class path. For more information, see “Setting up the CLASSPATH” on page 130.

Related information:

“Compiling applications” on page 156
To enable Java applications to be compiled in a managed or nonmanaged environment, the relevant .jar details must be added to the class path.

“Running the sample programs” on page 252
To run the sample programs, ensure that ctgsamples.jar and ctgclient.jar are referenced in your class path. If running the sample in local mode, ctgserver.jar is also required.

Enabling a Gateway intercept plug-in in a Java SE application

You can enable a Gateway intercept plug-in in a Java SE application, programmatically or by using a Java system property.

Procedure

1. Ensure that the plug-in class is available on the class path of your application.
2. To enable the plug-in, choose one of the following methods:
   - Programmatically:

   ```
   Properties connectionProps = new Properties();
   connectionProps.put(GatewayIntercept.CTG_PROP_INTERCEPT_PLUGIN,
   "plugin.package.name.PluginClassName");
   gateway.setProtocolProperties(connectionProps);
   gateway.open();
   ```

   - By using a Java system property:

   ```
   java -Dgateway.intercept.plugin=plugin.package.name.PluginClassName
   application.package.name.ApplicationClassName
   ```

Related information:

Chapter 10, “Programming in Java,” on page 113
This information provides an introduction to writing Java client programs for the CICS Transaction Gateway.

Problem determination for Java client programs

Use tracing to help determine the cause of any problems when running Java clients.

Tracing in Java client programs

You can control tracing in Java client programs by issuing various calls and by setting properties. Ideally applications should implement an option that activates trace.

Calling the com.ibm.ctg.client.T trace class

Here is an example of how to call this class 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.

Setting the gateway.T trace system property

Here is an example of how to set this property:

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

This example specifies full debug trace for `testprog1`.

For more information on the use of system properties see your Java documentation.

Standard trace

This is 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.

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

System property: gateway.T.trace=on

Full debug trace

This is 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.

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

System property: gateway.T=on

Exception stack trace

This is the exception stack option for application tracing. It traces most Java exceptions, including exceptions which are expected during typical 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.

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

System property: gateway.T.stack=on

Additional options for configuring trace

You can also configure additional options for trace, including: output destination, data block size, dump offset, and whether or not to include timestamps. Use these options, in addition to one of the directives, to activate trace. For example, the following command activates standard trace, and also sets the maximum size of any data blocks to be dumped to 20 000 bytes:

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

Output destination

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

System property: gateway.T.setTFile=*filename*

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 nested within quotation marks, for example: "trace output file.log".

Example: **`java -Dgateway.T.trace=on -Dgateway.T.setTFile="trace output file.log"`**

Data block size

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

System property: gateway.T.setTruncationSize=*number*

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.

Example: **`java -Dgateway.T.trace=on -Dgateway.T.setTruncationSize=20000`**

Dump offset

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

System property: gateway.T.setDumpOffset=*number*

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.

Example: **`java -Dgateway.T.trace=on -Dgateway.T.setDumpOffset=100`**

Display timestamps

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

System property: gateway.T.timing=on

Specifies whether or not to display timestamps in the trace.

Example: **`java -Dgateway.T.trace=on -Dgateway.T.setTimingOn="true"`**

Security for Java client programs

CICS Transaction Gateway provides the Java classes for implementing security. Java provides the Security Manager.

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.

  See 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.

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 works 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 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 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.

If you are implementing the security exits you must implement both a client-side security class and server-side security class.

For applications using Java base classes, the security classes are specified using the setSecurity method prior to opening the JavaGateway object. When using a JEE application server, the security classes are specified using the clientSecurity and serverSecurity connection factory properties. For non-managed JCA applications, the security classes are specified using the setClientSecurity and setServerSecurity methods.

The client-side security class must be available on the class path of the application for Java base classes and non-managed JCA applications, or on the class path of the resource adapter when using a JEE application server. The server-side security class must be available on the class path of the Gateway daemon.

To use the `com.ibm.ctg.security.ClientSecurity` security classes, you must configure the requiresecurity configuration parameter available with the TCP and SSL protocol handlers.

Using a Java 2 Security Manager

Java 2 provides a Security Manager system that controls access to Java resources.

The Security Manager restricts access to Java resources using a security policy. Some 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 JEE server), your application needs Java permissions to run correctly. The only exception to this is if you are using the JEE APIs in a managed environment.

Figure 5 on page 136 shows the minimum permissions that your application needs to use Gateway Java APIs. It might need additional permissions to run 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 5. 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 statement gives permission for the CICS Transaction Gateway to access and create an `ibm/ctg` (on UNIX and Linux) or `ibm\ctg` (on Windows) subdirectory in the users' home directory:

```
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 the information about Network security management in the *CICS Transaction Gateway: UNIX and Linux Administration* or the *CICS Transaction Gateway: Windows Administration* for information on how JSSE is selected as the implementation.

For example, on Windows, you can specify the following permission to allow access to the directory `c:\trace` and all subdirectories:

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

On Unix and Linux systems, you can specify the following permission to allow access to the directory `/tmp/ibm` and all subdirectories:

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

Signing Applets and Web Start Applications

The security configuration for Java 7 running in a browser has changed.

The default security configuration for Java 7 running in a browser changed significantly in the January 2014 CPU (Oracle 7u51, IBM 7 SR6-FP1).

When you run in these Java environments:

- All Applets and Web Start applications must be signed with a certificate from a trusted authority. Self-signed certificates are not accepted.
- All JARS must have the `Permissions` attribute set in the JAR Manifest.

The `ctgclient.jar` file that is included in CICS TG has the `Permissions` attribute set and is signed with trusted CA certificates from Symantec:

- Symantec Root CA for all SSL and Code Signing products enrolled after October 10, 2010.
- Symantec Intermediate CA Certificates: Code Signing Certificate

Any other JARs running as part of Applets and Web Start applications also need to have the `Permissions` attribute set in the JAR Manifest and be signed with a certificate from a trusted authority.

Using the CICS TG OSGi bundle

You can use any of the CICS TG Java APIs from an OSGi environment, this allows for CICS TG Java client applications running in remote mode to benefit from the advantages of running in an OSGi framework.

The CICS TG OSGi Bundle is called `com.ibm.ctg.client-1.0.0.jar` and is found in the SDK in the `cicstgsdk\api\java\runtime` directory.

The following packages are exported from the bundle:

- `com.ibm.ctg.client`
- `com.ibm.ctg.client.exceptions`
- `com.ibm.ctg.epi`
- `com.ibm.ctg.monitoring`
- `com.ibm.ctg.security`
- `com.ibm.ctg.util`

Building an OSGi client application

A client application bundle will need to import the required packages as part of it's manifest file. The client application bundle will need to be available for compilation and runtime.

Refer to the documentation of your OSGi environment for details on importing the packages and making the client application bundle available.

Using a Request Monitoring Exit in OSGi

The request monitoring exit needs to be packaged with the main application inside the same bundle or, in a separate bundle with the package containing the exit exported. This exit is then imported into the main application bundle.

Chapter 11. Programming using the JEE Connector Architecture

How to program using the ECI and EPI resource adapters provided by the CICS Transaction Gateway.

Overview of the JCA programming interface

JCA connects enterprise information systems such as CICS, to the JEE platform. JCA supports the qualities of service provided by a JEE application server (security credential management, connection pooling and transaction management).

Qualities of service are provided through system level contracts between a resource adapter provided by CICS Transaction Gateway and the JEE application server. There is often no need for any extra program code to be provided. The programmer is therefore free to concentrate on writing business code and need not be concerned with quality of service. For information about the provided qualities of service and configuration guidance see the documentation for your JEE application server.

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

CICS Transaction Gateway Desktop Edition: Support is not provided for the JCA resource adapters.

The Common Client Interface (CCI)

The CCI is a high-level programming interface defined by the JEE Connector Architecture (JCA).

The CCI is available to JEE developers who want to use the External Call Interface (ECI) and the External Presentation Interface (EPI) to enable client applications to communicate with programs running on a CICS server. The CCI does not support the External Security Interface (ESI).

The CCI has two class types:

- Generic CCI classes used for requesting a connection to an EIS such as CICS, and for executing commands on that EIS, passing input and retrieving output. These classes are generic because they do not pass information specific to a particular EIS. Examples are Connection and ConnectionFactory.
- CICS-specific classes used for passing specific information between the Java Client application and CICS. Examples are ECIInteractionSpec and ECIConnectionSpec.

The programming interface model

Applications that use the CCI have a common structure for all enterprise information systems. The JCA defines connections and connection factories that

© Copyright IBM Corp. 1998, 2016

represent the connection to the EIS. These connection objects allow a JEE application server to manage the security, transaction context and connection pools for the resource adapter.

An application must start by accessing a connection factory from which a connection can be acquired. The properties of the connection can be overridden by a ConnectionSpec object. The ConnectionSpec class is specific to CICS and can be either an ECIConnectionSpec or an EPIConnectionSpec.

After a connection has been acquired, an interaction can be created from the connection to make a particular request. The interaction, like the connection, can have custom properties which are set by the InteractionSpec class (ECIInteractionSpec or EPIInteractionSpec) which is specific to CICS. To perform the interaction, call the execute() method and use record objects, which are specific to CICS, to hold the data. For example:

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

If you are using a JEE application server, you create the connection factory by configuring it using an administration interface such as the WebSphere administrative console. You set custom properties such as the Gateway daemon connection URL. When you have created a connection factory, enterprise applications can access it by looking it up in the JNDI (Java Naming Directory Interface). This type of environment is called a managed environment, and allows a JEE application server to manage the qualities of service of the connections. For more information about managed environments see your JEE application server documentation.

If you are not using a JEE application server, you must create a managed connection factory and set its custom properties. You can then create a connection factory from the managed connection factory. This type of environment is called a nonmanaged environment and does not allow a JEE application server to manage the qualities of service of 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.

ECI resource adapter

The ECI resource adapter provides a high level CCI interface to the ECI for sending ECI requests to CICS.

The ECI resource adapter is used to connect to CICS server programs and for passing data to COMMAREAs or channels and containers. The resource adapter can be deployed into a JEE application server to allow JEE enterprise applications to access CICS. If JCA is used, connection pooling, security, and transaction context are managed by the JEE application server, not by the application.

CICS Transaction Gateway includes the cicseci.rar resource adapter.

You can use the `cicseci.rar` resource adapter for one-phase commit transactions over any server protocol, and for two-phase commit transactions over IPIC only. For information about the transaction management models that the resource adapter supports see "Transaction management" on page 146.

CICS Transaction Gateway Desktop Edition: Support is not provided for the JCA resource adapter.

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.

The EPI resource adapter can be deployed into a JEE application server to allow JEE enterprise applications to access CICS. When the JCA is used, connection pooling, security, and transaction context are managed by the JEE application server not the application. Automatic Transaction Initiation (ATI) is not supported.

CICS Transaction Gateway Desktop Edition: Support is not provided for the JCA resource adapter.

Managed and nonmanaged environments

The connection, transaction and security qualities of service can either be managed by the application server or they can be provided by the Java application.

In a managed environment, a JEE application server such as WebSphere® Application Server manages the connections, transactions, and security. In this situation, the application developer does not have to provide the code for these.

In a nonmanaged environment, the Java application uses the resource adapters directly without the intervention of a JEE application server. In this situation the application must contain code for the management of connections, transactions and security.

The Common Client Interface

The Common Client Interface (CCI) of the JEE 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 JEE application can send and receive data from an enterprise information system such as CICS.

When you are developing a JEE component you must complete these tasks:
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 run commands on the enterprise information system.

4. Close the interaction and the connection.

The following example shows the JEE CCI interfaces being used to run a command on an enterprise information system:

```
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

The CICS 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 adapter

A JEE developer can use the ECI resource adapter to access CICS programs, using COMMAREAs and channels, to pass information to and from the server.

The table below shows the JCA objects corresponding to the ECI terms listed in “I/O parameters on ECI calls” on page 9. The CCI interfaces for CICS are in the com.ibm.connector2.cics package.

Table 13. ECI terms and corresponding JCA objects

ECI term	JCA object: property
Abend code	CICSTxnAbendException
COMMAREA	Record
Channel	ECIChannelRecord. See "Introduction to channels and containers" on page 13.
Container with a data type of BIT	byte[]
Container with a data type of CHAR	String
ECI timeout	ECIInteractionSpec:ExecuteTimeout. See "Timeout of the ECI request" on page 14.
LUW identifier	JEE transaction
Password or password phrase	ECIConnectionSpec:Password. See "Security in the ECI" on page 15.
Program name	ECIInteractionSpec:FunctionName
Server name	ECIConnectionFactory:ServerName
SocketConnectTimeout	ECIConnection:SocketConnectTimeout
TPNName	ECIInteractionSpec:TPNName. See "ECI and CICS transaction IDs" on page 13.
TranName	ECIInteractionSpec:TranName. See "ECI and CICS transaction IDs" on page 13.
User ID	ECIConnectionSpec:UserName. See "Security in the ECI" on page 15.

Using the ECI resource adapter with channels and containers

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

The ECIChannelRecord 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 in the channel. You can have a combination of container types in one channel. The containers are of 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 ECIChannelRecord.getRecordName method obtains 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 ECIChannelRecord and Records differently, when it receives the data in the execute() method of ECIInteraction.

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

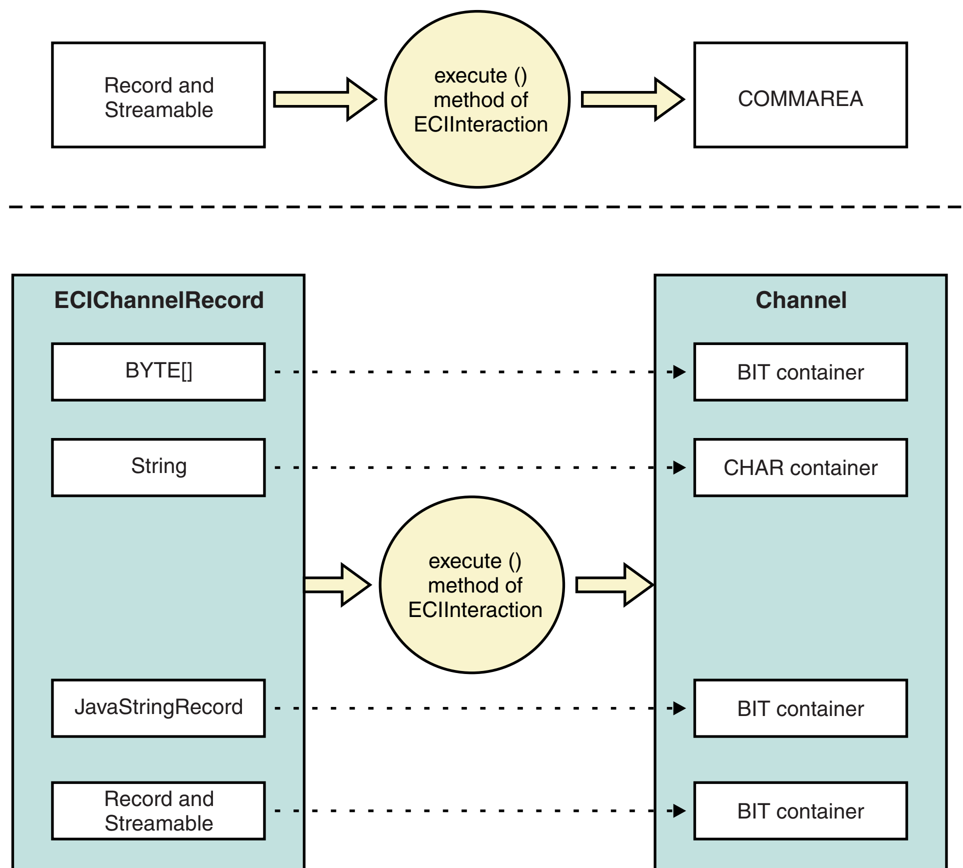


Figure 6. Data conversion by the execute() method of ECIInteraction, depending on whether it receives a Record or ECIChannelRecord

Connection 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 do not work directly with the ECI implementations. Use the ECIConnectionSpec class directly to define the properties of the connection.

The ECIConnectionSpec class allows the JEE component to override the user ID and password set at deployment time. Here is an example of how to code to obtain a connection using this class:

```
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 do not use this directly.

To define the properties of the interaction use the ECIInteractionSpec class directly.

1. Set the FunctionName property to the name of the CICS server program.
2. 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:

   a. When a SYNC_SEND call has been issued with the execute() method of a particular ECIInteraction object, that instance of ECIInteraction cannot issue another SYNC_SEND, or SYNC_SEND_RECEIVE, until a SYNC_RECEIVE has been run.
   b. Simultaneous asynchronous calls to the same connection are permitted, provided they do not result in two asynchronous calls being outstanding in the same transaction scope. In that case an exception is thrown.
   c. If you are using the adapter in local mode with IBM WebSphere Application Server for IBM 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 are issued with SYNCONRETURN, and are outside the scope of the current global transaction. In remote mode, asynchronous calls work in the usual way.
3. If you are using channels and containers, the program receiving the data does not need 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 it is not supplied, a default is used:

   SYNC_SEND, SYNC_SEND_RECEIVE
   Length of input record data

   SYNC_RECEIVE
   The value of ReplyLength
4. Set the ReplyLength property to the length of the data stream to be returned from the Gateway daemon to the JCA application. This value 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, but 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, CICS Transaction Gateway automatically strips trailing zeros from the COMMAREA sent from the Gateway daemon to the JCA application, without needing the size of the data in advance.

   For more information on COMMAREA stripping, see “ECI performance considerations when using COMMAREAs” on page 15.

As with ECIConnectionSpec, you can set properties on the ECIInteractionSpec class at either construction time or by using setters. Unlike ECIConnectionSpec, the ECIInteractionSpec class behaves like a Java bean. So, in a managed environment, your server might 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 default value.

A positive integer
Time in milliseconds.

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 the `javax.resource.cci.Streamable` interface. This 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);
RecordImpl in = new RecordImpl();
RecordImpl out = new RecordImpl();
in.read(stream);
interaction.execute(is, in, out);
```

To retrieve a byte array from the output record, use output records write() method using a ByteArrayOutputStream object as the parameter to reverse the process shown in the above example. The streams toByteArray() method then provides 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 function for your specific JEE components, you can write implementations of the Record interface that allow you to set the contents of the record using the constructor. In this way, you avoid the use of the ByteArrayInputStream used in the above example. A managed environment might provide tools that allow you to build implementations of the Record interface that are customized for your JEE components needs without writing any code.

Transaction management

CICS Transaction Gateway includes a resource adapter that can provide LocalTransaction support or XATransaction support.

The `cicseci.rar` resource adapter provides LocalTransaction support when deployed on any supported JEE application server. It can also provide XATransaction support when deployed with the custom property `xasupport=on` on

any supported JEE application server using local mode to connect to CICS Transaction Server using the IPIC protocol.

To provide for different transactional qualities of service for JEE applications, you can deploy the CICS resource adapter into the JEE application server and create multiple connection factories on it. Each of these connection factories can be configured with a different quality of service.

See the information about Deploying CICS resource adapters in the *CICS Transaction Gateway: UNIX and Linux Administration* for information about installing the resource adapters.

Managed mode

If you are running multiple interactions with CICS using the ECI resource adapter, you might want to group all actions together to ensure that they either all succeed or all fail. The preferred method is to let the JEE application server manage the transactions which are then known as *container-managed transactions*.

Alternatively, you can 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 in the application to be grouped. The `cicseci.rar` resource adapter with xasupport enabled and with bean-managed transactions supports the UserTransaction and LocalTransaction interfaces.

The `cicseci.rar` resource adapter with xasupport disabled and bean-managed transactions supports only the LocalTransaction interface.

For more information on *container-managed transactions* and *bean-managed transactions*, see The Java EE 6 Tutorial.

Nonmanaged mode

When the ECI resource adapter is used in nonmanaged mode, interactions with CICS will be non-transactional by default (synconreturn). Only the LocalTransaction interface is available for applications using nonmanaged mode, and any global transaction context, such as provided by the UserTransaction interface, will be ignored by the ECI resource adapter.

Samples

JEE ECI sample programs are provided in the `<install_path>`\samples subdirectory and as a deployable EAR file in the `<install_path>\deployable` subdirectory.

For more information, see "Resource adapter samples" on page 158.

XA overview

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

The resources that are updated by the transaction can take many forms, such as a database table, a messaging queue, or the resources updated by running 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. With this protocol the atomic nature of the transaction is maintained by ensuring that all resource managers update their resources in a consistent manner. The cicseci.rar supports the two-phase commit XA protocol and enables JEE 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 running of the transaction and for coordinating the resource managers to ensure that the transaction works in an atomic manner.

An example of where this behavior is 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 only if the flight is booked; and vice versa.

If a timeout occurs during an XA transaction it is recommended that the EJB sets the transaction to be rolled back.

For information on using XA transactions with JEE applications see Redpaper *Transactions in J2EE, (REDP-3659-00)* .

IBM WebSphere optimizations

The following optimizations are supported:
- Last participant support
- Only-agent optimization

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

Samples

JCA ECI sample programs are provided in the `samples` directory of your CICS Transaction Gateway installation or as a deployable EAR in the `<install_path>/deployable` directory.

These are documented in “Resource adapter samples” on page 158.

Using the EPI resource adapter

With the CICS EPI resource adapter a JEE component can communicate with CICS transactions that use 3270 data streams for input and output.

The resource adapter provides access to the CICS 3270 interface because each EPIConnection object is treated as a 3270 terminal by CICS. The following table shows the JEE objects corresponding to the EPI terms listed in “Terminal characteristics” on page 21. The CCI interfaces for CICS are in the com.ibm.connector2.cics package.

Note: ATIs are not supported.

