CICS® Universal Clients

C++ Programming

Version 3.1

SC33-1923-01

CICS® Universal Clients

C++ Programming

Version 3.1

SC33-1923-01

Note!
Before using this information and the product it supports, read the general information under “Notices” on page 109.

Second Edition (September 1999)

This edition applies to CICS Universal Clients, Version 3.1, program number 5648-B42, and to all subsequent versions, releases, and modifications until otherwise indicated in new editions.

The previous edition of this book, *CICS Family: OO Programming in C++ for CICS Clients*, SC33-1923-00, is still available and should be used for earlier versions of CICS Clients.

Contents

About this book vii
Who should read this book vii
Conventions and terminology used in this book vii
Prerequisite and related information . . . viii
How to send your comments viii
Obtaining books from IBM ix

Part 1. Client Classes—Guidance. . 1

Chapter 1. Introduction to OO programming 3
OO support in CICS Clients 3
Programming language support. 4

Chapter 2. Establishing the working environment 5
Environments supported 5

Chapter 3. Using the CICS client C++ classes. 7
Compiling and Linking 7
Using IBM VisualAge C++ OS/2 7
Using Microsoft Visual C++ on Windows NT and Windows 98 7
Using IBM VisualAge C++ for Windows . . 7
Using IBM Compilers on AIX 8
Using Sun Workshop 8
Multi-Threading 8
Handling Exceptions 9
Async Exception Handling 9
External call interface 10
Using Commareas 11
Finding potential servers. 11
Server connection 12
Passing data to a server program 12
Controlling server interactions 13
Monitoring server availability 17
Managing logical units of work 18
Security Management for ECI 19
C++ External presentation interface 20
Support for Automatic Transaction Initiation (ATI) 22
Starting a 3270 terminal connection to CICS 23
Accessing fields on CICS 3270 screens . . 24
EPI call synchronization types 25
EPI BMS conversion utility 28
Using EPI BMS Map Classes 30
Security Management for EPI 31

Part 2. CICS Client C++ classes - reference 33

Chapter 4. Ccl class 37
Enumerations 37
Bool. 37
Sync 37
ExCode 37

Chapter 5. CclBuf class 39
CclBuf constructors 40
CclBuf (1). 40
CclBuf (2). 40
CclBuf (3). 40
CclBuf (4). 40
Public methods 41
assign 41
cut 41
dataArea 41
dataAreaLength. 41
dataAreaOwner. 41
dataAreaType 41
dataLength 42
insert 42
listState 42
operator= (1). 42
operator= (2). 42
operator+= (1) 43
operator+= (2) 43
operator== 43
operator!=. 43
replace. 43
setDataLength 44
Enumerations 44
DataAreaOwner. 44
DataAreaType 44

Chapter 6. CclConn class 45

CclConn constructor 45
Public methods 46
alterSecurity 46
cancel 46
changed 46
changePassword 46
link 47
listState 47
makeSecurityDefault 48
password (1) 48
password (2) 48
serverName (1) 48
serverName (2) 48
status 48
serverStatus 49
serverStatusText. 49
userId (1) 49
userId (2) 49
verifyPassword 49
Enumerations 50
ServerStatus 50

Chapter 7. CclECI class 51
CclECI constructor (protected) 51
Public methods 51
exCode. 51
exCodeText 51
handleException 52
instance 52
listState 52
serverCount 52
serverDesc 52
serverName 52

Chapter 8. CclEPI class 55
CclEPI constructor 55
Public methods 55
diagnose 55
exCode. 55
exCodeText 56
handleException 56
serverCount 56
serverDesc 56
serverName 56
state 57
terminate 57
Enumerations 57
State 57

Chapter 9. CclException class 59
Public methods 59
abendCode 59
className 59
diagnose 59
exCode. 59
exCodeText 60
exObject 60
methodName 60

Chapter 10. CclField class 61
Public methods 61
appendText (1) 61
appendText (2) 61
backgroundColor 61
baseAttribute 61
column. 62
dataTag 62
foregroundColor 62
highlight 62
inputProt 62
inputType. 62
intensity 63
length 63
position 63
resetDataTag 63
row 63
setBaseAttribute. 63
setExtAttribute 63
setText (1). 64
setText (2). 64
text 64
textLength 64
transparency 64
Enumerations 65
BaseInts 65
BaseMDT 65
BaseProt 65
BaseType 65
Color 65
Highlight 65
Transparency. 65

Chapter 11. CclFlow class. 67
CclFlow constructor 67
CclFlow (1) 67
CclFlow (2) 67
Public methods 68
abendCode 68
callType 68
callTypeText 68

connection 68
diagnose 68
flowId 68
forceReset 68
handleReply 69
listState 69
poll 69
setTimeout 69
syncType 70
timeout 70
uow 70
wait 70
Enumerations 70
CallType 70