Table 14. Terminal attributes and corresponding JEE 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
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 do not work directly with the EPI implementations. Use the EPIConnectionSpec class directly to define the properties of the connection.

Setting terminal attributes

With JEE 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 is 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").

The EPIInteractionSpec class returns the following properties which can be used by the JEE 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 want 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 152.

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 you instantiate an **EPIScreenRecordImpl**.

```
EPIScreenRecord screen = new 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 want, 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. For more information see the information about the **com.ibm.connector2.cics** package in the *Javadoc information* for more information about these interfaces. 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.

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, is the same as the number of bytes on the screen. That is, the number of bytes 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 flow 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 user ID 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 158.

Using the 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 JEE 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 146 for information on managing transactions in a nonmanaged environment.

Note: Nonmanaged JCA applications are not able to exploit XA transaction support. Usage of Container-Managed Transactions (CMT) or Bean-Managed Transactions (BMT) can only influence CICS interactions with managed mode JCA applications.

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 "Saving and reusing connection factories" on page 155). 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 *Deployment parameters for the ECI resource adapters* in the *CICS Transaction Gateway: UNIX and Linux Administration*.

These are accessible using setter and getter methods. The *JEE 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("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 *Deployment parameters for the EPI resource adapters* in the *CICS Transaction Gateway: UNIX and Linux Administration*.

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();
```

Saving and reusing connection factories

When a connection factory has been created it can be saved and reused so that the application does not have to create one.

In a JEE application server environment, IBM recommends that you register your connection factory object, which has links to your enterprise information system connection information, in the JNDI (Java Naming Directory Interface) service. This makes upgrade from nonmanaged to managed Java environments easier because applications can acquire connection factory objects in the same way. However, this might not be possible outside a JNDI environment unless either an LDAP server, or an appropriate JNDI service provider is available within your environment.

Connection factories support the serializable and referenceable Java interfaces. This means that you can decide how to register them in the JNDI. For more information see the *JEE Connector Architecture Specification*.

If you plan to use serializable interfaces see “Tracing issues relating to serialized interfaces and ConnectionFactory objects” on page 158 for more information on how serialization and deserialization of connection factory objects affects the setting of the LogWriter property.

Running the JEE resource adapters in a nonmanaged environment

In a JEE environment all required Java libraries are available however, you might need to ensure that your JEE server adds the jar files to the class path.

The jar files are located in the `<install_path>\classes` subdirectory:

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

Outside a JEE environment, you must ensure that, in addition to the above libraries being listed in the class path, the following Java extensions are also available:

- JAAS (required for EPI resource adapter). JAAS is included with IBM JREs and JDKs by default.
- JCA 1.6 Connector class file (required for ECI resource adapter).
- Java Transaction API (required for XA transactions).

The JCA 1.6 Connector class file and the Java Transaction API (JTA) libraries are available for download from the Oracle Java Web site.

Compiling applications

To enable Java applications to be compiled in a managed or nonmanaged environment, the relevant .jar details must be added to the class path.

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

- cicsjee.jar (required for access to Connection and Interaction Specs)
- ctgclient.jar (required for AIDkey objects)
- ccf2.jar (required for creating LogonLogoff classes)
- screenable.jar (required if using the EPI Screen Record)

The JCA 1.6 Connector class file library is also required, and is available for download from the Oracle Java Web site at http://www.oracle.com/us/sun/index.htm.

Compiling and running JEE components

If you develop a JEE component that returns the EPI screen record parameter, the deployment tool you are using requires two Java archive (.jar) files.

The Java archive files required by the deployment tool are:

```
cicsj2ee.jar
screenable.jar
```

An EJB client that receives an EPI screen record requires that these jar files are defined on the class path.

Integration testing JEE components using an intercept plug-in

You can use integration test tools to intercept ECI and EPI calls from Java Enterprise Edition components. This feature allows such components to be tested without requiring a running CICS Transaction Gateway or CICS server.

Before you begin

To use a Gateway intercept plug-in, the plug-in class must be available on the class path. For information on developing an intercept plug-in, refer to "Integration testing Java applications using a Gateway Intercept plug-in" on page 130.

Procedure

1. Choose one of the following methods to enable the plug-in:
   - For a connection factory instance, set the **interceptPlugin** connection factory custom property to the plug-in class name.
   - Programmatically, use the **setInterceptPlugin** method of the ECIManagedConnectionFactory or EPIManagedConnectionFactory object before you obtain an `ECIConnectionFactory` or `EPIConnectionFactory` object.
2. Add the plug-in class to the JEE Application Server's classpath. The plug-in class can be contained in a jar file.

Related tasks:

"Enabling a Gateway intercept plug-in in a Java SE application" on page 131
You can enable a Gateway intercept plug-in in a Java SE application, programmatically or by using a Java system property.

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 non managed 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 JEE 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`.

JEE tracing

In a nonmanaged environment where the default connection manager is used, the application can set the **LogWriter** property on the class to define where trace messages are sent.

If the connection factory is serialized for storage in a nonmanaged environment, for the `LogWriter` to be used, it must be set after deserialization because 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);
```

Tracing issues relating to serialized interfaces and ConnectionFactory objects

If you use the serializable interface to store your ConnectionFactory objects, the reference to your LogWriter is lost.

If you use a serializable interface to store your ConnectionFactory objects, when you deserialize the interface the ConnectionFactory does not contain a reference to the LogWriter. This is because LogWriters are not serializable.

In a nonmanaged environment, you can ensure that your LogWriters are stored on any connections created from the ConnectionFactory object by configuring the connection using the following code.

```
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 defines that the log is set to output to the `System.err`. Managed environments are unaffected because the trace level applied to the `ManagedConnectionFactory` remains.

Resource adapter samples

The resource adapter samples consist of ECI COMMAREA, channels and containers, and EPI samples.

The samples show you how to use the CICS resource adapters and 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 *Deploying CICS resource adapters* in the *CICS Transaction Gateway: UNIX and Linux Administration*.

CICS Transaction Gateway Desktop Edition: Support is not provided for the JCA resource adapters.

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 want 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 might require further entries if it is to be deployed into an EJB 2.0–compliant environment.

See your JEE 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:

- cicsjee.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. See your JEE 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 JEE Server documentation for details of how to setup the bean with this JNDI name.

You will need to install the server 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 Chapter 18, “Sample programs,” on page 251.

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 want 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 might require further entries if it is to be deployed into an EJB 2.0-compliant environment.

Your JEE 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 JEE 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 JEE 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 takes a series of commands and drives 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 works 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 the information about Deploying CICS resource adapters in the *CICS Transaction Gateway: UNIX and Linux Administration*.

ECI channels and containers sample

The ECI channels and containers sample uses JCA to send an ECI request to a sample channel program in CICS called EC03. The CICS EC03 sample program adds containers to the channel which is then returned.

The sample can call the CICS sample program EC03, either through the ECI resource adapter, or through the ECI XA resource adapter. The sample includes a client application that invokes an enterprise bean. The enterprise bean then issues the ECI request to CICS.

The sample includes the following files:

EC03ChannelBean.java
: The implementation of the EC03 Channel EJB

EC03Channel.java
: The remote interface for the EC03 Channel EJB

EC03ChannelHome.java
: The home interface for the EC03 Channel EJB

EC03ChannelClient.java
: A basic client which calls the EC03 Channel EJB

Enterprise beans have a main body of code and two interfaces. The Remote interface contains the business methods that the bean implements, in this case, the execute() method. The Home interface handles the lifecycle of the enterprise bean.

EC03ChannelClient looks up the enterprise bean as EC03ChannelHome in the JNDI (Java Naming Directory Interface). It then locates an object using the remote interface as a type-cast. When execute() is called on this interface, the method is called remotely on the enterprise bean. The remote method then looks up the resource adapter connection factory (an instance of the resource adapter) under the

name ECI. The method runs EC03 in CICS, passing in a channel with one container. When the ECI call program returns, the containers returned from the program are enumerated and placed into a HashMap which is then returned back to the client application that issued the call.

To use the sample:

1. Deploy the CICS ECI resource adapter (cicseci.rar); this is located in the deployable directory of the CICS Transaction Gateway install path.
2. Create a connection factory with parameters to suit your CICS server environment. .

   Note: The connection factory must have a JNDI name of ECI for the sample to work.
3. Deploy your enterprise bean. This automatically generates code that handles remote method calls to your enterprise bean that are made by the enterprise bean client. This process is specific to your JEE application server, but mainly involves identifying the interfaces to the deployment tool, after setting any properties you need. The properties you will be asked for might include:

   Transaction Type
   This can be set to container-managed, or bean-managed, and determines whether you want to control transactions yourself. The JEE application server manages Container managed transactions. If you are prompted, select Container managed for the sample.

   Enterprise bean Type
   EC03Channel is a stateless session bean.

   JNDI Name
   The enterprise bean client uses JNDI to look up the name of the enterprise bean in the naming directory.

   Resource References
   The enterprise bean refers to a connection factory. You must add the connection factory (as defined in step 2) as a resource reference for this enterprise bean.
4. Run the client application. You can run the client either from the command line or with the launchClient utility (if you are using IBM WebSphere Application Server). The launchClient utility sets up the necessary parameters to communicate with the JNDI directory in IBM WebSphere to find the EC03Channel enterprise bean. The application calls the bean, passes a text string to the EC03 program, and displays the contents of the container that the EC03 program returns.

Assistance in coding CCI applications

When coding CCI applications, refer to the Javadoc and the specification for the JEE Connector Architecture (JCA).

See the Javadoc in the JEE resource adapter

JEE Connector Architecture API

Use this reference for help in coding your CCI applications.

Refer to the *Java EE Connector Architecture 1.6 Specification* document from Java EE Downloads. It contains information such as the exceptions used in CCI applications.

Chapter 12. Programming in C and COBOL

This information describes the external access interfaces specific to C and COBOL.

Overview of the programming interfaces for C and COBOL

C interfaces are provided for the ECI and ESI for building 32-bit applications that can be run in local mode. Separate C interfaces are provided for the ECI and ESI for building 32-bit or 64-bit applications that can be run in remote mode.

A user application must only use a single process to make API requests. However, in environments in which a process can generate several threads, the user application can be multithreaded and each thread can make API requests. On UNIX and Linux, a child process, started from a parent application that has already made a Client API request, cannot make subsequent Client API requests. A child process that makes Client API requests must be started before the parent application has made any Client API requests.

For local mode, interfaces are provided in C and COBOL for the ECI, EPI and ESI. For more information see the following links:

"Making ECI V1 calls from C and COBOL programs"

"Making EPI calls from C and COBOL programs" on page 176

"Making ESI V1 calls from C and COBOL programs" on page 183

For remote mode, a C interface is provided for the ECI and ESI. For more information see "Making ECI V2 and ESI V2 calls from C programs" on page 168.

Making ECI V1 calls from C and COBOL programs

This section describes how to make ECI V1 calls to a CICS server from a local 32-bit COBOL or C Client application. ECI V1 can be used only in local mode.

The following table shows the field names in C and COBOL data structures that correspond to the ECI terms described in "I/O parameters on ECI calls" on page 9.

Table 15. 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-EXTEND-MODE
LUW identifier	ECI_PARMS.eci_luw_token	ECI-PARMS.ECI-LUW-TOKEN
Message qualifier	ECI_PARMS.eci_message_qualifier	ECI-PARMS.ECI-MESSAGE-QUALIFIER
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
Server name	ECI_PARMS.eci_system_name	ECI-PARMS.ECI-SYSTEM-NAME

© Copyright IBM Corp. 1998, 2016

Table 15. ECI terms and corresponding fields in C and COBOL (continued)

ECI term	C structure.field	COBOL structure.field
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 example shows 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

Complete 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 an 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 an LUW, set the **eci_extend_mode** parameter to ECI_NO_EXTEND. To end an 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.

The following table 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 16. 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 Then 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 for Multiplatforms: 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 credentials in ECI V1

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 ESI V1 calls from C and COBOL programs" on page 183 for more information.

Making ECI V2 and ESI V2 calls from C programs

This section describes how to make ECI V2 and ESI V2 calls to a CICS server from a 32-bit or 64-bit C application. ECI V2 and ESI V2 are supported only in remote mode.

Making ECI V2 calls

You can make ECI V2 calls to a CICS server from a C Client application in remote mode.

Use the CTG_ECI_PARMS parameter block structure to communicate with a CICS server. The parameter block fields are used for input and output. To communicate with the CICS server using the Gateway daemon use the CTG_ECI_Execute function. The Remote Client interface requires Version 2 of the ECI Parameter block. Set the ECI parameter block to nulls before setting the input parameter fields. For guidance on how to use the ECI to manage logical units. See "Managing logical units of work" on page 171.

The following table shows the field names in C data structures that correspond to the ECI terms described in "I/O parameters on ECI calls" on page 9.

Table 17. ECI terms and corresponding fields in C in remote mode

ECI term	C structure.field
Abend code	CTG_ECI_PARMS.eci_abend_Code.
Channel	CTG_ECI_PARMS.channel. See "Introduction to channels and containers" on page 13.

Table 17. ECI terms and corresponding fields in C in remote mode (continued)

ECI term	C structure.field
COMMAREA	CTG_ECI_PARMS.eci_commarea.
ECI timeout	CTG_ECI_PARMS.eci_timeout. See "Timeout of the ECI request" on page 14.
LUW control	CTG_ECI_PARMS.eci_extend_mode. See "Program link calls" on page 10.
LUW identifier	CTG_ECI_PARMS.eci_luw_token. See "Managing logical units of work" on page 11.
Message qualifier	CTG_ECI_PARMS.eci_message_qualifier. See "Retrieving replies from asynchronous ECI requests" on page 12.
Password or password phrase	CTG_ECI_PARMS.eci_password_ptr. See "Security in the ECI" on page 15.
Program name	CTG_ECI_PARMS.eci_program_name.
Server name	CTG_ECI_PARMS.eci_system_name.
TPNName	CTG_ECI_PARMS.eci_tpn. See "ECI and CICS transaction IDs" on page 13.
TranName	CTG_ECI_PARMS.eci_transid. See "ECI and CICS transaction IDs" on page 13.
User ID	CTG_ECI_PARMS.eci_userid_ptr. See "Security in the ECI" on page 15.

Making ESI V2 calls

You can make ESI V2 calls to a CICS server from a C Client application in remote mode.

Verifying a password or password phrase

Use the **CTG_ESI_verifyPassword** function to verify a password or password phrase in CICS. Pass in the user ID and password or password phrase to verify, and the name of the CICS server to send the verify request to. If the password or password phrase is verified successfully, information about the user ID is returned in the ESI_DETAILS structure passed to the function. If information about the user ID is not required, NULL can be passed to the function.

```
ESI_DETAILS Details;
int Response;

Response = CTG_ESI_verifyPassword(GatewayToken, Userid, Password,
CicsServer, &Details);
```

Changing a password or password phrase

Use the **CTG_ESI_changePassword** function to change a password or password phrase in CICS. Pass in the user ID and current password or password phrase, the new password or password phrase, and the name of the CICS server to send the change request to. If the password or password phrase is changed successfully, information about the user ID is returned in the ESI_DETAILS structure passed to the function. If information about the user ID is not required, NULL can be passed to the function.

```
ESI_DETAILS Details;
int Response;

Response = CTG_ESI_changePassword(GatewayToken, Userid, CurrentPassword,
NewPassword, CicsServer, &Details);
```

Establishing a connection to a Gateway daemon

To use client applications in C in remote mode, you must establish a connection to the Gateway daemon Client protocol handler using the specified host name and port number.

The following functions establish a remote Client connection to a Gateway daemon:

```
int CTG_openRemoteGatewayConnection(
    char * address,
    int port,
    CTG_ConnToken_t* gwTokPtr,
    int connTimeout
)

int CTG_openRemoteGatewayConnectionApplid(
    char * address,
    int port,
    CTG_ConnToken_t* gwTokPtr,
    int connTimeout,
    char * applid,
    char * applidQualifier
)
```

Use CTG_openRemoteGatewayConnectionApplid in preference to CTG_openRemoteGatewayConnection, as this allows a Client APPLID and APPLID qualifier to be set enabling requests from the Client application to be tracked.

The connection to a Gateway daemon is established using the specified host name and port number. If the connection is successful the Gateway token is returned in the gwTokPtr parameter. The Gateway token is required to interact with that Gateway daemon on further API calls.

The following functions close a remote Client connection to a Gateway daemon:

```
CTG_closeGatewayConnection(CTG_GatewayToken_t * gwTokPtr)

CTG_closeAllGatewayConnections( )
```

The **CTG_closeGatewayConnection** function frees a single Gateway connection held by the API.

The **CTG_closeAllGatewayConnections** function attempts to free all resources held by the API, including open Gateway daemon connections. This function is for use in the event of a severe error because it enables some form of controlled shutdown even if all gateway tokens (gwTokens) have been lost.

Setting the client APPLID and APPLID qualifier using environment variables

The APPLID and APPLID qualifier of the client application can be overridden at run time by setting the environment variables CTG_APPLID and CTG_APPLIDQUALIFIER to the desired values. The environment variable values override any values passed to the **CTG_openRemoteGatewayConnectionApplid** function and are also available to existing ECI V2 and ESI V2 applications without

requiring the application to be recompiled.

Program link calls

For all program link calls, complete the required fields in the ECI parameter block (**CTG_ECI_PARMS** structure). All unused fields should be set to zero.

The **eci_call_type** field must be set to ECI_SYNC or ECI_ASYNC and the **eci_version** field must be set to ECI_VERSION_2A. The constant ECI_VERSION_2 is provided for compatibility with existing applications only and should not be used for new applications.

To specify a user ID and password or password phrase for the program link call, set the **eci_userid_ptr** and **eci_password_ptr** fields.

Program links calls with a COMMAREA

When calling a COMMAREA-based CICS program, provide a pointer to the COMMAREA data in the **eci_commarea** field and the COMMAREA length in the **eci_commarea_length** field.

The commarea_outbound_length and commarea_inbound_length fields can be used to limit the amount of data sent between the application and the CICS Transaction Gateway. For example, if there is a large difference between the size of the data that the CICS program reads from the COMMAREA and the size of the data that the CICS program writes to the COMMAREA.

To perform the program link call, call the CTG_ECI_Execute function, passing a Gateway token and a pointer to the CTG_ECI_PARMS structure:

```
int Response;
Response = CTG_ECI_Execute(gatewayToken, &EciBlock);
```

Program link calls with a channel

When calling a channel-based CICS program, create the channel and any required containers and then set the channel field of the ECI parameter block. For more information see “Using channels and containers in ECI V2 applications” on page 173.

To perform the program link call, call the CTG_ECI_Execute_Channel function, passing a Gateway token and a pointer to the CTG_ECI_PARMS structure:

```
int Response;
Response = CTG_ECI_Execute_Channel(gatewayToken, &EciBlock);
```

All unused fields must be set to zero.

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.

When a transaction is started, an LUW identifier is generated and 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 an LUW, set the **eci_extend_mode** parameter to ECI_NO_EXTEND. To end an 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.

The following table 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 18. 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 Then 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 and the returned **eci_luw_token** is non-zero, the changes made so far are still pending. You must end the logical unit of work with another program link call, either committing or backing out the changes. If the returned **eci_luw_token** is zero, the logical unit of work has ended.

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 are generated by the API.

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 the outstanding reply for the asynchronous call identified by the message qualifier.

ECI_GET_SPECIFIC_REPLY_WAIT
: For a reply solicitation call that gets the outstanding reply for the asynchronous call identified by the message qualifier, the call waits if no reply is available.

Using channels and containers in ECI V2 applications

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

1. Add the following code to your application program, to create a channel:

```
ECI_ChannelToken_t chanToken;
createChannel(&chanToken);
```

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

```
char custNumber[] = {0,1,2,3,4,5};
rc = ECI_createContainer(chanToken, "CUSTNO", ECI_BIT, 0, custNumber,
sizeof(custNumber));
```

   Here is a sample CHAR container that uses the CCSID of the channel:

```
char * company = "IBM";
rc = ECI_createContainer(chanToken, "COMPANY", ECI_CHAR, 0, company,
strlen(company));
```

3. The channel can now be used in an ECI request, as the example shows:

```
CTG_ECI_PARMS eciParms = {0};

eciParms.eci_version = ECI_VERSION_2A;
eciParms.eci_call_type = ECI_SYNC;
strncpy(eciParms.eci_system_name, "CICSA", ECI_SYSTEM_NAME_LENGTH);
eciParms.eci_userid_ptr = "USERNAME";
eciParms.eci_password_ptr = "PASSWORD";
strncpy(eciParms.eci_program_name, "CHANPROG", ECI_PROGRAM_NAME_LENGTH);
eciParms.eci_extend_mode = ECI_NO_EXTEND;
eciParms.channel = chanToken;
```

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

```
ECI_CONTAINER_INFO contInfo;

rc = ECI_getFirstContainer(chanToken, &contInfo);

while (rc == ECI_NO_ERROR) {
    printf("Container %s\n", contInfo.name);
```

```
        if (contInfo.type == ECI_BIT) {
             printf("Type BIT\n");
        } else {
             printf("Type CHAR\n");
        }

        /* Read block of data into buffer */
        ECI_getContainerData(channelToken, contInfo.name, dataBuff,
           sizeof(dataBuff), offset, &bytesRead);

        rc = ECI_getNextContainer(chanToken, &contInfo);
    }
```

Tracing in ECI V2 and ESI V2 applications

Applications should implement an option to enable trace. You can control tracing in ECI and ESI Version 2 applications using the functions and environment variables described here.

You can set trace level, file, data length and offset either by using a function call or by setting an environment variable. Examples of each are shown below. To avoid having to recompile applications, enable trace by setting the environment variable.

Trace level

You can set 5 trace levels:

CTG_TRACE_LEVEL0
: Disables all tracing. This is the default setting.

CTG_TRACE_LEVEL1
: Enables exception trace points. This level of tracing can be set on permanently to provide an error log capability. Messages are written only for system errors, socket errors, and other Gateway connection errors.

CTG_TRACE_LEVEL2
: Enables event trace points and those from lower trace levels.

CTG_TRACE_LEVEL3
: Enables function entry and exit trace points and those from lower trace levels.

CTG_TRACE_LEVEL4
: Enables debug trace points and those from lower trace levels.

Here is an example of the trace level function call:

```
CTG_setAPITraceLevel(CTG_TRACE_LEVEL1);
```

Here is an example of the trace level environment variable:

```
CTG_CLIENT_TRACE_LEVEL=1
```

Trace file

The default trace destination is the standard error stream.

Here is an example of the trace file function call:

```
CTG_setAPITraceFile("filename.trc");
```

Here is an example of the trace file environment variable:

```
CTG_CLIENT_TRACE_FILE=filename.trc
```

If the trace file is not set, trace is written to the standard error stream (stderr).

Trace data length

The trace data length specifies the maximum amount of data that is written to trace when communicating with CICS Transaction Gateway and the trace level is set to CTG_TRACE_LEVEL4. The default setting is 128 bytes.

Here is an example of the trace data length function call:

```
CTG_setAPITraceDataLength(256);
```

Here is an example of the trace data length environment variable:

```
CTG_CLIENT_DATA_LENGTH=256
```

Trace data offset

The trace data offset specifies an offset into data where tracing begins. When combined with the trace data length this allows a specific section of data to be traced, for example a section of data in a COMMAREA. The default setting is zero.

Here is an example of the trace data offset function call:

```
CTG_setAPITraceDataOffset(40);
```

Here is an example of the trace data offset environment variable:

```
CTG_CLIENT_DATA_OFFSET=40
```

Security credentials in ECI V2

The application can specify the user ID and password or password phrase by setting eci_userid_ptr and eci_password_ptr in the ECI V2 parameter block.

The fields eci_userid and eci_password are provided for compatibility with existing applications. New applications must use eci_userid_ptr and eci_password_ptr.

The maximum length of a user ID and password or password phrase depends on the CICS server version and communications protocol type. For more information see your CICS server documentation.

Multithreaded ECI V2 and ESI V2 applications

Considerations when using multithreaded ECI V2 and ESI V2 applications to connect to CICS.

ECI calls using a COMMAREA

It is the responsibility of the application to ensure that application threads do not read or update the contents of the COMMAREA while another thread is performing an ECI call using the same COMMAREA. If applications use the same COMMAREA for simultaneous ECI calls, unpredictable behavior could be experienced.

ECI calls using a channel

For ECI_SYNC calls using a channel, the channel is locked for the duration of the ECI call. For ECI_ASYNC calls using a channel, the channel is locked from the start of the ECI call until the response is retrieved by a subsequent reply solicitation call. While a channel is locked, other application threads block if they attempt to read or update the channel or its containers, or perform further ECI calls using the channel.

Making EPI 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 local 32-bit C or COBOL application. The EPI C interface can be used only in local mode.

The following table shows the field names in C and COBOL data structures that correspond to the terminal attributes described in “Terminal characteristics” on page 21.

Table 19. 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
User ID	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

Each index identifies a combination of server name and terminal ID. The terminal index supplied is the first available integer starting from 0.

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.

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.

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 might 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 user ID and password for the terminal "operator".
- Grant authority, based on the authenticated user ID, to access the resources required for the execution of each transaction.

The frequency with which the user ID 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 22.

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

You can make the **CICS_EpiGetEvent** call in a polling mode by specifying CICS_EPI_NOWAIT for the **`Wait`** parameter.

If no event is waiting to be collected an error code is immediately returned. Use this mechanism 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 can use this mechanism in a multithreaded environment, where a secondary thread can be dedicated to event processing. It can 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 carries 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 call it makes 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 processes 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 might 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 typical 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 might 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 can 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.

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 commands, which instruct the EPI to process the data, or read commands, which instruct the EPI to reply with data.

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.

The following table lists the order codes that occur in 3270 data streams, and shows whether they relate to inbound or outbound data streams, or both.

Table 20. 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*3270 Data Stream Programmer's Reference*. However, the contents of the attribute value field can vary from those defined in the 3270 Data Stream Programmer's Reference*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*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*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*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 ESI V1 calls from C and COBOL programs

You can make ESI V1 calls from a local 32-bit C or COBOL Client application to verify or change passwords for a user ID, known to an external security manager on a CICS server. ESI V1 can be used in local mode only.

The following table shows C and COBOL names that correspond to the ESI terms described in "I/O parameters on ESI calls" on page 27.

Table 21. 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
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 might refer to specific compilers and linkers.

For details of supported compilers, see the *Administration Guide* for your platform.

The following table shows the C header files required depending on the API being used and whether they can be used to build 32-bit and 64-bit applications:

Table 22. C header files

Use	File	32-bit support	64-bit support
ECI V1	`cics_eci.h`	✓	x
EPI	`cics_epi.h`	✓	x
ESI V1	`cics_esi.h`	✓	x
ECI V2	`ctgclient_eci.h` and `ctgclient.h`	✓	✓
ESI V2	`ctgclient_esi.h` and `ctgclient.h`	✓	✓
Type definitions	`cicstype.h`	✓	x

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 contain the `#pragma pack` directive, which must not be changed.

The following table shows the copybook files for COBOL required and whether they can be used to build 32-bit and 64-bit applications:

Table 23. COBOL copybooks

Use	File	32-bit support	64-bit support
ECI V1	cicseci.cbl	✓	x
EPI	cicsepi.cbl	✓	x
ESI V1	cicsesi.cbl	✓	x

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.

Refer to the Chapter 18, “Sample programs,” on page 251 supplied with your environment for examples of compiling and linking programs.

Windows

Some examples showing how to compile and link typical ECI, EPI, and ESI applications in a Windows environment.

For local C applications

- 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.
- 32-bit applications are supported.

For remote C applications (ECI V2 and ESI V2)

- The compiler option /DWIN32 is used to select the correct Windows function and is a standard Win32 option. This options is 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 ctgclient.lib library in addition to the standard C runtime and Windows libraries.
- 32-bit and 64-bit applications are supported.

For statistics applications

- 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 ctgstats.lib library in addition to the standard C runtime and Windows libraries.
- 32-bit applications are supported.

For local 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.

IBM AIX

Some examples showing how to compile and link typical ECI, EPI, and ESI applications in an IBM AIX environment.

For local C applications

- The constant CICS_AIX must be defined to the compiler using the -DCICS_AIX option.
- The application must be linked with the standard IBM AIX libpthreads.a and libc_r.a libraries, as well as the libcclaix.a library.
- 32-bit applications are supported.

For remote C applications (ECI V2 and ESI V2)

- The constant CICS_AIX must be defined to the compiler using the -DCICS_AIX option.
- The application must be linked with the standard IBM AIX libpthreads.a and libc_r.a libraries, as well as the libcgtclient.a library.
- 32-bit and 64-bit applications are supported.

For statistics applications

- 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 libctgstats.a library.
- 32-bit applications are supported.

For local COBOL applications

- 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

Compiling and linking ECI, EPI, and ESI applications in an Solaris environment.

For local C applications

- 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, libc.so, and libcclsol.so libraries.
- 32-bit applications are supported.

For remote C applications (ECI V2 and ESI V2)

- 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, libc.so, and libctgclient.so libraries.
- 32-bit and 64-bit applications are supported.

For statistics applications

- 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, libc.so, and libctgstats.so libraries
- 32-bit applications are supported.

For local COBOL applications
- It is important to use the correct calling convention when calling the ECI or EPI from COBOL. When you are 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 libcclsol.so library. Only 32-bit applications are supported by the API.
- ECI or EPI callback functions are not supported in COBOL applications.

Solaris is supported exclusively on SPARC.

Linux

Some examples showing how to compile and link typical ECI, EPI, and ESI applications in a Linux environment.

To compile and link applications the Linux distribution will require a 32-bit or 64-bit glibc-devel Linux package to be installed.

For local C applications
- 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.
- 32-bit applications are supported.

For remote C applications (ECI V2 and ESI V2)
- 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 `libctgclient.so` library.
- 32-bit and 64-bit applications are supported.

For statistics applications
- 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 `libctgstats.so` library.
- 32-bit applications are supported.

For local COBOL applications
- It is important to use the correct calling convention when calling 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 `libccllnx.so` library file. Only 32-bit applications are supported by the API.
- ECI or EPI callback functions are not supported in COBOL applications.

HP-UX

Compiling and linking ECI, EPI, and ESI applications in a HP-UX environment is supported exclusively on Itanium.

For local C programs and statistics applications

- The constants CICS_HPUX and CICS_HPIT must be defined to the compiler using the –DCICS_HPUX and -DCICS_HPIT options.
- The application must be linked with the system specific library files: `libpthread.so`, `libc.so` and `libcclhpux.so`.
- 32-bit applications are supported.

For remote C applications (ECI V2 and ESI V2)

- The constants CICS_HPUX and CICS_HPIT must be defined to the compiler using the –DCICS_HPUX and -DCICS_HPIT options.
- The application must be linked with the system specific library files: `libpthread.so`, `libc.so` and `libctgclient.so`.
- 32-bit and 64-bit applications are supported.

For statistics applications

- The constants CICS_HPUX and CICS_HPIT must be defined to the compiler using the –DCICS_HPUX and -DCICS_HPIT options.
- The application must be linked with the system specific library files: `libpthread.so`, `libc.so` and `libctgstats.so`.
- 32-bit applications are supported.

For local COBOL programs

- Use the correct calling convention when calling 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, ensure that object files are be linked with the system specific library file: `libcclhpux.so` for Itanium. 32-bit applications only are supported by the API.
- ECI and EPI callback functions are not supported in COBOL applications.

Chapter 13. Programming in C++

This information contains information about the external access interfaces specific to C++.

Overview of the programming interface for C++

C++ classes are provided for the ECI, EPI and ESI for building 32-bit applications that can be run in local mode over TCP/IP or SNA connections.

Writing C++ Client applications

You must ensure that you have the required work environment.

Establishing the working environment

You are provided with C++ (OO) support on AIX, HP-UX on Itanium, Linux, Solaris on SPARC 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 on the left hand menu of the appropriate Web page:

- www.ibm.com/software/cics/ctg

If the version you require is not shown as a flash on the Web page you can either click **View all Flashes** or search for the appropriate version using the following example format:

Supported Software for CICS Transaction Gateway V5.1.

Multi-threading

The CICS Transaction Gateway C++ libraries are not completely threadsafe.

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 usual technique is for each thread to have its own instance of lightweight objects, such as `CclConn`, `CclFlow`, `CclBuf`.

Making ECI 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:

© Copyright IBM Corp. 1998, 2016

- 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 CICS 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 cannot 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 might 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 can be active at the same time. Many servers can be used simultaneously by the same CICS Transaction Gateway.

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 following example 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 7. 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 like this:

```
ECICPO1 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 following example 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 8. 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 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

There are some changes to the main Client application to indicate deferred synchronous reply handling.

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 following example, 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).

To use these features you first must have constructed a Connection object. The two methods available are **verifyPassword** which checks the user ID 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;
}
```

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.

For more information about supported servers and protocols, see the information about Supported software in the *CICS Transaction Gateway: UNIX and Linux Administration*.

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 (ctg.ini). The existence of a server 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 following example 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 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

A table that summarizes the classes provided for programming using the C++ interface.

Table 24. 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 and used for managing updates to recoverable resources.

Making EPI calls from a C++ Client program

In procedural programming, the 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 previous example 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 might 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 might 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 start 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 can send more than one reply in response to a **CclTerminal send** call. More than one **CclTerminal poll** call might therefore be needed to collect all the replies. Use the **CclTerminal state** method to find out whether 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

Sending and receiving data using CICS 3270 screens.

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 might need to be changed to meet local conventions. For example, an IBM AIX server might 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.

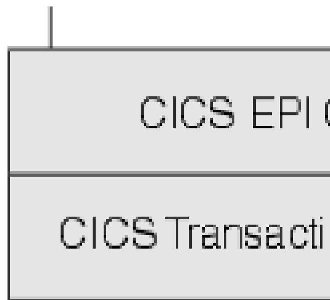


Figure 9. Use of BMS map classes

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 EPI calls from a C++ Client program" on page 197 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 this example 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 10. Sample Map Class—BMS Source

The BMS Conversion Utility generates the C++ class definition (shown in Figure 11 on page 203) 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 11. 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 usual 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 (that is, 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).

To use these features you first must have constructed a **CclTerminal** object which is sign-on incapable, in other words it must have a user ID and password (even if they are null). The two methods available are **verifyPassword** which checks the user ID 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;
}
```

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.

For more information about supported servers and protocols, see the information about Supported software in the *CICS Transaction Gateway: UNIX and Linux Administration* .

C++ EPI classes

This table summarizes the C++ EPI classes by object.

For full details of the methods each class provides, refer to the information about C++ in the *CICS Transaction Gateway for z/OS: Programming Reference*.

Table 25. 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.
Exception	CclException	Encapsulates error information.
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 EPI calls from a C++ Client program" on page 197 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 linking a C++ application

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.

The CICS Transaction Gateway C++ interface supports 32-bit multi-threaded applications.

On a Linux distribution the libstdc++ Linux package is required when compiling a C++ application.

Refer to the sample programs for more information about compiling and linking programs; see "Building C++ sample programs" on page 266.

Define the following macros when compiling C++ applications that use the CICS C++ libraries.

Operating system	Macro
IBM AIX	CICS_AIX
HP-UX on Itanium	CICS_HPUX and CICS_HPIT
Linux	CICS_LNX
Solaris on SPARC	CICS_SOL
Windows	CICS_W32

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 -DCICS_HPIT file.cpp
```

On UNIX and Linux operating systems a C++ application must link to the libcclcp.so shared object file available in `<install_path>/lib`.

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. The Visual Studio exception handling compiler option used with building the CICS Transaction Gateway API DLL is `/EHsc`. When building a C++ application with Visual Studio 2012 link with `cclcpw32.lib`, when building with Visual Studio 2013 link with `cclcpw32vc12.lib`, the lib files are available in `<install_path>/lib`.

Problem determination for C++ Client programs

There is a default exception handler in the handleException method in the CclECI and CclEPI classes. If you are using the async synchronization mode you must override the ECI handleException routine by subclassing CclECI.

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, for example:

```
MyCclECI myeci;
```

or by using the new operator, for example:

```
MyCclECI *pmyeci;
pmyeci = new MyCclECI;
```

Chapter 14. Programming using 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 for use in local mode over TCP/IP or SNA connections.

Writing COM Client applications

The working environment, interfaces and type libraries.

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 uses the type libraries if you register them to each Visual Basic Project. 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 212 for details about project enablement.

VBScript does not use type libraries.

All the COM libraries support automatic registration and unregistration.

Use the Microsoft supplied program REGSVR32 to register or unregister a server.

For example to register or re-register the ECI COM libraries issue the following command:

```
REGSVR32 CCLIECI.DLL
```

to unregister, issue the following command:

```
REGSVR32 /U CCLIECI.DLL
```

© Copyright IBM Corp. 1998, 2016

Windows COM DLL Registration:

The installation and uninstallation processes use Windows REGSVR32 to register and unregister two DLL files as Windows COM libraries.

During installation, the following Windows library files and will be registered as COM libraries:
- *cclieci.dll*
- *ccliepi.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:

Things to consider when passing a COM object to another COM object.

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 must 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 Ccl0Terminal
Set Terminal = New Ccl0Terminal
```

Figure 12. Creating a terminal object in Visual Basic

or you can combine the above into a single statement if you want

```
Dim Terminal as New Ccl0Terminal
```

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 13. 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 the example in “Object creation and interfaces” on page 212, 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 ECI calls from a COM Client program

ECI support that CICS Transaction Gateway provides for COM Clients.

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 information about programming in COM in the *CICS Transaction Gateway for z/OS: 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), user ID 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.

The amount of data sent may differ from the amount of data received, so you must create a COMMAREA that is big enough for the larger of the two.. 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, the 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 an LUW

Managing link calls to a CICS server as a single unit of work.

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 calls use the SetSyncType method on the Flow object.

Deferred synchronous reply handling:

When making a deferred synchronous call, control returns immediately to the Visual Basic program and the reply must be retrieved later.

In the examples in section "Linking to a CICS server program using Visual Basic" on page 214 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 Ccl0Conn
    Set Flow = New Ccl0Flow
    Set Buffer = New Ccl0Buf
    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 information about Password expiry management in the *CICS Transaction Gateway: UNIX and Linux Administration* or the *CICS Transaction Gateway: Windows Administration* for more information on supported servers and protocols.

To use these features you must first create a connection object and start the `Details` method to associate a user ID and password with the object. The two methods available are `Verify Password` that checks the user ID 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, a `Ccl0SecAttr` object is returned. This object provides access to information such as last verified time, expiry time and last access time. If, for example, you queried the last verified time, a `Ccl0SecTime` object is returned and you can 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

The ECI COM classes that the CICS COM servers provide.

More information about the methods these COM classes provide can be found in *CICS Transaction Gateway for Multiplatforms: Programming Reference*.

Table 26. ECI COM classes

COM class	Description
Buffer	Buffer used for passing data to and from a CICS server
Connect	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 EPI calls from a COM Client Program

EPI support that CICS Transaction Gateway provides for COM Clients.

Adding a terminal to CICS

How to add a terminal to CICS using Visual Basic or VBScript.

Adding a terminal to CICS using Visual Basic

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.

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.

```
Dim EPI As Ccl0EPI
Dim Terminal As Ccl0Terminal
Dim Session As Ccl0Session
Dim Screen As Ccl0Screen
Dim Field As Ccl0Field
```

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 the information about programming in COM in the *CICS Transaction Gateway for z/OS: Programming Reference* for additional information.

```
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
```

Adding a terminal to CICS using VBScript

VBScript is again very similar to Visual Basic but differs in how you create the objects.

You do not 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

When you have connected a CclOTerminal object to the required CICS server you can use the **Terminal**, **Session**, **Screen** and **Field** COM classes 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 might send more than one reply in response to a **Terminal.Start** or **Terminal.Send** call. More than one **Terminal.PollForReply** call can therefore be needed to collect all the replies. Use the **Terminal.State** method to find out whether 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** 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 Ccl0Terminal
Set terminal = new Ccl0Terminal
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 information about Password expiry management in the *CICS Transaction Gateway: UNIX and Linux Administration* or the *CICS Transaction Gateway: Windows Administration* 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 user ID and password with the object. The two methods available are **VerifyPassword** which checks the user ID 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

The EPI COM classes that the CICS COM servers provide.

Details of the methods these classes provide are in the information about COM in the ../progref/topics/progref_landing.dita*CICS Transaction Gateway for IBM z/OS: Programming Reference*.

Table 27. 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

How to check for problems with the ECI and EPI classes when starting methods.

Handling exceptions

How to check for problems with the ECI and EPI classes when starting methods, using Visual Basic and VBScript.

One way of handling exceptions is 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. The ErrorWindow method is not the recommended way and exists only to support compatibility with earlier versions for old applications.

The recommended way os handling exceptions 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 object:

```
ECI.SetErrorFormat 1
```

or, for EPI:

```
EPI.SetErrorFormat 1
```

The following sample 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 14. Visual Basic exception handling sample

The following sample 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 15. VBScript exception handling sample

Chapter 15. Developing Microsoft .NET Framework-based applications

The .NET Framework offers a number of advantages when developing remote client applications.

- A consistent model, provided by the .NET class library, for all supported programming languages.
- High levels of security for applications used in remote mode topologies; method-level security using industry standard security technologies can be explicitly defined.
- Separation of application logic from presentation logic for easier maintenance and upgrade.
- Simplified debugging plus the availability of runtime diagnostics.
- Simpler application deployment.

Overview of the programming interface

The Microsoft .NET Framework classes are supported on all Windows platforms and can be used to build 32-bit and 64-bit remote mode applications. The .NET Framework classes are not supported in local mode.

The **GatewayConnection** class represents a connection to CICS Transaction Gateway and the **SslGatewayConnection** class represents a connection that uses Secure Sockets Layer (SSL). The connection is opened in the constructor and remains open until the Close() method is invoked. The class provides two methods for interacting with CICS Transaction Gateway: Flow(request) which flows a request to CICS Transaction Gateway, and ListSystems() which returns a list of all CICS servers that have been defined in CICS Transaction Gateway. Transaction tracking can be enabled on the GatewayConnection class by setting the Applid and ApplidQualifier properties.

The **EciRequest** class represents an ECI call to CICS, and allows data to be flowed in either COMMAREAs or channels. The **Channel** and **Container** classes are used to construct and manage channel and container data. If you specify both a channel and a COMMAREA on an ECI call, the channel is flowed and the COMMAREA is ignored.

The **EsiVerifyRequest** and **EsiChangeRequest** classes provide methods for verifying security credentials and changing passwords and password phrases.

The **Trace** class provides methods for controlling tracing within the API.

Making ECI calls from Microsoft .NET Framework-based programs

Table showing how the .NET properties map to the component parts of an ECI request.

Use the IBM.CTG.EciRequest class to pass details of an ECI request to CICS Transaction Gateway. The following table shows the .NET class properties that

© Copyright IBM Corp. 1998, 2016

correspond to the ECI terms described in "I/O parameters on ECI calls" on page 9. For more information see, the GatewayConnection information in the .NET section of the *Programming Reference*.

ECI term	.NET class property
Abend code	EciRequest.AbendCode.
Channel	EciRequest.Channel. See "Introduction to channels and containers" on page 13.
COMMAREA	EciRequest.SetCommareaData. EciRequest.GetCommareaData. EciRequest.CommareaLength.
ECI return code	EciRequest.EciReturnCode.
ECI timeout	EciRequest.Timeout. See "Timeout of the ECI request" on page 14.
LUW control	EciRequest.ExtendMode. See "Program link calls" on page 10.
LUW identifier	EciRequest.LuwToken. See "Managing logical units of work" on page 11.
Password or password phrase	EciRequest.Password. See "Security in the ECI" on page 15.
Program name	EciRequest.Program.
Server name	EciRequest.ServerName.
TPNName	EciRequest.MirrorTransId. See "ECI and CICS transaction IDs" on page 13.
TranName	EciRequest.TransId. See "ECI and CICS transaction IDs" on page 13.
Userid	EciRequest.UserId. See "Security in the ECI" on page 15.

Making ESI calls from Microsoft .NET Framework-based programs

Table showing how the .NET properties map to the component parts of an ESI request.

Use the IBM.CTG.EsiVerifyRequest and IBM.CTG.EsiChangeRequest classes to pass details of an ESI request to CICS Transaction Gateway. The following table shows the .NET class properties that correspond to the ESI terms described in I/O parameters on ESI calls(link). For more information see, and in the *CICS Transaction Gateway Programming Reference*:

ESI term	.NET class property
Current password or password phrase	EsiVerifyRequest.Password
New password or password phrase	EsiChangeRequest.NewPassword
Server name	EsiVerifyRequest.ServerName
User ID	EsiVerifyRequest.UserId

Using channels and containers in Microsoft .NET Framework-based programs

You can use channels and containers for connections to CICS over the IPIC protocol. You must construct a channel before it can be used in an ECI request.

To construct a channel to hold containers add the following code to your application program:

```
C#:

Channel myChannel = new Channel("CHANNELNAME");

VB.NET:

Dim myChannel As New Channel("CHANNELNAME")
```

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

```
C#:

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

VB.NET:

Dim custNumber() As Byte = {1, 2, 3, 4, 5}
myChannel.CreateContainer("CUSTNO", custNumber)
```

Here is a sample CHAR container:

```
C#:

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

VB.NET:

Dim company As String = "IBM"
myChannel.CreateContainer("COMPANY", company)
```

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

```
C#:

EciRequest eciReq = new EciRequest();
eciReq.ServerName = "CICSA";
eciReq.Program = "CHANPROG";
eciReq.ExtendMode = EciExtendMode.EciNoExtend;
eciReq.Channel = myChannel;

gwyConnection.Flow(eciReq);


VB.NET:

Dim eciReq As New EciRequest()
eciReq.ServerName = "CICSA"
eciReq.Program = "CHANPROG"
eciReq.ExtendMode = EciExtendMode.EciNoExtend
eciReq.Channel = myChannel

gwyConnection.Flow(eciReq)
```

When the request is complete, you can retrieve the contents of the containers in the channel by interpreting the type, as this example shows:

```
C#:

Channel myChannel = eciReq.Channel;

foreach (Container aContainer in myChannel.GetContainers()) {
 Console.WriteLine(aContainer.Name);
 if (aContainer.Type == ContainerType.BIT) {
  byte[] data = aContainer.GetBitData();
 } else if (aContainer.Type == ContainerType.CHAR){
  String data = aContainer.GetCharData();
 }
}
VB.NET:

Dim myChannel As Channel = eciReq.Channel

For Each aContainer In myChannel.GetContainers()
 Console.WriteLine(aContainer.Name)
 If (aContainer.Type = ContainerType.BIT) Then
  Dim data() As Byte = aContainer.GetBitData()
 ElseIf (aContainer.Type = ContainerType.CHAR) Then
  Dim data As String = aContainer.GetCharData()
 End If
Next aContainer
```

Developing ECI and ESI applications based on the Microsoft .NET Framework

How to develop ECI and ESI applications using the .NET Framework.

Developing using Microsoft Visual Studio

If you are developing using Microsoft Visual Studio, you must add a reference to the IBM.CTG.Client.dll assembly.

When you have added the reference, the types in the IBM.CTG namespace can be used to perform ECI and ESI calls to CICS. To avoid the need to fully qualify each type, you can add the IBM.CTG namespace to the imports section of your code.

See Microsoft Visual Studio documentation for further information on creating and building projects.

Compiling and linking from the command line

The .NET Framework provides command line tools for compiling and linking .NET applications. Applications that are written in C# can be compiled and linked using the csc tool:

```
csc /target:exe /out:"AppName.exe" /reference:"IBM.CTG.Client.dll"
"SourceFile.cs"
```

Applications that are written in Visual Basic.NET can be compiled and linked using the vbc tool:

```
vbc /target:exe /out:"AppName.exe" /reference:"IBM.CTG.Client.dll"
"SourceFile.vb"
```

For more information on the csc and vbc command line tools see the Microsoft documentation.

Problem determination for Microsoft .NET Framework-based client programs

Use tracing to help determine the cause of any problems when running .NET Framework-based client programs.

Tracing for Microsoft NET Framework-based client programs

Trace is activated for the `IBM.CTG.Client.dll` either by specifying it as an application configuration file or by using the Trace class.

Trace levels

The following trace levels are available:

CtgTrcDisabled
disables tracing

CtgTrcLevel1
includes exception trace points but nothing else

CtgTrcLevel2
includes event trace points and all CtgTrcLevel1 trace points

CtgTrcLevel3
includes function entry and exit trace points and all CtgTrcLevel1 and CtgTrcLevel2 trace points

CtgTrcLevel4
includes debug trace points and all CtgTrcLevel1, CtgTrcLevel2 and CtgTrcLevel3 trace points (the most verbose tracing level)

Specifying trace in an application configuration file

Trace can be enabled using the CtgTrace trace switch in an application configuration file (an XML file). The switch allows the trace to be specified as an IBM.CTG.TraceLevel value, a System.Diagnostics.TraceLevel value, or an integer between 0 and 4 inclusive. In the following example the switch `value="CtgTrcLevel4"` specifies Level 4 tracing, with tracing of data blocks limited to the first 128 bytes.