Chapter 12. CclMap class 71
CclMap constructor 71
Public methods 71
exCode 71
exCodeText 71
field (1) 72
field (2) 72
Protected methods 72
namedField 72
validate 72

Chapter 13. CclScreen class 75
Public methods 75
cursorCol 75
cursorRow 75
depth 75
field (1) 75
field (2) 75
fieldCount 76
mapName 76
mapSetName 76
setAID 76
setCursor 76
width 77
Enumerations 77
AID 77

Chapter 14. CclSecAttr 79
Public Methods 79
expiryTime 79
invalidCount 79
lastAccessTime 79
lastVerifiedTime 79

Chapter 15. CclSecTime 81
Public Methods 81
day 81
get_time_t 81
get_tm 81
hours 81
hundredths 81
minutes 81
month 82
seconds 82
year 82

Chapter 16. CclSession class 83
CclSession constructor 83
Public methods 83
diagnose 83
handleReply 83
state 84
terminal 84
transID 84
Enumerations 84
State 84

Chapter 17. CclTerminal class 85
CclTerminal constructor 85
CclTerminal 85
Public methods 86
alterSecurity 86
changePassword 86
CCSid 87
diagnose 87
disconnect 87
discReason 87
exCode 87
exCodeText 87
install 88
makeSecurityDefault 88
netName 88
password 88
poll 88
queryATI 89
readTimeout 89
receiveATI 89
screen 89
send (1) 89
send (2) 90
setATI 90
signonCapability 90
state 91
serverName 91
termID 91

transID. 91
userId 91
verifyPassword 91
Enumerations 91
ATIState 91
signonType 92
State 92
EndTerminalReason 92

Chapter 18. CclUOW class 93
CclUOW constructor 93
Public methods 93
backout 93
commit 93
forceReset. 94
listState 94
uowId 94

Part 3. Appendixes 95

Appendix. Exception Objects 97

Glossary 103

Bibliography 105
C++ Programming 105
The CICS Transaction Gateway and CICS Universal Clients library 105
CICS Transaction Gateway books . . . 105
CICS Universal Clients books. 106
CICS Family publications 106
Book filenames. 106
Sample configuration documents. . . . 107
Other publications 107
Viewing the online documentation . . . 107

Notices 109
Trademarks and service marks 110

Index. 111

About this book

This book describes object-oriented programming for the CICS external call interface (ECI) and the CICS external presentation interface (EPI). It provides guidance on writing programs, with examples, using the classes and methods provided with IBM CICS Universal Clients version 3.1.

Who should read this book

This book is for CICS application programmers who want to know how to use the object oriented (OO) classes provided in IBM CICS Universal Clients version 3.1 to develop object oriented CICS client programs.

Object oriented programs should be more readily understood because they are written at a higher level, are easier to maintain, and should lend themselves to reuse. Not only does the code included in the class libraries take some of the more error-prone and repetitive elements of programming away from the programmer, but the classes themselves are written to make full and appropriate use of CICS facilities.

CICS services are available to clients through the External Call Interface (ECI) and External Presentation Interface (EPI). The CICS client classes allow a C++ programmer to access the ECI and EPI interfaces in an object oriented manner.

C++ is the programming language supported in this version.

The previous edition of this book, *CICS Family: OO Programming in C++ for CICS Clients*, SC33-1923-00, is still available and should be used for earlier versions of CICS Clients.

Conventions and terminology used in this book

Here are some of the conventions used in this book:

Filenames are shown in a monospaced font — `cuc.ini`

OO classes are shown in bold — **CclBuffer**

OO methods are shown in bold — **fieldCount**

Parameters are shown in italic — *serverName*

Prerequisite and related information

This document assumes that you are familiar with OO concepts and the C++ language, and have a reasonable understanding of the existing services that CICS provides.

For more information about using the CICS services, see *CICS Family: Client/Server Programming*.

How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book, or any other CICS documentation:

- Visit our Web site at:

  `http://www.ibm.com/ts/cics/`

  and follow the **library** link to our feedback form.

  Here you will find the feedback page where you can enter and submit your comments.
- Send your comments by e-mail to idrcf@hursley.ibm.com
- Fax your comments to:

  +44-1962-870229 (if you are outside the UK)
  01962-870229 (if you are in the UK)
- Mail your comments to:

  Information Development
  Mail Point 095
  IBM United Kingdom Laboratories
  Hursley Park
  Winchester
  Hampshire
  SO21 2JN
  United Kingdom

Whichever method you use, ensure that you include:

- The name of the book
- The form number of the book
- If applicable, the version of the product
- The specific location of the text you are commenting on, for example, a page number or table number.