```
<?xml version="1.0" encoding="utf-8" ?>
 <configuration>
  <system.diagnostics>
   <switches>
    <add name="CtgTrace" value="CtgTrcLevel4" dataDumpOffset="0"
dataDumpLength="128"/>
   </switches>
  </system.diagnostics>
 </configuration>
```

A sample trace configuration file called `App.config` is included in the SDK package or in `<install_path>\samples\csharp\eci` and `<install_path>\samples\vb\eci` on a Windows machine with CICS Transaction Gateway installed.

Using the Trace class

The Trace class includes the following members:

TraceLevel
gets or sets the trace level

DataDumpOffset
: gets or sets the starting offset in each data blocks when tracing at CtgTrcLevel4

DataDumpLength
: gets or sets the maximum amount of data traced in each data block at CtgTrcLevel4

For more information see the Trace information in the .NET section of the *Programming Reference*.

Chapter 16. Request monitoring exits

Two sets of request monitoring exits exist; one for Java exits, one for C exits.

Java request monitoring exits

Request monitoring exits enable Java user exit code to obtain the details of requests as they are processed by CICS Transaction Gateway and Java client applications.

The following flow topology diagrams show when the request monitoring time stamps are generated depending on the CICS Transaction Gateway configuration. In each diagram, points T1, T2, T3, and T4 show where time stamps are collected for each request.

A request exit running inside the Gateway daemon can be called with the following event types:

RequestEntry
When a request is received by the Gateway daemon.

RequestDetails
Before the request is sent to CICS and after any DSS routing decision has been made.

ResponseExit
When the Gateway daemon sends the response back to the client application.

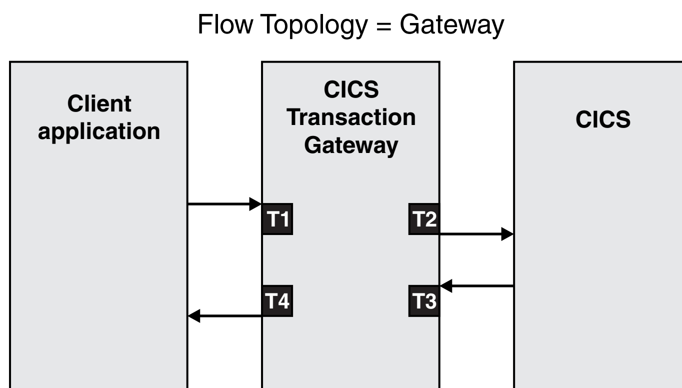


Figure 16. Request flow through the Gateway daemon

A request exit running inside the Java API for both the RemoteClient and LocalClient configurations can be called with the following event types:

RequestEntry
Before the request is sent to the Gateway daemon or CICS.

ResponseExit
After the response is received from the Gateway daemon or CICS.

© Copyright IBM Corp. 1998, 2016

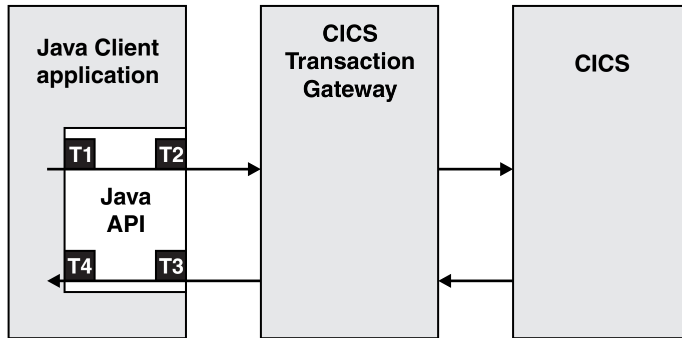


Figure 17. Request flow through the Gateway classes in remote mode

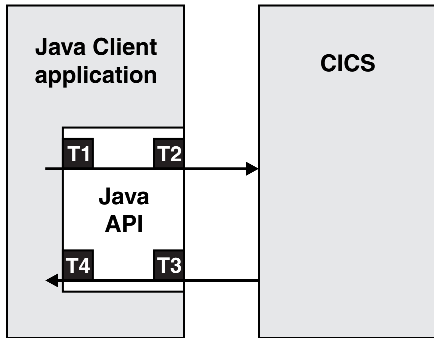


Figure 18. Request flow through the Gateway classes in local mode

Considerations for using request monitoring exits

- Request monitoring exits are configured independently in the Gateway daemon and Java client applications.
- Multiple exits can be configured but the order in which they are called is not defined.
- Exits running in the Gateway daemon are loaded at startup and remain active until disabled using the systems management command or the Gateway daemon is shut down.
- Exits running in a Java client application are loaded when the JavaGateway object is opened and remain active until the JavaGateway object is closed.
- Exits should be coded to have minimal impact on performance.
- An exit that throws a runtime exception or error is disabled.

Writing a monitoring application to use the exits

A request monitoring exit is a Java class that implements the com.ibm.ctg.monitoring.RequestExit interface. When the exit is created the default `no argument` constructor is called. At this point, the exit can create any resources it needs when processing events from the Gateway daemon or Java client application. The eventFired() method is called at each of the exit points; when a

systems management command is sent; or when the exit is shutdown. The shutdown event should be used to release any resources obtained during the lifetime of the exit.

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, RequestDetails, 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.

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 affects performance and must be kept to a minimum. An example exit com.ibm.ctg.samples.requestexit.ThreadedMonitor shows you how to transfer this processing to a separate thread to reduce the impact on performance.

Controlling request monitoring user exits dynamically

Request monitoring exits running in the Gateway daemon can be controlled through the `rmexit` option of the ctgadmin command.

The `enable` and `disable` options allow you to enable or disable all the exits running within the Gateway daemon. When exits are disabled they are not called as part of the Gateway daemon processing.

The `command` option allows you to send system management commands to your request monitoring user exits so you can interact with the request monitoring user exits, to perform tasks such as dynamically starting or stopping a particular user exit.

When you issue a system 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 system 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.BasicMonitor 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 identify 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.

You can use FlowTopology to distinguish between Gateway daemon flows and flows in the Gateway classes, in both local and remote mode. The underlying ECIRequest object is not accessible from the exits.

Flow correlators

Individual flows through the Gateway daemon or Gateway classes have a CtgCorrelator. This correlator is a Java integer which is available at all RequestEvents: RequestEntry to ResponseExit, and can take any value from Integer.MinValue to 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 JavaGateway object of a Client application 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.

For transactions that use IPIC, the origin data is available to associate the flow from a Java application through to a CICS server.

Access to any user correlation data in the COMMAREA is through the PayLoad object, which is read-only, and available only during the eventFired() method.

Transaction correlators

For XA transactions the XID is available.

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, RequestDetails, and ResponseExit, data is available from several fields.

The RequestEvent type is passed with associated data on the eventFired method. Data is represented by a Map object, whose keys are of type RequestData and values are of type Object. The Map object can contain RequestData keys with values of “null”.

The following tables cover the data available for each FlowType.

Non-XA flows at RequestEntry

Data available for non-XA flows at RequestEvent = RequestEntry.

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.

Flow Type	EciStatus	EciSynconreturn	ExtendedModeEci	ExtendedModeCommit	ExtendedModeRollback
Channel 11 on page 238	N	Y	Y	N	N
CicsAbendCode	N	N	N	N	N
CicsReturnCode	N	N	N	N	N
CicsServer	N	N	N	N	N
ClientCtgApplid 7 on page 238, 11 on page 238	Y	Y	Y	Y	Y
ClientCtgApplidQualifier 7 on page 238, 11 on page 238	Y	Y	Y	Y	Y
ClientCtgCorrelator 7 on page 238, 11 on page 238	Y	Y	Y	Y	Y
ClientLocation 1	Y	Y	Y	Y	Y
ClientProtocol 1	Y	Y	Y	Y	Y
ClientType 1	Y	Y	Y	Y	Y
ClientVersion 1, 11 on page 238	Y	Y	Y	Y	Y
CtgApplid	Y	Y	Y	Y	Y
CtgApplidQualifier	Y	Y	Y	Y	Y
CtgCorrelator	Y	Y	Y	Y	Y
CtgReturnCode	N	N	N	N	N
DistributedIdentity	N	Y	Y	Y	Y
FlowTopology	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y
GatewayUrl 5	Y	Y	Y	Y	Y
HttpPath 10 on page 238	N	Y	N	N	N
HttpPayload10 on page 238	N	Y	N	N	N
HttpStatusCode 10 on page 238	N	N	N	N	N
HttpVerb 10 on page 238	N	Y	N	N	N
Location 6	Y	Y	Y	Y	Y
LUW Token	N	N	Y	Y	Y
OriginData 2	N	N	N	N	N
PayLoad 11 on page 238	N	Y	Y	N	N
Program 12 on page 238	N	Y	Y	N	N
RequestReceived	Y	Y	Y	Y	Y
RequestSent 3	N	N	N	N	N
ResponseReceived 3	N	N	N	N	N
ResponseSent	N	N	N	N	N
RetryCount	N	N	N	N	N
Server 8 on page 238, 12 on page 238	Y	Y	Y	Y	Y
TranName TpnName 4, 11 on page 238	N	Y	Y	Y	Y
Userid	N	Y	Y	Y	Y
WireSize 1, 11 on page 238	Y	Y	Y	Y	Y
WorkerWaitTime 1	N	N	N	N	N
XaReturnCode	N	N	N	N	N
XctCurrent 9 on page 238	N	Y	Y	N	N
XctParent 9 on page 238	N	Y	Y	N	N
XctRoot 9 on page 238	N	Y	Y	N	N
Xid	N	N	N	N	N

Note:

1. ClientLocation, ClientProtocol, ClientType, ClientVersion, WorkerWaitTime and WireSize are available only when FlowTopology=Gateway.
2. OriginData is available only for IPIC flows to CICS servers when FlowTopology=Gateway and FlowTopology=LocalClient.
3. The timestamps from and to another system are set only if the flow goes to another system. For EciStatus and for non-IPIC XA flows, except XaEci, this will be when FlowTopology=RemoteClient only.
4. TranName and TpnName are mutually exclusive. Either might be set, but not both.
5. GatewayUrl is available only when FlowTopology=RemoteClient.
6. Location is available exclusively when FlowTopology=Gateway and FlowTopology=RemoteClient.

7. ClientCtgApplid, ClientCtgApplidQualifier, and ClientCtgCorrelator are available to clients that support these data fields when FlowTopology=Gateway. These data fields are supported by Java clients using classes from CICS Transaction Gateway V7.1 and later, ECI V2, and .NET clients using libraries from CICS Transaction Gateway V8.1 and later.
8. Server is only available if a server was specified on the request.
9. XCT data is only available if the Cross Component Trace (XCT) facility is enabled in WebSphere Application Server.
10. HttpPath, HttpStatusCode, HttpVerb and HttpPayload are only available for HTTP requests.
11. Channel, ClientCtgApplid, ClientCtgApplidQualifier, ClientCtgCorrelator, ClientVersion, Payload, TranName, TpnName and WireSize are not available for HTTP requests.
12. Program and Server are only available for HTTP requests if the URI matches a defined web service.

XA flows at RequestEntry

Data available for XA flows at RequestEvent = RequestEntry.

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.

Flow Type	XaStart	XaEci	Xa1PhaseCommit	XaPrepare	XaCommit	XaRollback	XaForget	XaRecover
Channel	N	Y	N	N	N	N	N	N
CicsAbendCode	N	N	N	N	N	N	N	N
CicsReturnCode	N	N	N	N	N	N	N	N
CicsServer	N	N	N	N	N	N	N	N
ClientCtgApplid 7 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgApplidQualifier 7 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgCorrelator 7 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
ClientLocation 1 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
ClientProtocol 1 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
ClientType 1 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
ClientVersion 1 on page 239	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
CtgCorrelator	Y	Y	Y	Y	Y	Y	Y	Y
CtgReturnCode	N	N	N	N	N	N	N	N
DistributedIdentity	N	Y	N	N	N	N	N	N
FlowTopology	Y	Y	Y	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y	Y	Y	Y
GatewayUrl 5 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
HttpPath	N	N	N	N	N	N	N	N
HttpPayload	N	N	N	N	N	N	N	N
HttpStatusCode	N	N	N	N	N	N	N	N
HttpVerb	N	N	N	N	N	N	N	N
Location 6 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
LUW Token	N	N	N	N	N	N	N	N
OriginData 2 on page 239	N	N	N	N	N	N	N	N
PayLoad	N	Y	N	N	N	N	N	N
Program	N	N	N	N	N	N	N	N
RequestReceived	Y	Y	Y	Y	Y	Y	Y	Y
RequestSent 3 on page 239	N	N	N	N	N	N	N	N
ResponseReceived 3 on page 239	N	N	N	N	N	N	N	N
ResponseSent	N	N	N	N	N	N	N	N
RetryCount	N	N	N	N	N	N	N	N
Server 8 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
TranName TpnName 4 on page 239	N	Y	N	N	N	N	N	N
Userid	N	Y	N	N	N	N	N	N
WireSize 1 on page 239	Y	Y	Y	Y	Y	Y	Y	Y
WorkerWaitTime 1 on page 239	N	N	N	N	N	N	N	N
XaReturnCode	N	N	N	N	N	N	N	N

Flow Type	XaStart	XaEci	Xa1PhaseCommit	XaPrepare	XaCommit	XaRollback	XaForget	XaRecover
XctCurrent	N	N	N	N	N	N	N	N
XctParent	N	N	N	N	N	N	N	N
XctRoot	N	N	N	N	N	N	N	N
Xid	Y	Y	Y	Y	Y	Y	Y	N

Note:

1. ClientLocation, ClientProtocol, ClientType, ClientVersion, WorkerWaitTime and WireSize are available only when FlowTopology=Gateway.
2. OriginData is available only for IPIC flows to CICS servers when FlowTopology=Gateway and FlowTopology=LocalClient.
3. The timestamps from and to another system are set if the flow goes to another system. For EciStatus and for non-IPIC XA flows, except XaEci, this will be when FlowTopology=RemoteClient only.
4. TranName and TpnName are mutually exclusive. Either TranName or TpnName can be set, but not both.
5. GatewayUrl is available only when FlowTopology=RemoteClient.
6. Location is available exclusively when FlowTopology=Gateway and FlowTopology=RemoteClient.
7. ClientCtgApplid, ClientCtgApplidQualifier, and ClientCtgCorrelator are available to clients that support these data fields when FlowTopology=Gateway. These data fields are supported by Java clients using classes from CICS Transaction Gateway V7.1 and later, ECI V2, and NET Framework-based clients using libraries from CICS Transaction Gateway V8.1 and later.
8. Server is only available if a server was specified on the request.

Non-XA flows at RequestDetails

Data available for non-XA flows at RequestEvent = RequestDetails.

The RequestDetails request monitoring exit is only applicable when FlowTopology=Gateway.

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.

Flow Type	EciStatus	EciSynconreturn	ExtendedModeEci	ExtendedModeCommit	ExtendedModeRollback
Channel 10 on page 240	N	Y	Y	N	N
CicsAbendCode	N	N	N	N	N
CicsReturnCode	N	Y	Y	N	N
CicsServer	N	Y	Y	Y	Y
ClientCtgApplid 4 on page 240, 10 on page 240	Y	Y	Y	Y	Y
ClientCtgApplidQualifier 4 on page 240, 10 on page 240	Y	Y	Y	Y	Y
ClientCtgCorrelator 4 on page 240, 10 on page 240	Y	Y	Y	Y	Y
ClientLocation	Y	Y	Y	Y	Y
ClientProtocol	Y	Y	Y	Y	Y
ClientType	Y	Y	Y	Y	Y
ClientVersion 10 on page 240	Y	Y	Y	Y	Y
CtgApplid	Y	Y	Y	Y	Y
CtgApplidQualifier	Y	Y	Y	Y	Y
CtgCorrelator	Y	Y	Y	Y	Y
CtgReturnCode	N	N	N	N	N
DistributedIdentity	N	Y	Y	Y	Y
FlowTopology	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y
GatewayUrl	N	N	N	N	N
HttpPath 9 on page 240	N	Y	N	N	N
HttpPayload9 on page 240	N	Y	N	N	N

Flow Type	EciStatus	EciSynconreturn	ExtendedModeEci	ExtendedModeCommit	ExtendedModeRollback
HttpStatusCode 9	N	N	N	N	N
HttpVerb 9	N	Y	N	N	N
Location	Y	Y	Y	Y	Y
LUWToken 6	N	N	Y	Y	Y
OriginData 1	N	Y	Y	Y	Y
PayLoad 10	N	Y	Y	N	N
Program 11	N	Y	Y	N	N
RequestReceived	Y	Y	Y	Y	Y
RequestSent 2	N	N	N	N	N
ResponseReceived 2	N	N	N	N	N
ResponseSent	N	N	N	N	N
RetryCount 7	N	Y	Y	N	N
Server 5, 11	Y	Y	Y	Y	Y
TranName TpnName 3, 10	N	Y	Y	Y	Y
Userid	N	Y	Y	Y	Y
WireSize 10	Y	Y	Y	Y	Y
WorkerWaitTime	Y	Y	Y	Y	Y
XaReturnCode	N	N	N	N	N
XctCurrent 8	N	Y	Y	N	N
XctParent 8	N	Y	Y	N	N
XctRoot 8	N	Y	Y	N	N
Xid	N	N	N	N	N

Note:

1. OriginData is available only for IPIC flows to CICS servers.
2. The timestamps from and to another system are set only if the flow goes to another system.
3. TranName and TpnName are mutually exclusive. Either might be set, but not both.
4. ClientCtgApplid, ClientCtgApplidQualifier, and ClientCtgCorrelator are available to clients that support these data fields when FlowTopology=Gateway. These data fields are supported by Java clients using classes from CICS Transaction Gateway V7.1 and later, ECI V2, and .NET clients using libraries from CICS Transaction Gateway V8.1 and later.
5. Server is only available if one was specified on the request.
6. LuwToken is available only for IPIC flows to CICS servers.
7. RetryCount is available for only the first request of the transaction.
8. XCT data is only available if the Cross Component Trace (XCT) facility is enabled in IBM WebSphere Application Server.
9. HttpPath, HttpStatusCode, HttpVerb and HttpPayload are only available for HTTP requests.
10. Channel, ClientCtgApplid, ClientCtgApplidQualifier, ClientCtgCorrelator, ClientVersion, Payload, TranName, TpnName and WireSize are not available for HTTP requests.
11. Program and Server are only available for HTTP requests if the URI matches a defined web service

Non-XA flows at ResponseExit

Data available for non-XA flows at RequestEvent = ResponseExit.

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.

Flow Type	EciStatus	EciSynconreturn	ExtendedModeEci	ExtendedModeCommit	ExtendedModeRollback
Channel 11 on page 242	N	Y	Y	N	N
CicsAbendCode	N	Y	Y	N	N
CicsReturnCode	N	Y	Y	Y	Y
CicsServer 1 on page 241	N	Y	Y	Y	Y

Flow Type	EciStatus	EciSynconreturn	ExtendedModeEci	ExtendedModeCommit	ExtendedModeRollback
ClientCtgApplid 7, 11 on page 242	Y	Y	Y	Y	Y
ClientCtgApplidQualifier 7, 11 on page 242	Y	Y	Y	Y	Y
ClientCtgCorrelator 7, 11 on page 242	Y	Y	Y	Y	Y
ClientLocation 1	Y	Y	Y	Y	Y
ClientProtocol 1	Y	Y	Y	Y	Y
ClientType 1	Y	Y	Y	Y	Y
ClientVersion 1, 11 on page 242	Y	Y	Y	Y	Y
CtgApplid	Y	Y	Y	Y	Y
CtgApplidQualifier	Y	Y	Y	Y	Y
CtgCorrelator	Y	Y	Y	Y	Y
CtgReturnCode	Y	Y	Y	Y	Y
DistributedIdentity	N	Y	Y	Y	Y
FlowTopology	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y
GatewayUrl 5	Y	Y	Y	Y	Y
HttpPath 10 on page 242	N	Y	N	N	N
HttpPayload10 on page 242	N	Y	N	N	N
HttpStatusCode 10 on page 242	N	Y	N	N	N
HttpVerb 10 on page 242	N	Y	N	N	N
Location 6	Y	Y	Y	Y	Y
LUW Token	N	N	Y	Y	Y
OriginData 2	N	Y	Y	Y	Y
PayLoad 11 on page 242	Y	Y	Y	N	N
Program 12 on page 242	N	Y	Y	N	N
RequestReceived	Y	Y	Y	Y	Y
RequestSent 3	Y	Y	Y	Y	Y
ResponseReceived 3	Y	Y	Y	Y	Y
ResponseSent	Y	Y	Y	Y	Y
RetryCount 1	N	Y	Y	N	N
Server 8 on page 242, 12 on page 242	Y	Y	Y	Y	Y
TranName TpnName 4, 11 on page 242	N	Y	Y	Y	Y
Userid	N	Y	Y	Y	Y
WireSize 1, 11 on page 242	Y	Y	Y	Y	Y
WorkerWaitTime 1	Y	Y	Y	Y	Y
XaReturnCode	N	N	N	N	N
XctCurrent 9 on page 242	N	Y	Y	N	N
XctParent 9 on page 242	N	Y	Y	N	N
XctRoot 9 on page 242	N	Y	Y	N	N
Xid	N	N	N	N	N

Note:

1. CicsServer, ClientLocation, ClientProtocol, ClientType, ClientVersion, RetryCount, WorkerWaitTime and WireSize are available only when FlowTopology=Gateway. RetryCount is only available for the first request of the transaction.
2. OriginData is available only for IPIC flows to CICS servers when FlowTopology=Gateway and FlowTopology=LocalClient.
3. The timestamps from and to another system are set only if the flow goes to another system. For EciStatus and for non-IPIC XA flows, except XaEci, this will be when FlowTopology=RemoteClient only.
4. TranName and TpnName are mutually exclusive. Either can be set, but not both.
5. GatewayUrl is available exclusively when FlowTopology=RemoteClient.
6. Location is available only for FlowTopology=Gateway and FlowTopology=RemoteClient.
7. ClientCtgApplid, ClientCtgApplidQualifier, and ClientCtgCorrelator are available to clients that support these data fields when FlowTopology=Gateway. These data fields are supported by Java clients using

classes from CICS Transaction Gateway V7.1 and later, ECI V2, and .NET clients using libraries from CICS Transaction Gateway V8.1 and later.

8. Server is only available on EciStatus flows if one was specified on the request.
9. XCT data is only available if the Cross Component Trace (XCT) facility is enabled in WebSphere Application Server.
10. HttpPath, HttpStatusCode, HttpVerb and HttpPayload are only available for HTTP requests.
11. Channel, ClientCtgApplid, ClientCtgApplidQualifier, ClientCtgCorrelator, ClientVersion, Payload, TranName, TpnName and WireSize are not available for HTTP requests.
12. Program and Server are only available for HTTP requests if the URI matches a defined web service

XA flows at ResponseExit

Data available for XA flows at RequestEvent = ResponseExit.

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.

Flow Type	XaStart	XaEci	Xa1PhaseCommit	XaPrepare	XaCommit	XaRollback	XaForget	XaRecover
Channel	N	Y	N	N	N	N	N	N
CicsAbendCode	N	Y	N	N	N	N	N	N
CicsReturnCode	N	Y	N	N	N	N	N	N
CicsServer 1 on page 243	N	Y	Y	Y	Y	Y	Y	N
ClientCtgApplid 7 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgApplidQualifier 7 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
ClientCtgCorrelator 7 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
ClientLocation 1 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
ClientProtocol 1 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
ClientType 1 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
ClientVersion 1 on page 243	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
CtgCorrelator	Y	Y	Y	Y	Y	Y	Y	Y
CtgReturnCode	Y	Y	Y	Y	Y	Y	Y	Y
DistributedIdentity	N	Y	N	N	N	N	N	N
FlowTopology	Y	Y	Y	Y	Y	Y	Y	Y
FlowType	Y	Y	Y	Y	Y	Y	Y	Y
GatewayUrl 5 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
HttpPath	N	N	N	N	N	N	N	N
HttpPayload	N	N	N	N	N	N	N	N
HttpStatusCode	N	N	N	N	N	N	N	N
HttpVerb	N	N	N	N	N	N	N	N
Location 6 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
LUW Token	N	N	N	N	N	N	N	N
OriginData 2 on page 243	N	Y	N	N	N	N	N	N
PayLoad	N	Y	N	N	N	N	N	N
Program	N	Y	N	N	N	N	N	N
RequestReceived	Y	Y	Y	Y	Y	Y	Y	Y
RequestSent 3 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
ResponseReceived 3 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
ResponseSent	Y	Y	Y	Y	Y	Y	Y	Y
RetryCount 1 on page 243	N	Y	N	N	N	N	N	N
Server8 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
TranName TpnName 4 on page 243	N	Y	N	N	N	N	N	N
Userid	N	Y	N	N	N	N	N	N
WireSize 1 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
WorkerWaitTime 1 on page 243	Y	Y	Y	Y	Y	Y	Y	Y
XaReturnCode	Y	N	Y	Y	Y	Y	Y	Y
XctCurrent	N	N	N	N	N	N	N	N
XctParent	N	N	N	N	N	N	N	N

Flow Type	XaStart	XaEci	Xa1PhaseCommit	XaPrepare	XaCommit	XaRollback	XaForget	XaRecover
XctRoot	N	N	N	N	N	N	N	N
Xid	Y	Y	Y	Y	Y	Y	Y	N

Note:

1. CicsServer, ClientLocation, ClientProtocol, ClientType, ClientVersion, RetryCount, WorkerWaitTime and WireSize are available only when FlowTopology=Gateway. CicsServer and RetryCount are available only for the first request of the transaction.
2. OriginData is available only for IPIC flows to CICS servers when FlowTopology=Gateway and FlowTopology=LocalClient.
3. The timestamps from and to another system are set only if the flow goes to another system. For non-IPIC XA flows, except XaEci, this will be when FlowTopology=RemoteClient only.
4. TranName and TpnName are mutually exclusive. Either might be set, but not both.
5. GatewayUrl is available only when FlowTopology=RemoteClient.
6. Location is available exclusively when FlowTopology=Gateway and FlowTopology=RemoteClient.
7. ClientCtgApplid, ClientCtgApplidQualifier, and ClientCtgCorrelator are available to clients that support these data fields when FlowTopology=Gateway. These data fields are supported by Java clients using classes from CICS Transaction Gateway V7.1 and later, ECI V2, and NET Framework-based clients using libraries from CICS Transaction Gateway V8.1 and later.
8. Server is only available if a server was specified on the request.

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 the following table, from the `<install_path>\bin` subdirectory, and to call the corresponding entry points.

Table 28. ECI and EPI exits

Exit name	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 can give the exits any valid names. In this book, conventional names are used for the exits.

For upgrade purposes, the CICS Transaction Gateway first looks for a lowercase named object, and then for an uppercase named object. If the objects are not found, no exit processing occurs.

Sample exits and interface definitions

The locations of sample exits and how to edit them for your use.

Sample user exit files are supplied in `<install_path>\samples\c\exits`.

See Chapter 18, "Sample programs," on page 251 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 to make files `cicsecix` and `cicsepix`:
   - On Windows, run `ecix1mak.cmd` and `epix1mak.cmd`
   - On UNIX and Linux, issue the following command to compile the user exit sample programs with the supplied sample `makefile`:

     ```
     make -f samp2.mak
     ```

     To compile the user exit sample programs on Linux on POWER using the IBM XL C compiler, issue the following command:

     ```
     make -f samp2.mak COMPILER=XL
     ```
2. Copy `cicsecix` and `cicsepix` to the `<install_path>\bin` subdirectory

The following header files define the exit interfaces:

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

How to write and use your own user exits.

Follow these rules when writing your own user exits:
- 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 threadsafe.
- Name the primary entry points as follows:

  ECI
  CICS_ECIEXITINIT

  EPI, cicsterm, and cicsprnt
  CICS_EPIEXITINIT

  You can 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 required.

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 244 for information about required exits.

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. 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. **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 request, 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 installation 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.

Chapter 17. Creating a CICS request exit

The CICS request exit is called by CICS Transaction Gateway in remote mode, to select a CICS server name for an ECI or ESI request. The CICS request exit can be used for request retry, dynamic server selection and for rejecting non-valid requests. If the server name returned by a CICS request exit is null, the request is sent to the default CICS server if one is specified in the configuration file (ctg.ini).

Before you begin

If a request fails with a retryable error and the retry limit has not been reached, the Gateway daemon calls the CICS request exit to select an alternative CICS server. The following errors are retryable:

- The specified CICS server is no longer available (ECI_ERR_CICS_DIED or ESI_ERR_CICS_DIED).
- There are insufficient communication resources to complete the request (ECI_ERR_RESOURCE_SHORTAGE or ESI_ERR_RESOURCE_SHORTAGE).
- The specified CICS server is not available (ECI_ERR_NO_CICS or ESI_ERR_NO_CICS).

You can pass a command to a CICS request exit dynamically using the CREXIT administration option; for more information see the *CICS Transaction Gateway for IBM z/OS: Administration Guide*.

About this task

To configure and deploy a CICS request exit use the following steps:

Procedure

1. Create a Java class that implements the **com.ibm.ctg.ha.CICSRequestExit** interface.
2. Compile the Java class and package it into a JAR file.
3. Copy the JAR file to a location in your file system accessible by the Gateway daemon.
4. Update the CLASSPATH environment variable in the Gateway daemon configuration to include the location of the JAR file containing your exit.
5. Specify the fully qualified package name of your exit class by using the **cicsrequestexit** parameter in the configuration file (ctg.ini). For example, to deploy the sample RoundRobinCICSRequestExit, specify this:
```
cicsrequestexit=com.ibm.ctg.samples.ha.RoundRobinCICSRequestExit
```
6. Start the Gateway daemon.

Related information:

CICS request exit

Writing a CICS request exit

Methods implemented by the CICS request exit interface.

The CICS request exit must implement the com.ibm.ctg.ha.CICSRequestExit interface. Two methods defined by the interface must be implemented by the class:

- getRetryCount
- getCICSServer

If the CICS request exit fails to load and then initialize, the Gateway daemon fails to start. When the Gateway daemon loads the CICS request exit class, the default constructor is called, enabling any setup information to be initialized before the CICS request exit is used.

getRetryCount
If the initialization is successful; that is, no exceptions are thrown from the default constructor, the `getRetryCount` method is called to determine how many times a request for a new transaction can be retried following a retryable error. The `getRetryCount` method is called once only, so the value will be constant for the lifetime of the Gateway daemon and used for the start of every transaction.

getCICSServer
The `getCICSServer` method is called by the Gateway daemon at the start of each ECI unit of work and each ESI request to determine the CICS server that the unit of work or request is sent to. A unit of work is started by a SYNCONRETURN ECI request or the first ECI request in an extended LUW. If the request fails with a retryable error and the maximum number of retries has not been reached, the `getCICSServer` method is called again to allow a different CICS server to be used. However, if the request fails and the maximum number of retries has been reached the error from the last request is returned to the Java client application. For more information about the request data available to a `getCICSServer` method, see CICS Transaction Gateway High Availability Exit Programming Reference. The retryable errors are:

- ECI_ERR_NO_CICS
- ECI_ERR_CICS_DIED
- ECI_ERR_RESOURCE_SHORTAGE
- ESI_ERR_NO_CICS
- ESI_ERR_CICS_DIED
- ESI_ERR_RESOURCE_SHORTAGE

InvalidRequestException
If the `getCICSServer` method determines that the request is invalid it can throw a `com.ibm.ctg.ha.InvalidRequestException` that stops the request from being sent to CICS or from being retried. If the request is an ECI request, ECI_ERR_INVALID_CALL_TYPE is returned to the caller. If the request is an ESI request, ESI_ERR_PEM_NOT_ACTIVE is returned.

EventFired
The EventFired method is called if:

- The CICSRequestExit is disabled at shutdown of the Gateway daemon.
- A Gateway daemon receives an administration request for the CICS request exit that includes a command string.

This method is called for each defined ExitEvent. The CICS request exit can selectively process these using the event parameter.

Chapter 18. Sample programs

A wide selection of sample programs for the supported programming languages are included with CICS Transaction Gateway.

The CICS Transaction Gateway product installs sample programs to the `<install_path>/samples` directory. The Software Development Kit supplies the same sample programs to enable application development on a system where CICS Transaction Gateway is not installed.

Sample CICS programs and maps

These samples are for running on a CICS server.

To run the sample programs, the sample CICS programs must be built, defined and installed on your CICS server. The sample CICS programs and maps are in `<install_path>/samples/server`. Refer to your CICS server documentation for details on how to compile, define and install these sample CICS programs.

COBOL samples

EC01.CCP
: This sample returns the current date and time in its COMMAREA.

EC02.CCP
: This sample returns the number of times it has been run in a unit of work in its COMMAREA.

EC03.CCP
: This sample receives CHAR container INPUTDATA and queries the length of the received data; it then returns this length in a BIT container, plus the current date and time and a message indicating success or failure.

EC04.CCP

This sample returns the following data in the COMMAREA:

- Date and timestamp
- CICS APPLID of the region where this program was run
- CICS platform (e.g. CICS TS, IBM TXSeries)
- Version and release of CICS
- CICS USERID that the mirror used when it ran (e.g. CICSUSER)
- CICS mirror used (e.g. CPMI, CSMI)
- CICS transid used
- CICS protocol used (e.g. IPIC, SNA)

This sample can be called from the “Java EciB2 sample” on page 253.

EP01.CCP
: When this sample has been compiled you must define a transaction called EP01 to call EP01.CCP. This program returns the number of times it has been run as the contents of a 3270 data stream.

EP02.CCP
: When this sample has been compiled you must define a transaction called

© Copyright IBM Corp. 1998, 2016

EP02 to call EP02.CCP. This program returns the number of times it has been run as information in fields of a BMS map.

EP03.CCP
This sample is referenced by HighEpiI1.java. The sample issues an EXEC CICS START to run itself again to simulate automatic transaction initiation (ATI). This sample returns two strings of data: "Issued EXEC CICS Start" and "Started as an ATI".

BMS samples

EP02MAP.BMS
This sample is a map source for use with EP02.

MAPINQ.BMS
This sample is a map source for EPIINQ.CCS. The sample contains a mapset MAPINQ, and maps MAPINQ1 and MAPINQ2.

C sample

EPIINQ.CCS
This sample is for an EPI transaction containing two screens. To compile this sample on IBM z/OS, you require **SP** in the translation parameters JCL.

For information about how to build and install these programs, refer to your CICS server documentation.

Java client samples

These samples show the use of the ECI, EPI, and ESI Java APIs.

To use these samples, you must ensure that the required server programs or transactions are installed on your CICS server. These samples do not demonstrate all the techniques required for a large application. They are not templates and should not be used as the basis for developing production applications.

Compiled Java samples

These samples are already compiled and are provided together with their source code.

The samples are in `<install_path>/classes/ctgsamples.jar`.

The source for these samples is in the `<install_path>/samples/java` directory under the package structure, which is in the following form:

`com.ibm.ctg.samples.`*`type_of_sample`*

Running the sample programs

To run the sample programs, ensure that ctgsamples.jar and ctgclient.jar are referenced in your class path. If running the sample in local mode, ctgserver.jar is also required.

These files are in the classes directory.

```
CLASSPATH=<install_path>/classes/ctgsamples.jar
:<install_path>/classes/ctgclient.jar
:<install_path>/classes/ctgserver.jar
```

Alternatively you can run the sample programs by using the Java **-classpath** option, specifying the same information.

When running a sample program, if you provide any command line parameters, you must enter them in the order specified by the usage statement of the particular sample program.

Connecting to CICS Transaction Gateway

You can provide a URL that specifies the location of the CICS Transaction Gateway to which you want to connect.

This should be of the form *protocol://address*. For example, for a remote mode connection using the SSL protocol to a Gateway daemon with address "myserver.test.com":

```
ssl://myserver.ibm.com
```

If you are using IPv6, you must enclose the address in square brackets. For example, for a remote mode connection using the TCP/IP protocol to a Gateway daemon with IPv6 address "2002:914:fc12:632:7:36:66:134":

```
tcp://[2002:914:fc12:632:7:36:66:134]
```

If you want to use local mode, the URL is "local:".

Java ECI base class samples

Samples demonstrating the use of the ECI Java base class API. These samples include simple, intermediate, and advanced ECI Java base classes.

Java EciB1 sample

This sample lists the systems defined in the Gateway daemon configuration file (ctg.ini) and allows you to choose the one to which an ECI request is sent. This request is then sent, and the date and time are returned in ASCII by CICS program EC01, alongside a representation in hexadecimal.

Usage:

```
 java com.ibm.ctg.samples.eci.EciB1 [Gateway Url] [Gateway Port Number]
there [SSL Keyring] [SSL Password]
```

Java EciB2 sample

This sample is used for testing ECI requests sent to CICS. It controls the parameters values from the command line.

Usage:

```
 java com.ibm.ctg.samples.eci.EciB2  [jgate=gateway_URL]
                                      [jgateport=gateway_port]
                                      [clientsecurity=client_security_class]
                                      [serversecurity=server_security_class]
                                      [server=cics_server_name or IPIC_url]
                                      [userid=cics_userid]
                                      [password=cics_password]
                                      [prog<0..9>=prog_name]
                                      [commarea=comm_area]
                                      [commarealength=comm_area_length]
                                      [status]
                                      [trace]
                                      [ascii | ebcdic | asis]
```

You can specify the Gateway URL and relevant ECI request parameters as input to the application, and either call a single CICS program or call multiple CICS programs within one extended LUW. You can control the code page of the COMMAREA flowed on the ECI request as an input parameter. To run the sample in local mode, specify `jgate=local`.

For example, to run the sample in local mode to an IPIC connected server, you can use a command like:

```
java com.ibm.ctg.samples.eci.EciB2 jgate=local: server=tcp://cicssrv1.company.com:1234 prog0=EC01 com
```

You can also use `EciB2` to run `EC04` which returns more data in the COMMAREA. For more information, see "Sample CICS programs and maps" on page 251.

For example:

```
C:\>java com.ibm.ctg.samples.eci.EciB2 jgate=localhost jgateport=2006 prog0=EC04
 server=CICSIPIC commarealength=122 ebcdic

CICS Transaction Gateway Basic ECI Sample 2

 Test Parameters
CICS TG address : localhost:2006
Client security : null
Server security : null
CICS Server     : CICSIPIC
UserId          : null
Password        : null
Data Conversion : IBM037

COMMAREA        : null
COMMAREA length : 122

Number of programs given : 1
  [0] : EC04

Connect to Gateway

Successfully created JavaGateway

Call Programs

About to call : EC04
  COMMAREA    :
  extend_mode : 0
  LUW_token   : 0
  Commarea    : date=05/05/15,time=10:21:45,applid=myapplid,version=CICS TS  5.3
,userid=CICSUSER,mirror=CPMI,transid=CPMI,protocol=IPIC
Return code   : ECI_NO_ERROR(0)
Abend code    : null
Successfully closed JavaGateway

C:\>
```

Java EciB3 sample

This sample is for using with the channels and containers components of the CICS Transaction Gateway API.

Usage:

```
java com.ibm.ctg.samples.eci.EciB3 [Gateway URL] [Gateway Port Number]
[SSL Keyring] [SSL Password]
```

When using remote mode, the sample program connects to a Gateway daemon and obtains a list of available CICS servers. It then flows an ECI request for CICS program EC03 to the selected server.

When using local mode, the sample program prompts for the URL of a CICS TCPIPSERVICE listening for IPIC requests, before flowing an ECI request for CICS program EC03 to that CICS server. This URL is of the form *protocol*://*hostname*:*port*, where *protocol* is "tcp" or "ssl".

Java EciI1 sample

This sample shows the use of the ECI Request classes with an asynchronous extended request and a "callbackable" object.

Usage:

```
java com.ibm.ctg.samples.eci.EciI1 [Gateway URL] [Port]
[SSL keyring] [SSL password]
```

The sample queries the Gateway daemon for a list of servers, then runs transaction EC02 on the selected server.

You can provide a gateway URL and port number, along with an SSL keyring and SSL password as command-line parameters. If you do not provide a URL, the sample programs default to local.

When you start the Gateway daemon, ensure that the ctgsamples.jar file is referenced in the class path.

This sample program also illustrates the use of the ClientCompression and ServerCompression samples. For more information, see "Java security exit data compression samples" on page 273.

Java EciA1 sample

This sample shows the use of the ECI request classes within the framework of a servlet.

To compile EciA1, the servlet packages (2.2) javax.servlet and javax.servlet.http must be referenced in the class path or added to the <install_path>/samples/java directory.

When the servlet is initialized, it reads values supplied for the Gateway URL, SSL classname and SSL password if they have been specified as initialization parameters. Otherwise the default URL is local. The initial page displays the URL of the connected Gateway daemon and a number of areas for user input: Server, Program, CommArea Size, User ID, and Password.

- Server is a combination box containing the names of all the servers listed in the configuration file (ctg.ini).
- Program is a list limited to EC01 and EC02; these must be available on the CICS Server.
- CommArea Size can be set for EC01 only; for EC02 the size is always 50.
- The user ID and password can be specified in the two remaining text areas.

The servlet takes the submitted data and runs the program, automatically backing out if the transaction terminates abnormally, or committing if it runs successfully. The results of the transaction are displayed on a new page.

You can use a servlet properties file to provide initialization parameters. The sample servlet looks for the following case sensitive parameters:

- GatewayURL
- SSLClassname

- SSLPassword

For example:

```
servlet.EciA1.initArgs=GatewayURL=tcp://localhost:2006
```

If your JEE application server requires Java 2 Security permissions, or if you have enabled this facility on your server, you might have to give the permissions described in "Using a Java 2 Security Manager" on page 135.

Refer to the documentation for your JEE application server on setting servlet initialization parameters.

Java EPI base class samples

Samples demonstrating the use of the EPI Java base class API.

Java EpiB1 sample

This sample lists the systems defined in the Gateway daemon configuration file (ctg.ini) and allows you to select the one to which an EPI request is made. This request is then made, and the data returned in the 3270 data stream from the transaction EP01 is displayed on the screen.

Usage:

```
java com.ibm.ctg.samples.epi.EpiB1 [Gateway URL] [Gateway port number]
[debug] [SSL keyring] [SSL password]
```

By default the data is returned as an ASCII string, but if you specify **debug** as the third command line option, the data stream is displayed in hexadecimal alongside the ASCII text.

Java ESI base class samples

Samples demonstrating the use of the ESI Java base class API.

Java EsiB1 sample

This sample lists the systems defined in the Gateway daemon configuration file (ctg.ini) and allows you to select one. Using the ESI API, you then enter a user ID and password for verification on the selected CICS server. Information about the account being used is displayed on the screen.

Usage: java com.ibm.ctg.samples.esi.EsiB1 [Gateway URL] [Gateway port number] [SSL keyring] [SSL password]

Java EPI support class samples

These samples include simple and intermediate EPI support classes.

Java HighEpiB1 sample

This sample shows use of the higher level EPI classes.

Usage:

```
java com.ibm.ctg.samples.epi.HighEpiB1 [GatewayUrl] [Port]
```

The Gateway address and port can be provided as command line parameters. If no URL is provided the sample programs default to local.

The sample program then obtains a list of available servers from which the user selects one. A basic terminal object is constructed. The user is prompted to enter

the name of a transaction, which must be located on the CICS server and must be able to return the terminal to the idle state with the PF3 key. The selected transaction is run on the selected server and the screen displayed to the user. If the terminal is not idle, the sample program sends the PF3 key and then disconnects.

Java HighEpil1 sample

This sample shows the use of ATI (automatic transaction initiation). To use this sample, the CICS program EP03 must be installed on the CICS server. This sample also uses the client and server compression classes.

Usage:

```
java com.ibm.ctg.samples.epi.HighEpiI1 [GatewayUrl] [Port]
```

The Gateway address and port can be provided as command line parameters. If no URL is provided the sample programs default to local: mode.

A list of available servers is displayed. When a server has been selected, the application creates an extended, sign-on incapable terminal, and makes a synchronous connection to it. The user is prompted for a transaction ID; this transaction must be located on the CICS server and it must be able to return the terminal to the idle state with the PF3 key. Note that it is program EP03 (see "Sample CICS programs and maps" on page 251) that demonstrates the use of an ATI.

Because the design of the client application requires the reply to the start transaction to be asynchronous, the sample program must instantiate an object that implements the Session interface. The sample program contains an inner class named ReplyHandler that does this. Therefore before the transaction is sent, the ReplyHandler is instantiated and passed as a parameter to either setSession() or send().

When the sample program sends the transaction, its thread enters a loop that tests to see if a reply has been received. A different application could take this opportunity to do other tasks, but all that is necessary in the sample program is to monitor whether a reply has been received. When the reply is sent, the method handleReply() is run on the ReplyHandler object in its own thread.

This method sets a Boolean value in the sample program thread to indicate that a reply has been received. Upon receipt of the reply, the state of the terminal is interrogated. If it is in server state the client application continues to wait for a reply, otherwise the application continues, depending on the derived state (see the comments in the source for further details). The sample program terminates when the terminal has been returned to the idle state and disconnected.

JEE samples

These samples are based on the JEE (Java 2 Enterprise Edition) standard.

The JEE samples are in `<install_path>/samples/java/com/ibm/ctg/samples/jee`.

JEE ECIDateTime sample

This sample uses the ECI resource adapter, and calls the CICS program EC01. The program uses an enterprise bean that makes CCI calls; a client to the enterprise bean is also provided.

The ECIDateTime sample program includes the following files:

ECIDateTimeBean.java
The enterprise bean ECIDateTime implementation code

ECIDateTime.java
The enterprise bean Remote interface

ECIDateTimeHome.java
The enterprise bean Home interface

JavaStringRecord.java
The sample program record interface that wraps an ECI COMMAREA

ECIDateTimeClient.java
The client for the enterprise bean

Enterprise beans have a main body of code and two interfaces. The Remote interface contains the business methods that the bean implements (in this case, the execute() method.) The Home interface manages the lifecycle of the enterprise bean.

ECIDateTimeClient looks up the enterprise bean as ECIDateTimeBean1 in Java Naming Directory Interface (JNDI), and then narrows the search to a specific object using the remote interface as a type-cast. When execute() is called on this interface, the method is called remotely on the enterprise bean. This remote method in turn looks up the resource adapter's connection factory (an instance of the resource adapter) under the name ECI and runs EC01 in CICS and gets the date and time back as a COMMAREA, which it then returns to the caller (the client application).

To use the sample program:

1. Deploy the CICS ECI resource adapter; this is a file called <install_path>/deployable/cicseci.rar.
2. Create a connection factory with parameters that are valid for your CICS server environment (on IBM WebSphere Application Server, these settings are on the Custom properties tab of the J2C connection factory settings), for more information, see the information about deploying resource adapters in the *CICS Transaction Gateway Administration Guide*. The connection factory must have a JNDI name of ECI for the sample program to work.
3. Deploy the ECIDateTime sample. The sample is a file called ECIDateTime.ear and is located in the `<install_path>/deployable` directory. The deployment process is specific to your JEE application server, but mainly involves identifying the interfaces to the deployment tool, after setting any properties you need. The properties you are asked for might include:

   Transaction type
   This can be set to Container-managed or Bean-managed. This determines whether you want to control transactions yourself. The JEE application server manages Container-managed transactions; if prompted, select this type for the sample program.

   Enterprise bean type
   ECIDateTime is a stateless session bean.

   JNDI name
   The enterprise bean client uses JNDI to look up the enterprise bean. This allows you to find the name of the enterprise bean in the directory. The ECIDateTimeClient requires this name to be set to ECIDateTimeBean1.

Resource references

The enterprise bean refers to another resource, the ECI resource adapter. To enable this to happen, you need to:

a. Deploy a ConnectionFactory for the ECI resource adapter with a JNDI name of ECI.

b. List this ConnectionFactory as a resource reference for this enterprise bean.

4. Run the Client application. You can run it from a command line, but if using IBM WebSphere, use the launchClient utility, which sets up the necessary parameters to allow you to talk to the JNDI directory in IBM WebSphere to find the ECIDateTime enterprise bean. The application returns the current date and time from CICS application EC01.

JEE EPIPlayScript sample

This sample uses the EPI resource adapter and drives a CICS terminal using a command script interpreted by the CCI enterprise bean.

The EPIPlayScript sample program includes the following files:

EPIPlayScriptBean.java
The EPIPlayScript implementation code

EPIPlayScript.java
The Remote interface

EPIPlayScriptHome.java
The Home interface

CICSCESNLogon.java
A sample logon/logoff class to drive the CICS transaction CESN via the terminal

EPIPlayScriptClient.java
The client for the enterprise bean

Enterprise beans have a main body of code and two interfaces.

- The Remote interface contains the business methods that the bean implements (in this case, the playScript(String script) method).
- The Home interface controls the lifecycle of the enterprise bean.

EPIPlayScriptClient looks up the enterprise bean as EPIPlayScriptBean1 in Java Naming Directory Interface (JNDI), and then narrows the search to a specific object using the remote interface as a type-cast. When execute() is called on this interface, the method is called remotely on the enterprise bean. This remote method in turn looks up the resource adapter's connection factory (an instance of the RA) under the name EPI and then uses a simple script to drive a CICS terminal.

The script commands consist of a letter and a bracketed argument. The script is provided as string by the EPIPlayScriptClient application.

Commands are as follows:

S(*xxxx***)**
Start Transaction *xxxx*

F(*nn***)=***text*
Set field number *nn* to *text*.

P(*aid*) Press key *aid*. An AID key is a function key in a CICS terminal, such as Enter or Clearscreen. See the Java Programming Reference information for AIDKey for the available keys.

C(*row,col*)
Set the cursor to *row,col*.

R(*field*)
Returns the value of *field* as a string array.

The script provided by the client is as follows:

```
S(EP02)P(enter)P(enter)P(enter)P(enter)R(2)R(6)
```

To run this sample program:

1. Deploy the CICS EPI resource adapter. This is a file called cicsepi.rar in the <install_path>/deployable directory.
2. Create a connection factory with parameters that are valid for your CICS server environment (on IBM WebSphere Application Server, these settings are on the Custom properties tab of the J2C connection factory settings). See the information about deploying resource adapters in the *CICS Transaction Gateway Administration Guide* for more information. The connection factory must have a JNDI name of "EPI" for the sample program to work.
3. Deploy the EPIPlayScript sample. The sample is a file called EPIPlayScript.ear and is located in the `<install_path>/deployable` directory. You might also need to set these properties:

   EJB type
   EPIPlayScript is a stateful session bean.

   JNDI name
   The enterprise bean is looked up by the enterprise client through JNDI. This allows the enterprise client to find the bean in the directory. The EPIPlayScriptClient requires this name to be set to "EPIPlayScriptBean1".

   Resource References
   The enterprise bean refers to another resource, the EPI resource adapter. To enable this to happen:

   a. Deploy a ConnectionFactory for the EPI resource adapter with a JNDI name of "EPI".
   b. This ConnectionFactory should then be listed as a resource reference for this enterprise bean.
4. If your terminal is sign-on capable, include com.ibm.ctg.samples.jee.CICSCESNLogon as your LogonLogoff class. Place this in the application. See "Using a Java 2 Security Manager" on page 135 for information about security privileges that you might need to grant to your enterprise bean.
5. Run the client application. You can run it from a command line, but if you are using IBM WebSphere, use the launchClient utility. This sets up the necessary parameters to allow the enterprise client to look up the bean in the JNDI directory in IBM WebSphere to find the EPIPlayScript bean. The application returns two fields from the EP02 screen:
   - The number of times the EP02 has been run, which is five (Transaction started, then enter was pressed four times)
   - The current time

JEE EC03Channel sample

This sample calls the CICS program EC03 using the CICS ECI resource adapter. The program uses an enterprise bean that makes ECI calls; a client to the enterprise bean is provided.

The EC03Channel sample program includes the following files:

EC03ChannelBean.java
The implementation of the EC03 channel EJB

EC03Channel.java
The Remote interface for the EC03 channel EJB

EC03ChannelHome.java
The Home interface for the EC03 channel EJB

EC03ChannelClient.java
A basic client which calls the EC03 channel EJB

Enterprise beans have a main body of code and two interfaces. The Remote interface contains the business methods that the bean implements (in this case, the execute() method). The Home interface manages the lifecycle of the enterprise bean.

EC03ChannelClient looks up the enterprise bean as EC03ChannelHome in Java Naming Directory Interface (JNDI), and then narrows the search to a specific object using the remote interface as a type-cast. When execute() is called on this interface, the method is called remotely on the enterprise bean. This remote method in turn looks up the resource adapter's connection factory (an instance of the resource adapter) under the name ECI and runs EC03 in CICS, passing in a channel with one container. When the ECI call program returns, the containers returned from the program are enumerated and placed into a HashMap, which is then returned to the client.

To use the sample program:

1. Deploy the CICS ECI resource adapter (cicseci.rar); this is located in the deployable directory of the CICS Transaction Gateway install path.
2. Create a connection factory with parameters that are valid for your CICS server environment (on IBM WebSphere Application Server, these settings are on the Custom properties tab of the J2C connection factory settings). See the information about deploying resource adapters in the *CICS Transaction Gateway Administration Guide* for more information. The connection factory must have a JNDI name of "ECI" for the sample program to work.
3. Deploy the EC03Channel sample. The sample is a file called EC03Channel.ear and is located in the `<install_path>/deployable` directory. The deployment process is specific to your JEE application server, but mainly involves identifying the interfaces to the deployment tool, after setting any properties you need. The properties you are asked for might include:

   Transaction type
   Can be set to container-managed or bean-managed. This determines whether you want to control transactions yourself. The JEE application server manages Container-managed transactions; if prompted, select this type for the sample program.

   Enterprise bean type
   EC03Channel is a stateless session bean.

JNDI name
: The enterprise bean client uses JNDI to look up the enterprise bean. This allows the enterprise client to find the name of the enterprise bean in the directory.

Resource references
: The enterprise bean refers to a connection factory. To enable this to happen you need to add the connection factory defined in Step 2 on page 261 as a resource reference for this enterprise bean.

4. Run the Client application. You can run it from a command line, but if using IBM WebSphere, use the launchClient utility, which sets up the necessary parameters to allow the enterprise client to look up the bean in the JNDI directory in IBM WebSphere to find the EC03Channel enterprise bean. The application calls the bean, passing a string of text to the EC03 program, and displays the contents of the containers returned.

C remote client samples

These samples demonstrate the use of the ECI V2 and ESI V2 APIs.

The ECI V2 samples are written in C and are installed in the `<install_path>/samples/c/eci_v2` product directory or the `<SDK_path>/api/c/remote/samples/eci_v2` SDK directory.

The ESI V2 sample is written in C and is installed in the `<install_path>>/samples/c/esi_v2` product directory or the `<SDK_path>/api/c/remote/samples/esi_v2` SDK directory.

C ctgecib1 sample

This sample lists the CICS servers defined on a remote CICS Transaction Gateway, and allows you to select the CICS server to which an ECI program call is made. This call is then made and the date and time are returned by the CICS program EC01.

The `ctgecib1` sample is written in C and installed in `<install_path>/samples/c/eci_v2` or the `<SDK_path>/api/c/remote/samples/eci_v2` SDK directory.

To build the product sample, change to the `<install path>/samples/c/eci_v2 directory` directory and issue the following command:

Table 29. Commands used to build the product sample on different platforms

Platform/compiler	32-bit sample	64-bit sample
UNIX and Linux	`make -f samp.mak`	`make -f samp64.mak`
Linux on POWER using IBM XL C	`make -f samp.mak COMPILER=XL`	`make -f samp64.mak COMPILER=XL`
Windows	`ctgecib1mak.cmd 32`	`ctgecib1mak.cmd 64`

To build the SDK sample, change to the `<SDK_path>>/api/c/remote/samples/eci_v2` directory and issue the following command:

Table 30. Commands used to build the SDK sample on different platforms

Platform/compiler	32-bit sample	64-bit sample
UNIX and Linux	make -f make32 OS=<platform>	make -f make64 OS=<platform>
Linux on POWER using IBM XL C	make -f make32 OS=<platform> COMPILER=XL	make -f make64 OS=<platform> COMPILER=XL
Windows	make 32	make 64

where `<platform>` is one of: IBM AIX, HPUX, LinuxI, LinuxP, LinuxZ, Solaris.

Once compiled, the sample program can be executed using the following command:

```
ctgecib1 [host name] [port number]
```

C ctgecib2 sample

This sample lists the CICS servers defined on a remote CICS Transaction Gateway, and allows you to select the CICS server to which a number of asynchronous ECI program calls are made. The CICS program EC01 returns the date and time on each call. A separate thread retrieves the responses for the program calls and displays the results of each call.

The `ctgecib2` sample is written in C and is installed in the `<install_path>/samples/c/eci_v2` product directory or the `<SDK_path>/api/c/remote/samples/eci_v2` SDK directory.

To build the product sample, change to the `<install_path>/samples/c/eci_v2` directory and issue the following command:

Table 31. Commands used to build the product sample on different platforms

Platform/compiler	32-bit sample	64-bit sample
UNIX and Linux	make -f samp.mak	make -f samp64.mak
Linux on POWER using IBM XL C	make -f samp.mak COMPILER=XL	make -f samp64.mak COMPILER=XL
Windows	ctgecib2mak.cmd 32	ctgecib2mak.cmd 64

To build the SDK sample, change to the `<SDK_path>/api/c/remote/samples/eci_v2` directory and issue the following command:

Table 32. Commands used to build the SDK sample on different platforms

Platform/compiler	32-bit sample	64-bit sample
UNIX and Linux	make -f make32 OS=<platform>	make -f make64 OS=<platform>
Linux on POWER using IBM XL C	make -f make32 OS=<platform> COMPILER=XL	make -f make64 OS=<platform> COMPILER=XL
Windows	make 32	make 64

where `<platform>` is one of: IBM AIX, HPUX, LinuxI, LinuxP, LinuxZ, Solaris.

Once compiled, the sample program can be started using the following command:

```
ctgecib2 [host name] [port number] [num calls] [user id] [password]
```

C ctgecib3 sample

This sample lists the systems defined on a remote CICS Transaction Gateway, and allows you to select the one to which an ECI program call is made. The supplied CICS program EC03 is called with a channel and a single CHAR container. The program updates the channel by adding new containers. The sample program lists all the containers that are returned from the EC03 program.

The `ctgecib3` sample is written in C and is installed in the `<install_path>/samples/c/eci_v2` product directory or the `<SDK_path>/api/c/remote/samples/eci_v2` SDK directory.

To build the product sample, change to the `<install_path>/samples/c/eci_v2` directory and issue the following command:

Table 33. Commands used to build the product sample on different platforms

Platform/compiler	32-bit sample	64-bit sample
UNIX and Linux	`make -f samp.mak`	`make -f samp64.mak`
Linux on POWER using IBM XL C	`make -f samp.mak COMPILER=XL`	`make -f samp64.mak COMPILER=XL`
Windows	`ctgecib3mak.cmd 32`	`ctgecib3mak.cmd 64`

To build the SDK sample, change to the `<SDK_path>/api/c/remote/samples/eci_v2` directory and issue the following command:

Table 34. Commands used to build the SDK sample on different platforms

Platform/compiler	32-bit sample	64-bit sample
UNIX and Linux	`make -f make32 OS=<platform>`	`make -f make64 OS=<platform>`
Linux on POWER using IBM XL C	`make -f make32 OS=<platform> COMPILER=XL`	`make -f make64 OS=<platform> COMPILER=XL`
Windows	`make 32`	`make 64`

where `<platform>` is one of: IBM AIX, HPUX, LinuxI, LinuxP, LinuxZ, Solaris.

Once compiled, the sample program can be started using the following command:

```
ctgecib3 [host name] [port number]
```

C ctgesib1 sample

This sample lists the CICS servers defined on a remote CICS Transaction Gateway, and allows you to select a server. You are prompted to input the user ID and password or password phrase which are then verified on the chosen server using the ESI V2 API. The last verified time of the user ID and the password expiry time are displayed.

The `ctgesib1` sample is written in C and is installed in the `<install_path>>/samples/c/esi_v2` or the `<SDK_path>/api/c/remote/samples/esi_v2` SDK directory.

To build the sample, change to the `<install_path>/samples/c/esi_v2` directory and issue the following command:

Table 35. Commands used to build the product sample on different platforms

Platform/compiler	32-bit sample	64-bit sample
UNIX and Linux	`make -f samp.mak`	`make -f samp64.mak`
Linux on POWER using IBM XL C	`make -f samp.mak COMPILER=XL`	`make -f samp64.mak COMPILER=XL`
Windows	`ctgesib1mak.cmd 32`	`ctgesib1mak.cmd 64`

To build the SDK sample, change to the `<SDK_path>/api/c/remote/samples/esi_v2` directory and issue the following command:

Table 36. Commands used to build the SDK sample on different platforms

Platform/compiler	32-bit sample	64-bit sample
UNIX and Linux	`make -f make32 OS=<platform>`	`make -f make64 OS=<platform>`
Linux on POWER using IBM XL C	`make -f make32 OS=<platform> COMPILER=XL`	`make -f make64 OS=<platform> COMPILER=XL`
Windows	`make 32`	`make 64`

where `<platform>` is one of: IBM AIX, HPUX, LinuxI, LinuxP, LinuxZ, Solaris.

When compiled, the sample program can be executed using the following command:

```
ctgesib1 [host name] [port number]
```

C, C++ and COBOL local client samples

These samples show the use of the C and C++ APIs for ECI V1, EPI, and ESI V1 in C, C++ and COBOL applications

The local client samples are installed as part of the CICS Transaction Gateway product in the `<install_path>/samples` directory. They are also available as part of the Software Development Kit in the `<SDK_path>/api/*/local/samples` directories where `*` can be `c`, `cobol`, or `cpp`.

To compile the C and C++ sample programs on 64-bit Linux ensure that the 32-bit compatibility development packages, including `glibc-devel-32bit`, are installed. This applies to all supported architectures.

On Windows, the output from building the samples can be found under the `<product_data_path>/samples` directory.

On UNIX and Linux, the output from building the samples can be found in the current working directory.

The C and C++ sample programs require you to change the user ID and password in the code for your environment.

Building C sample programs

Each sample program is in a subdirectory of the <install_path>/samples/c directory.

The software that is supported for developing C applications is listed in Development environments.

On Windows, the sample programs can be built using the supplied command files. For example, to build the basic ECI sample ecib1.c, go to the eci subdirectory and process the ecib1mak.cmd file. All C samples can be built by changing to the `<install path>/samples/c` directory and processing the sampmak.cmd file.

On UNIX and Linux, you can build all the C sample programs, by changing the directory to the `<install_path>/samples/c` directory and running the sample make file, samp.mak. The make file `<install_path>/samples/c/samp.mak` is a parent make file that invokes the sample make files in each of the subdirectories. The executable files are created in the subdirectories. To process the make file, issue the following command:

```
make -f samp.mak
```

Linux on POWER supports the use of IBM XL C and GNU C. To compile the C sample programs using the IBM XL C compiler, use the command.

```
make -f samp.mak COMPILER=XL
```

Building C++ sample programs

Each sample program is in a subdirectory of the <install_path>/samples/cpp directory.

The software that is supported for developing C applications is listed in Development environments.

On Windows, each sample program can be built using the supplied command files. For example, to build the basic ECI sample ecib1.cpp, go to the eci subdirectory and process the ecib1mak.cmd file. All C++ samples can be built by changing to the `<install_path>/samples/cpp` directory and procssing the sampmak.cmd file.

On UNIX and Linux, you can build all the C++ sample programs by changing to the `<install_path>/samples/cpp` directory and processing the sample make file, samp.mak. The make file `<install_path>/samples/cpp/samp.mak` is a parent make file that invokes the sample make files in each of the subdirectories. The executable files are created in the subdirectories. To process the make file, issue the following command:

```
make -f samp.mak
```

Linux on POWER supports the use of IBM XL C++ and GNU C++. To compile the C++ sample programs using the IBM XL C++ compiler, use the command:

```
make -f samp.mak COMPILER=XL
```

Building COBOL sample programs

Each sample program is in a subdirectory of the <install_path>/samples/cobol directory.

On Windows, the sample programs can be built using the supplied command files. For example, to build the basic ECI sample ecib1.cbl, use the command **ecib1cbl.cmd**, which is in the same directory as the source. Both COBOL samples can be built using the <install_path>\samples\cobol\sampmak.cmd file with either the **MF** or **VAC** parameter. The command file uses either the Micro Focus

COBOL compiler or the IBM COBOL for Windows compiler to compile the program. Specify the compiler to use by issuing an ecib1cbl command, for example:

```
ecib1cbl MF (for Micro Focus)
ecib1cbl VAC (for IBM COBOL for Windows)
```

If you are using the Micro Focus COBOL compiler, refer to the comments in the COBOL samples for information about what you need to do before you can run them.

On UNIX and Linux, both COBOL sample programs can be built using the Korn shell build file <install_path>/samples/cobol/sampmak. The build file sampmak must be run from the <install_path>/samples/cobol directory. To display the usage syntax of sampmak issue the command:

```
./sampmak -?
```

To build the COBOL sample programs you must specify the **mf** or **ibm** parameter (the **ibm** parameter is supported on IBM AIX only). For example: **sampmak mf** (for MicroFocus) or **sampmak ibm** (for IBM COBOL for IBM AIX).

The source code for the ECI sample program is located in the eci subdirectory. The source code for the EPI sample program is located in the epi subdirectory.

C/C++ ECIB1 sample

This sample lists the CICS server defined in the Gateway daemon configuration file (ctg.ini) and allows you to select the one to which an ECI call is made. This call is then made and the date and time are returned by the CICS program EC01.

C/C++ ECII1 sample

This sample lists the CICS servers defined in the Gateway daemon configuration file (ctg.ini) and allows you to select the one to which an ECI call is made. A unit of work is then started and the first ECI call is made to the CICS program EC02.

You then have the choice of running EC02 again or finishing the unit of work. On exit from the loop you can choose either to commit or back out the current unit of work.

In this sample program, all calls are asynchronous and are handled using event functions that are found in the supplied cclcalls.h header file. This header file must be in the `<install_path>/include` directory when you build this sample program.

C/C++ EPIB1 sample

This sample lists the CICS servers defined in the Gateway daemon configuration file (ctg.ini) and allows you to select the one to which an EPI call is made.

You are then prompted to enter the name of a transaction; enter EP01. The call is then made and the 3270 data stream returned by transaction EP01 is displayed on the screen.

C/C++ EPII1 sample

This sample lists the CICS servers defined in the Gateway daemon configuration file (ctg.ini), and allows you to select the one to which an EPI call is made.

A terminal is then installed asynchronously and you are prompted to run a transaction. Enter EP01 or EP02 for the transaction, which is called asynchronously. When a reply has been received, the screen returned by EP01 or EP02 is displayed, and you are prompted for further input options. The header file cclcalls.h is needed for the event handling functions within this sample program.

C/C++ EPIA1 sample

This sample uses multithreading to call the CICS program EP01 or EP02 on every CICS server defined in the Gateway daemon configuration file `ctg.ini`.

The call passes parameters that define the following:

- The number of threads run on each server
- The number of calls to EP01 or EP02 per thread
- Call type, synchronous or asynchronous

This sample program needs the supplied header file `cclcalls.h` for the threading functions used.

C/C++ ECIA1 sample

This sample uses multithreading to call CICS program EC01 on every CICS server defined in the Gateway daemon configuration file, `ctg.ini`.

The call passes parameters that define the following:

- The number of threads run on each server
- The number of calls to EC01 per thread
- Call type (synchronous or asynchronous)

This sample program needs the supplied header file `cclcalls.h` for the threading functions used.

C/C++ ESIB1 sample

This sample lists the CICS servers defined in the Gateway daemon configuration file (ctg.ini) and authenticates you before calling CICS program EC01.

You are prompted to select the server to which an ECI call is made, and on which your security is verified. You then enter a user ID and password for verification on the chosen server using the ESI API. When they have been verified, the user ID and password are used in a synchronous ESI call of EC01. The reply by EC01 is then displayed on the screen.

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 the table in “ECI return notification” on page 270, and the information about ECI_ASYNC call type in the *CICS Transaction Gateway for z/OS: Programming Reference*.

For more information about the status information calls, see the table in “ECI return notification” on page 270, and the information about ECI_STATE_ASYNC call type in the *CICS Transaction Gateway for z/OS: Programming Reference*.

Asynchronous ECI call with notification by message:

The asynchronous ECI call type with notification by message (ECI_ASYNC_NOTIFY_MSG) is available only for programs running on Windows.

The calling application regains control when the ECI accepts the request. This does not indicate that the program has started to run, it indicates 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 information about the message format, see "Reply message formats" on page 270. When this notification has been received, the calling application should use `ECI_GET_SPECIFIC_REPLY` to receive the actual response; the `ECI_GET_REPLY` call type is no longer supported.

The following fields are required parameters for notification by message:

eci_async_notify.window_handle
The handle of the window to which the reply message is posted.

eci_message_id
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 ECI call with notification by semaphore:

The asynchronous ECI call type with notification by semaphore (ECI_ASYNC_NOTIFY_SEM) 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 uses 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.

Fields to support ECI extensions

These 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

ECI notifications are returned to the user of the ECIRequest object.

Table 37. 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.

Table 37. CICS_ExternalCall return codes — environment-dependent extensions (continued)

Return code	Meaning
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 ECI input parameters

The ECI call input parameters shown here are either required (mandatory) or optional other parameters are not applicable.

Table 38 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 38. 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

Table 38. Input parameters for CICS_ExternalCall — environment-dependent extensions (continued)

Parameter, eci_	ANM	ANS	SAM	SAS	SYN	SSN
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

C# and Visual Basic samples based on Microsoft .NET Framework

These samples show how NET Framework-based client applications written in C# and Visual Basic can make ECI and ESI calls to CICS.

C# and Visual Basic EciB1 sample based on Microsoft .NET Framework

This sample lists the CICS servers defined on a CICS Transaction Gateway, and allows you to select the CICS server to which an ECI program call is made. The call is made and the date and time are returned by program EC01.

The sample is provided in C# and Visual Basic .NET. The C# sample is in `<install_path>/samples/csharp/eci`, and the Visual Basic .NET sample is in `<install_path>/samples/vb/eci`.

You can compile the sample using Microsoft Visual Studio or from a Windows command prompt. A Microsoft Visual Studio project file is provided for each language.

To build the sample program from a command prompt, change to the appropriate directory and run the supplied command file EciB1mak.cmd. The file compiles the program for Windows using the C# or Visual Basic .NET compiler which are provided by the Microsoft .NET Framework.

When compiled, you can execute the sample program using the following command:

```
EciB1 [host name] [port number]
```

C# and Visual Basic EciB3 sample based on Microsoft .NET Framework

This sample lists the systems defined on a CICS Transaction Gateway, and allows you to select the one to which an ECI program call is made. The supplied CICS program EC03 is called with a channel and a single CHAR container. The program updates the channel by adding new containers. The sample program lists all the

containers that are returned from the EC03 program. The name, type and data contained within the returned containers is displayed to the console.

The sample is provided in C# and Visual Basic .NET. The C# sample is in `<install_path>/samples/csharp/eci`, and the Visual Basic .NET sample is in `<install_path>/samples/vb/eci`.

You can compile the sample using Microsoft Visual Studio or from a Windows command prompt. A Microsoft Visual Studio project file is provided for each language.

To build the sample program from a command prompt, change to the appropriate directory and run the supplied command file **EciB3mak.cmd**. The file compiles the program for Windows using the C# or Visual Basic .NET compiler which is provided by the Microsoft .NET Framework.

When compiled, you can execute the sample program using the following command:

```
EciB3 [host name] [port number]
```

C# and Visual Basic EsiB1 sample based on Microsoft .NET Framework

This sample lists the CICS servers defined on a CICS Transaction Gateway and allows you to select one. Using the ESI API, you then enter a user ID, and password or password phrase, for verification on the selected CICS server. Information about the account being used is displayed on the screen.

The sample is provided in C# and Visual Basic .NET. The C# sample is in `<install_path>/samples/csharp/esi`, and the Visual Basic .NET sample is in `<install_path>/samples/vb/esi`. You can compile the sample using Microsoft Visual Studio or from a Windows command prompt. A Microsoft Visual Studio project file is provided for each language.

To build the sample program from a command prompt, change to the appropriate directory and run the supplied command file **EsiB1mak.cmd**. The file compiles the program for Windows using the C# or Visual Basic .NET compiler which is provided by the Microsoft .NET Framework.

When compiled, you can execute the sample program using the following command:

```
EsiB1 [host name] [port number]
```

User exit samples

These samples illustrate the use of CICS Transaction Gateway user exits.

Java security exit data compression samples

These samples illustrate the use of the Java security exits principally to compress the data stream between the client application and the Gateway daemon.

- ClientCompression implements ClientSecurity and demonstrates data compression.
- ServerCompression implements ServerSecurity and demonstrates data compression.

- SSLServerCompression implements JSSEServerSecurity and demonstrates how to expose an SSL client certificate.

The source for these samples is in `<install_path>/samples/java/com/ibm/ctg/samples/security`.

Java request monitoring exit samples

These samples show basic and extended use of the CICS Transaction Gateway Java request monitoring exits.

Java BasicMonitor request monitoring exit sample

This sample shows the basic use of the CICS Transaction Gateway request monitoring exits. The sample program writes the data available at each exit point to STDOUT or to a file specified by the Java property `com.ibm.ctg.samples.requestexit.out`.

The class name for this sample is `com.ibm.ctg.samples.requestexit.BasicMonitor`

To enable the sample program on the Gateway daemon you must do the following:

1. Add `ctgsamples.jar` to the class path used when starting the CICS Transaction Gateway.
2. Set the **requestexits** value in the configuration file (`ctg.ini`) to com.ibm.ctg.samples.requestexit.BasicMonitor.
3. On UNIX and Linux the sample program writes the data available at each exit point to STDOUT by default. On Windows the sample program writes the data available at each exit point to file `basicMonitorOutput.txt` in the CICS TG data directory by default. You can override the default output destination by setting the Java property `com.ibm.ctg.samples.requestexit.out` to specify the name of the file to be written.

Java ThreadedMonitor request monitoring exit sample

This sample extends the BasicMonitor sample program. The sample uses a background thread to reduce the overhead for each monitored request. The sample program writes the data available at each exit point to STDOUT or to a file specified by the Java property `com.ibm.ctg.samples.requestexit.out`. Errors are logged to STDERR or to a file specified by the Java property `com.ibm.ctg.samples.requestexit.err`.

The class name of this sample is `com.ibm.ctg.samples.requestexit.ThreadedMonitor`.

To enable the sample program on the Gateway daemon you must do the following:

1. Add `ctgsamples.jar` to the class path used when starting CICS Transaction Gateway.
2. Set the **requestexits** value in the configuration file to `com.ibm.ctg.samples.requestexit.ThreadedMonitor`.
3. On UNIX and Linux, errors are logged to STDERR by default. On Windows, errors are logged to file `threadedMonitorError.txt` in the CICS TG data directory by default. You can override the default error destination by setting the Java property `com.ibm.ctg.samples.requestexit.err` to specify the name of the file to write errors to
4. Errors are written to STDERR by default. To capture data to a file use the Java property `com.ibm.ctg.samples.requestexit.err`, for example:

```
ctgstart -j-Dcom.ibm.ctg.samples.requestexit.err=/hfs.error.file
```

5. An alert is logged for any transactions that take longer than 15 seconds. To change this time, use the Java property `com.ibm.ctg.samples.requestexit.lrt`, for example:

   ```
   ctgstart -j-Dcom.ibm.ctg.samples.requestexit.lrt=5000
   ```

   (time is in milliseconds).

The sample program code details additional optional parameters that can be set.

Java CICS request exit samples

Two sample CICS request exits are provided. The first sample exit returns the CICS server to use for an ECI or ESI request. The second sample exit supports workload management using a round-robin algorithm.

Location of sample files

The source code for the CICS request exit samples is provided in the following location: `<install_path>`/samples/java/com/ibm/ctg/samples/ha.

BasicCICSRequestExit

This sample shows you how to implement a basic CICS request exit. The getCICSServer method returns the CICS server to be used on an ECI or ESI request, based on a predefined server mapping. If the CICS server on the ECI or ESI request is defined in the server mapping, the actual CICS server that it maps to is returned. If the CICS server on the ECI or ESI request is not defined in the server mapping, the CICS server is returned unchanged.

RoundRobinCICSRequestExit

This sample shows you how to implement a CICS request exit to perform workload management. Each time that the getCICSServer method is called, it returns the next CICS server, in a threadsafe manner, from a predefined list. The CICS server specified on the ECI or ESI request by the application is ignored. The retry count is set so that each server in the list is called at most once for each request.

Using the CICS request exit samples

Before using these samples modify the code so that the samples reference known CICS servers.

When these changes have been made, compile the sample, for example by using the **javac** command.

When configuring each sample exit for use in a specific environment refer to the following information:

BasicCICSRequestExit

The constructor for this class populates a hash table with mappings between a name that would be used by the Java client application and an actual CICS server. Change the contents of the hash table so that there is a mapping between the CICS server specified on the ECI or ESI request, by the Java client application, and an actual CICS server.

RoundRobinCICSRequestExit

The list of available CICS servers is contained in the serverList array. Change the values stored in this array to a list of actual CICS servers.

C ECI and EPI user exit template samples

These samples provide templates containing the ECI and EPI user exit functions, their return codes and meanings. The templates support the writing of user exit applications but do not contain any function.

The templates are:

- ecix1.c
- epix1.c

When you have built a user exit application, put it in `<install_path>/bin`. In addition to the standard ECI or EPI header files, the sample programs require the cicsecix.h or cicsepix.h header files in `<install_path>/include`. For more information on how to install a user exit, see "ECI and EPI C exits" on page 243.

C ECI and EPI user exit samples

These samples redirect requests to a server other than that specified by the Client application.

If your server configuration changes, you can continue to use your existing applications, without changing the application code. The samples are:

- ecix2.c
- epix2.c

You define server aliases through two arrays: redirectFrom and redirectTo. Any request to a server that appears in redirectFrom is redirected to the server defined in the corresponding entry in the redirectTo array. For example, the sample programs redirect any requests sent to a server named SERVER2 to a server named SERVERAA. Requests sent to SERVER4 are sent to SERVERBB.

To change the names of the servers in the sample program to match your configuration:

1. Set the NUM_SERVERS constant to the number of elements in your arrays (set to 4 in the sample programs)
2. Change the elements of the requestFrom and requestTo arrays. The number of elements must be the same in each array.
3. Recompile the sample program.

When you have built a user exit application, put it in `<install_path>/bin`. In addition to the standard ECI or EPI header files, the sample programs require the cicsecix.h or cicsepix.h header files in `<install_path>/include`. For more information on how to install a user exit, see "ECI and EPI C exits" on page 243.

Building C user exit samples

Two "make" files are provided for you to build the user exit samples.

The samples are in subdirectories of `<install_path>/samples/c/exits`.

The user exit directory `<install_path>/samples/c/exits` provides two "make" files. The user exit template sample programs ecix1.c and epix1.c are built by the parent makefile `<install_path>/samples/c/samp.mak`.

The user exit redirection sample programs `ecix2.c` and `epix2.c` can be built on Windows using the supplied command files. For example, to build ecix2.c, use the command ecix2mak.cmd, which is in the same subdirectory as the source.

To build the user exit redirection sample programs `ecix2.c` and `epix2.c` on UNIX and Linux, do the following:

1. Change to `<install_path>/samples/c/exits`.
2. To compile the user exit sample programs using the supplied sample makefile, issue the following command:
   ```
   make -f samp2.mak
   ```
   To compile the user exit sample programs on Linux on POWER using the IBM XL C compiler, issue the following command:
   ```
   make -f samp2.mak COMPILER=XL
   ```

C and Java statistics API samples

These samples show use of the statistics API for C and Java clients.

C ctgstat1 statistics API sample

This sample shows how Gateway daemon statistics can be obtained by C clients.

The statistics sample program is written in C and can be found in the `<install_path>/samples/c/stats` directory or the `<SDK_path>/spi/statistics/c/samples` SDK directory.

The ctgstat1 C sample program demonstrates the following functions:

1. Connecting to the statistical API port.
2. Querying running Gateway daemons for statistics in the connection manager resource group.
3. Obtaining values for these statistics.
4. Retrieving and displaying information about the Gateway daemon running time and the total number of requests made.

Building the product sample

1. Change directory to `<install_path>/samples/c/stats`.
2. On Windows, run the supplied command file
   ```
   ctgstat1mak.cmd
   ```
3. On UNIX and Linux, issue the following command:
   ```
   make -f samp.mak
   ```
4. On Linux on POWER using the IBM XL C compiler, issue the following command:
   ```
   make -f samp.mak COMPILER=XL
   ```

Building the SDK sample

1. Change directory to `<SDK_path>/spi/statistics/c/samples`.
2. On Windows, run the command file
   ```
   make.cmd
   ```

3. On UNIX and Linux, issue the command:
```
make OS=<platform>
```
4. On Linux for POWER using the IBM XL C compiler, issue the command:
```
make OS=<platform> COMPILER=XL
```

where `<platform>` is one of: IBM AIX, HPUX, LinuxI, LinuxP, LinuxZ, Solaris.

Java Ctgstat1 statistics API sample

This sample shows how Gateway daemon statistics can be obtained by Java clients.

The statistics sample program is written in Java and is in `samples/java/com/ibm/ctg/samples/stats/Ctgstat1.java`.

The Ctgstat1 Java sample program demonstrates the following functions:
1. Connecting to the statistical API port.
2. Querying running Gateway daemons for statistics in the connection manager resource group.
3. Obtaining values for these statistics.
4. Retrieving and displaying information about the Gateway daemon running time and the total number of requests made.

A precompiled version of com.ibm.ctg.samples.stats.Ctgstat1 is included in the Java archive file `classes/ctgsamples.jar`.

The `ctgstats.jar` file must be on the class path.

For information about the API see "Statistics Java API" on page 108.

Java Ctgstat2 statistics recording sample

This sample shows how an application can parse an XML statistics log file, validate it against the `ctgstatslog.xsd` XML schema file and output statistics in a simple text format.

The supplied sample `Ctgstat2` demonstrates a simple application using the XML statistics log file and XML schema file. The XML schema file, `ctgstatslog.xsd`, will need to be in the same directory as the XML statistics log file.

The statistics sample program is written in Java and supplied in `samples/java/com/ibm/ctg/samples/stats/Ctgstat2.java`. A precompiled version of `com.ibm.ctg.samples.stats.Ctgstat2` is included in the Java archive file `classes/ctgsamples.jar`.

Usage:
```
java com.ibm.ctg.samples.stats.Ctgstat2 filename
```

where *filename* is the name of the XML statistics file.

Part 2. Appendixes

© Copyright IBM Corp. 1998, 2016

Glossary

This glossary defines the terms and abbreviations used in CICS Transaction Gateway and in the information centers.

A

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 systems.

APAR See *Authorized program analysis report*.

API See *application programming interface*.

APPC See *Advanced program-to-program communication*.

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

1. On CICS Transaction Gateway: The application identifier that is used to identify connections on the CICS server and tasks in a CICSplex. See also *APPLID qualifier* and *fully qualified APPLID*.
2. On CICS Transaction Server: 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.

APPLID qualifier
: Optionally used as a high-level qualifier for the APPLID to form a fully qualified APPLID. See also *APPLID* and *fully qualified APPLID*.

ARM See *automatic restart manager*.

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.

Authorized Program Analysis Report (APAR)
: A request for correction of a defect in a current release of an IBM-supplied program.

autoinstall
: A method of creating and installing resources dynamically as terminals log on, and deleting them at logoff.

© Copyright IBM Corp. 1998, 2016

automatic restart manager (ARM)
An IBM 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.

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.

B

Basic Mapping Support
Basic mapping support is an interface between CICS and CICS application programs that move 3270 data streams to and from a terminal. The format of the input and output display data is defined by the BMS commands.

bean A definition or instance of a JavaBeans component. See also *JavaBeans*.

bean-managed transaction
A transaction where the JEE 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).

BMS see Basic Mapping Support

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*.

C

CA See *certificate authority*.

callback
A way for one thread to notify another application thread that an event has happened.

CCIN The CCIN transaction is invoked by the Client daemon, for each TCP/IP or SNA connection established. CCIN installs a Client connection on the CICS server.

CCSID
Coded Character Set Identifier. A 16-bit number that includes a specific set of encoding scheme identifiers, character set identifiers, code page identifiers, and other information that uniquely identifies the coded graphic-character representation.

certificate authority (CA)
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
The Client daemon, the EXCI (External CICS Interface), and the IPIC (IP Interconnectivity) protocol are collectively called the 'CICS connectivity components'. The Client daemon handles the TCP/IP and the SNA protocols.

CICS Request Exit
An exit that is invoked by the CICS Transaction Gateway for IBM z/OS at run time to determine which CICS server to use.

CICS server name
A defined server known to CICS Transaction Gateway.

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 interact with 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 that uses one or more of the CICS Transaction Gateways APIs.

Client daemon
The Client daemon manages TCP/IP and SNA connections to CICS servers on UNIX, Linux, and Windows. It processes ECI, EPI, and ESI requests, sending and receiving the appropriate flows to and from the CICS server to satisfy Client application requests. It can support concurrent requests to one or more CICS servers. The CICS Transaction Gateway initialization file defines the operation of the Client daemon and the servers and protocols used for communication.

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.

COMMAREA
See *communication area*.

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.

CTIN The CTIN transaction is invoked by the Client daemon to install a Client terminal definition on the CICS server.

D

daemon
: A program that runs unattended to perform continuous or periodic systemwide functions, such as network control. A daemon can 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*.

default CICS server
: The CICS server that is used if a server name is not specified on an ECI, EPI, or ESI request. The default CICS server name is defined as a product wide setting in the configuration file (ctg.ini).

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.

distinguished name
: The name that uniquely identifies an entry in a directory. A distinguished name is made up of attribute:value pairs, separated by commas. The format of a distinguished name is defined by RFC4514. For more information, see `http://www.ietf.org/rfc/rfc4514.txt`. See also *realm name* and *identity propagation*.

distributed application
: An application for which the component application programs are distributed between two or more interconnected processors.

distributed identity
: User identity information that originates from a remote system. The distributed identity is created in one system and is passed to one or more other systems over a network. See also *distinguished name* and *realm name*.

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.

DLC See *data link control*.

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 IP 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.

dynamic server selection (DSS)
The mapping of a logical CICS server name to an actual CICS server name at run time.

E

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 JEE applications. There are three types of enterprise beans: entity beans, session beans, and message-driven beans.

Enterprise Information System (EIS)
The applications that comprise an enterprise's existing system for handling company-wide information. An enterprise information system offers a well-defined set of services that are exposed as local or remote interfaces or both.

Enterprise JavaBeans (EJB)
A component architecture defined by Oracle for the development and deployment of object-oriented, distributed, enterprise-level applications (JEE).

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*.

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 call interface (ECI)
A facility that allows a non CICS program to run a CICS program. Data is exchanged in a COMMAREA or a channel as for usual CICS interprogram communication.

external communications interface (EXCI)
An MVS™ application programming interface provided by CICS Transaction Server for IBM z/OS that enables a non-CICS program to call a CICS program and to pass and receive data using a COMMAREA. The CICS application program is started 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.

External Security Manager (ESM)
A security manager that operates outside CICS. For example, RACF can be used as an external security manager with CICS Transaction Server.

F

firewall
A configuration of software that prevents unauthorized traffic between a trusted network and an untrusted network.

FMH See *function management header*.

fully qualified APPLID
Used to identify CICS Transaction Gateway connections on the CICS server and tasks in a CICSplex. It is composed of an APPLID with an optional network qualifier. See also *APPLID* and *APPLID qualifier*.

function management header (FMH)
One or more headers, optionally present in the leading request units (RUs) of an RU chain, that allow one LU to (a) select a transaction program or device at the session partner and control the way in which the user data it sends is handled at the destination, (b) change the destination or the characteristics of the data during the session, and (c) transmit between session partners status or user information about the destination (for example, a program or device). Function management headers can be used with LU type 1, 4, and 6.2 protocols.

G

Gateway
A device or program used to connect two systems or networks.

Gateway classes
The Gateway classes provide APIs for ECI, EPI, and ESI that allow communication between Java client applications and the Gateway daemon.

Gateway daemon
A long-running Java process that listens for network requests from remote Client applications. It issues these requests to CICS servers using the CICS connectivity components. The Gateway daemon on IBM z/OS processes ECI requests and on UNIX, Windows, and Linux platforms it process EPI and ESI requests as well. The Gateway daemon uses the GATEWAY section of ctg.ini for its configuration.

Gateway group
A set of Gateway daemons that share an APPLID qualifier, and where each Gateway daemon has a unique APPLID within the Gateway group.

Gateway token
A token that represents a specific Gateway daemon, when a connection is established successfully. Gateway tokens are used in the C language statistics and ECI V2 APIs.

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.

H

HA group
See *highly available Gateway group*.

highly available Gateway group (HA group)
A Gateway group that utilizes TCP/IP load balancing, and can be viewed

as a single logical Gateway daemon. A Gateway daemon instance in a HA group can recover indoubt XA transactions on behalf of another Gateway daemon within the HA group.

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 IP 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 can 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 (HTTP)
In the Internet suite of protocols, the protocol that is used to transfer and display hypertext and XML documents.

Hypertext Transfer Protocol Secure (HTTPS)
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.

I

ID data
An ID data structure holds an individual result from a statistical API function.

identity propagation
The concept of preserving a user's security identity information (the distributed identity) independent of where the identity information has been created, for use during authorization and for auditing purposes. The distributed identity is carried with a request from the distributed client application to the CICS server, and is incorporated in the access control of the server as part of the authorization process, for example, using RACF. CICS Transaction Gateway flows the distributed identity to CICS. See also *distributed identity*.

identity propagation login module
A code component that provides support for identity propagation. The identity propagation login module is included with the CICS Transaction

Gateway ECI resource adapter (cicseci.rar), conforms to the JAAS specification and is contained in a single Java class within the resource adapter. See also *identity propagation*.

iKeyman
A tool for maintaining digital certificates for JSSE.

in doubt
The state of a transaction that has completed the prepare phase of the two-phase commit process and is waiting to be completed.

in flight
The state of a transaction that has not yet completed the prepare phase of the two-phase commit process.

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*.

<install_path>
This term is used in file paths to represent the directory where you installed the product. For more information, see ../installing/topics/cclahlinstfiles.dita.

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, run 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.

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.

IPIC See *IP interconnectivity*.

J

Java An object-oriented programming language for portable interpretive code that supports interaction among remote objects.

Java 2 Platform, Enterprise Edition (J2EE, Java EE)
An environment for developing and deploying enterprise applications, defined by Oracle. The JEE platform consists of a set of services, application programming interfaces (APIs), and protocols that allow multi-tiered, Web-based applications to be developed.

JavaBeans
As defined for Java by Oracle, 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 Oracle provided for the Java platform.

JavaGateway
The URL of the CICS Transaction Gateway with which the Java Client application communicates. The JavaGateway takes the form `protocol://address:port`. These protocols are supported: `tcp://`, `ssl://`, and `local:`. 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).

JavaScript Object Notation (JSON)
A lightweight data-interchange format that is based on the object-literal notation of JavaScript. JSON is programming-language neutral but uses conventions from languages that include C, C++, C#, Java, JavaScript, Perl, Python.

JCA See *JEE Connector Architecture*.

JDK See *Java development kit*.

JEE (formerly J2EE)
See *Java 2 Platform Enterprise Edition*.

JEE Connector architecture (JCA)
A standard architecture for connecting the JEE platform to heterogeneous enterprise information systems (EIS).

JNI See *Java Native Interface*.

JRE See *Java Runtime Environment*.

JSON See *JavaScript Object Notation (JSON)*.

JSON Schema
A JavaScript Object Notation (JSON) document that describes the structure and constrains the contents of other JSON documents.

JSON web service
A web service that accepts and produces JSON payloads.

JSSE See *Java Secure Socket Extension*.

JVM See *Java Virtual Machine*.

K

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.

Keystore
In the JSSE protocol, a file that contains public keys, private keys, trusted roots, and certificates.

L

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 CICS server
An alias that can be passed on an ECI request when running in remote mode to CICS Transaction Gateway. The alias name is mapped to an actual CICS server name by a dynamic server selection (DSS) mechanism.

logical end of day
The local time of day on the 24-hour clock to which a Gateway daemon aligns statistics intervals. If the statistics interval is 24 hours, this is the local time at which interval statistics will be reset and, on IBM z/OS, optionally recorded to SMF. This time is set using the **stateod** parameter in the configuration file (ctg.ini).

logical unit (LU)
In SNA, a port through which a user accesses the SNA network to communicate with another user and through which the 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 might be capable of supporting many sessions with other logical units. See also *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)
The processing that a program performs between synchronization points.

LU See *logical unit*.

LU 6.2 See *logical unit 6.2*.

LU-LU session
In SNA, a session between two logical units (LUs) in an SNA network. It provides communication between two users, or between a 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*.

M

managed mode
Describes an environment in which connections are obtained from connection factories that the JEE server has set up. Such connections are owned by the JEE server.

media 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.

N

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 IP addresses.

NAU See *network addressable unit*.

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 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.

nonextended logical unit of work
See *SYNCONRETURN*.

nonmanaged mode
An environment in which the application is responsible for generating and configuring connection factories. The JEE server does not own or know about these connection factories and therefore provides no Quality of Service facilities.

O

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.

OO
See *object-oriented*.

OSGi
A specification that describes a modular system and a service platform for the Java programming language that implements a complete and dynamic component model.

P

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.

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.

password phrase
A character string, between 9 and 100 characters in length, that is used for authentication when a user signs on to CICS. Because a password phrase can provide an exponentially greater number of possible combinations of characters than a standard 8 character password, the use of password

phrases can enhance system security. Password phrases are verified by the External Security Manager (ESM), and can contain alphanumeric characters, and any of the other non alphanumeric characters that are supported by the ESM. See also *External Security Manager (ESM)*.

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.

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.

port sharing
A way of load balancing TCP/IP connections across a group of servers running in the same IBM z/OS image.

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*.

<product_data_path>
This term represents the directory used by the Windows CICS Transaction Gateway for common application data. For more information, see ../installing/topics/cclahlinstfiles.dita.

protocol boundary
The signals and rules governing interactions between two components within a node.

Q

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.

R

RACF See *Resource Access Control Facility*.

realm A named collection of users and groups that can be used in a specific security context. See also *distinguished name* and *identity propagation*.

Recoverable resource management services (RRMS)
The registration services, context services, and resource recovery services provided by the IBM z/OS sync point manager that enable consistent changes to be made to multiple protected resources.

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 monitoring exits
Exits that provide information about individual requests as they are processed by the CICS Transaction Gateway.

request unit (RU)
In SNA, a message unit that contains control information such as a request code, or function management (FM) headers, 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 also *CID methodology*.

response unit (RU)
A message unit that acknowledges a request unit; it can contain prefix information received in a request unit.

Resource adapter
A system-level software driver that is used by an EJB container or an application client to connect to an enterprise information system (EIS). A resource adapter plugs in to a container; the application components deployed on the container then use the client API (exposed by adapter) or tool-generated, high-level abstractions to access the underlying EIS.

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)
An IBM z/OS facility that provides two-phase sync point support across participating resource managers.

RESTful
Pertaining to applications and services that conform to Representational State Transfer (REST) constraints.

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.

RPC See *remote procedure call*.

RRMS
: See *Recoverable resource management services*.

RRS See *Resource Recovery Services*.

RU See *Request unit* and *Response unit*.

S

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.

server name remapping
: See *dynamic server selection*.

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.

sign-on capable terminal
: A sign-on capable terminal allows sign-on transactions that are either supplied with CICS (CESN) or written by the user, to be run. Contrast with sign-on incapable terminal.

silent installation
: Installation that does not display messages or windows during its progress. Silent installation is not a synonym of "unattended installation", although it is often improperly used as such.

single-byte character set (SBCS)
: A character set in which each character is represented by 1 byte. Contrast with double-byte character set.

SIT See *system initialization table*.

SLU See *secondary logical unit*.

SMF The IBM z/OS System Management Facility (SMF) collects and records system and job-related information that your IBM z/OS installation can use for reporting, billing, analysis, profiling, and maintaining system security. CICS TG for IBM z/OS writes statistical data to SMF.

SMIT See *System Management Interface Tool*.

SNA See *Systems Network Architecture*.

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*.

SSLight
An implementation of SSL, written in Java, and no longer supported by CICS Transaction Gateway.

standard error
In many workstation-based operating systems, the output stream to which error messages or diagnostic messages are sent.

statistic data
A statistic data structure holds individual statistical result returned after calling a statistical API function.

statistic group
A generic term for a collection of statistic IDs.

statistic ID
A label referring to a specific statistic. A statistic ID is used to retrieve specific statistical data, and always has a direct relationship with a statistic group.

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
Synchronization point. During transaction processing, a reference point to which protected resources can be restored if a failure occurs.

SYNCONRETURN
A request where the CICS server takes a sync point on successful completion of the server program. Changes to recoverable resources made by the server program are committed or rolled-back independently of changes to recoverable resources made by the client program issuing the

ECI request, or changes made by the server in any subsequent ECI request. Also referred to as a *nonextended logical unit of work*.

system initialization table (SIT)
A table containing parameters used to start a CICS control region.

System Management Command
An administrative request received by a Gateway daemon (or Gateway daemon address space on IBM z/OS) from the **ctgadmin** command (on UNIX, Linux, or Windows) or the IBM z/OS console. The request might be made to retrieve information about the Gateway daemon, or to alter some aspect of Gateway daemon behavior. Typically, a **ctgadmin** command in the form **ctgadmin** <command string> is entered by an operator using the command line interface, or a modify command in the form `/F <job name>,APPL=<command string>` is entered by an operator on the IBM z/OS console.

System Management Interface Tool (SMIT)
An interface tool of the IBM AIX operating system for installing, maintaining, configuring, and diagnosing tasks.

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 IBM z/OS.

T

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 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*.

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 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.

U

unattended installation
: Unattended installation is installation performed without user interaction during its progress, or, with no user present at all, except for the initial launch of the process.

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.

UOW See *unit of work*.

UR See *unit of recovery*.

URL See *Uniform Resource Locator*.

user registry
The location where the distinguished name of a user is defined and authenticated. See also *distinguished name*.

user session
Any APPC session other than a SNASVCMG session.

V

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.

W

WAN See *wide area network*.

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 can use or provide public communication facilities.

Wrapping trace
On Windows, UNIX, and Linux, 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**.

WSBind file
A Web service bind file is a resource that describes the specifics of the Web service.

X

XA request
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).

Related literature

Other documentation relating to CICS Transaction Gateway.

IBM Redbooks® titles are available on a wide range of subjects relevant to CICS Transaction Gateway programming, installation, operation and troubleshooting. See the: IBM Redbooks site for more information.

Documentation for many IBM products is available online from the IBM Publications Center.

© Copyright IBM Corp. 1998, 2016

Accessibility

Accessibility features help users with a physical disability, for example restricted mobility or limited vision, to use information technology products successfully. CICS Transaction Gateway is compatible with the JAWS screen reader. CICS Transaction Gateway provides accessibility by enabling keyboard-only operation.

For more information about the IBM commitment to accessibility, visit the IBM Accessibility Center.

Installation

The InstallAnywhere wizard is not fully accessible to screen readers.

To use the installer with a screen reader you must use console mode installation from a command prompt, specifying the -i console option.

Console mode displays text over multiple screens, and enables you to make choices during the installation process. The command prompt interface does not provide a cursor for navigating over the displayed text. When you use the JAWS screen reader you can repeat the displayed text with the command used for reading the current window (key combination Insert+B).

The first screen displayed by the installer is for language selection; the default language depends on the values in the system regional settings. To bypass the language selection screen, use the -l *lang* command option; where *lang* is one of the following:

- `de` German
- `en` English
- `es` Spanish
- `fr` French
- `it` Italian
- `ja` Japanese
- `ko` Korean
- `tr` Turkish
- `zh_CN` Chinese

For example, to install with the console interface in French:

```
installer -i console -l fr
```

Configuration Tool accessibility

The configuration file uses the number sign (#) character to denote a comment; consider configuring your screen reader accordingly.

Starting the Gateway daemon

You can start the Gateway daemon from a command prompt using a screen reader.

© Copyright IBM Corp. 1998, 2016

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'.
```

When using a screen reader on Windows, the Gateway daemon should be started and stopped with Windows services by starting and stopping the IBM CICS Transaction Gateway service. To determine if the CICS Transaction Gateway has started or stopped use the Windows Event Log viewer to check the messages in the Application log.

cicsterm

Although cicsterm is accessible, it relies on the application that is being processed to define an accessible 3270 screen.

Keyboard mapping depends on the terminal type that you are using, for more information, see *Keyboard mapping for cicsterm*.

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.

Uppercase
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.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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 grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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 jurisdictions 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 publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those

© Copyright IBM Corp. 1998, 2016

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

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 Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

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 document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data discussed herein is presented as derived under specific operating conditions. Actual results may vary.

The client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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-IBMproducts. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the

names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

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 the application programming interface 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

© (your company name) (year).
Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. _enter the year or years_.

Programming interface information

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Terms and conditions for product documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

Safety and environmental notices

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Readers' Comments — We'd Like to Hear from You

CICS Transaction Gateway for Multiplatforms
Version 9 Release 2
Developing Applications

Publication No. SC34-7339-00

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your 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. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +44 1962 816151
- Send your comments via email to: idrcf@uk.ibm.com

If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	Email address

Readers' Comments — We'd Like to Hear from You
SC34-7339-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM United Kingdom Limited
User Technologies Department (MP189)
Hursley Park
Winchester
Hampshire
United Kingdom
SO21 2JN

Fold and Tape | **Please do not staple** | Fold and Tape

SC34-7339-00

Cut or Fold
Along Line

SC34-7339-00