Obtaining books from IBM

You can order publications through your IBM representative or the IBM branch office serving your locality.

You can check the availability of books from our Web site at:
`http://www.ibm.com/ts/cics/`

Follow the **library** link and download books as required.

Also, you can order books from the Web site at:
`http://www.elink.ibmlink.ibm/pbl/pbl`

Part 1. Client Classes—Guidance

Chapter 1. Introduction to OO programming 3
OO support in CICS Clients 3
Programming language support. 4

Chapter 2. Establishing the working environment 5
Environments supported 5

Chapter 3. Using the CICS client C++ classes. 7
Compiling and Linking 7
Using IBM VisualAge C++ OS/2 7
Using Microsoft Visual C++ on Windows NT and Windows 98 7
Using IBM VisualAge C++ for Windows . . 7
Using IBM Compilers on AIX 8
Using Sun Workshop 8
Multi-Threading 8
Handling Exceptions 9
Async Exception Handling 9
External call interface 10
Using Commareas 11
Finding potential servers. 11
Server connection 12
Passing data to a server program 12
Controlling server interactions 13
Synchronous reply handling 14
Asynchronous reply handling 15
Deferred synchronous reply handling 16
Monitoring server availability 17
Managing logical units of work 18
Security Management for ECI 19
C++ External presentation interface 20
Support for Automatic Transaction Initiation (ATI) 22
Starting a 3270 terminal connection to CICS 23
Accessing fields on CICS 3270 screens . . 24
EPI call synchronization types 25
EPI BMS conversion utility 28
Mapset containing a single map . . . 29
Using EPI BMS Map Classes 30
Security Management for EPI 31

Chapter 1. Introduction to OO programming

The CICS® family provides robust transaction processing capabilities across the major hardware platforms that IBM® offers, and also across key non-IBM platforms such as UNIX. It offers a wide range of features for supporting client/server applications, and allows the use of modern graphical interfaces for presenting information to the end user. The CICS family now supports the emerging technology for object oriented programming and offers CICS users a way of capitalizing on many of the benefits of object technology while making use of their investment in CICS skills, data and applications.

Object oriented programming allows more realistic models to be built in flexible programming languages. You can define new types or classes of objects, as well as employing a variety of structures to represent these objects.

Object oriented programming also allows you to associate more meaning with data by creating methods (member functions) that define the behavior associated with objects of a certain type, thereby capturing more of the semantics associated with the underlying data.

The hiding or encapsulating of much of the complexity of a piece of software inside a simpler external shell provides the key to reuse of code. An object defined in such a way can be used from a wide range of different applications. The provision of discrete, well defined objects can be the foundation of a library of reusable parts from which future applications can be built more quickly and cheaply. The reuse of existing parts leads to better levels of software quality as they have already been tested and used in other applications.

OO support in CICS Clients

The principal communication mechanism provided on the CICS Client is the External Call Interface (ECI). The provision of an ECI class library, modelling the full function of the ECI in an object oriented way, provides the base upon which extended support has been built.

Supplied classes offer the capability of making links to servers, making calls to the CICS programs on the server, finding out status, and making use of units of work (UOW's).

The second interface available to application programmers on the CICS Client is the External Presentation Interface (EPI). Communication with 3270

terminal based CICS applications is provided by classes encapsulating terminals, screens, fields and BMS maps. The EPI classes make it unnecessary to work directly with the 3270 datastream and make it easier to examine and update the contents of an output screen.

Programming language support

OO libraries are provided with CICS Clients Version 3.1 for C++ and BASIC programmers. This book covers the C++ programming topics. If you wish to use the COM libraries, please refer to *OO Programming in C++ for CICS Clients*.

Chapter 2. Establishing the working environment

You are provided with C++ (OO) support for CICS clients in OS/2®, AIX, Solaris and Windows environments. This includes the class library, C++ header files, the BMS map utility, and sample code.

Environments supported

OS/2

IBM OS/2 Version 4 or later
IBM Visual Age C++ for OS/2 Version 3.0 or later

Windows NT

Windows NT Workstation Version 4 SP3 or later
Windows NT Server Version 4 SP3 or later
IBM Visual Age C++ for Windows Version 3.5 or later
Microsoft Visual C++ Version 5.0 or later

Windows 98

Windows 98
Microsoft Visual C++ Version 5.0 or later

AIX®

AIX V4.3.1 or later
IBM C and C++ Compilers Version 3.6
IBM Visual Age C++ V4
IBM C Set C++ V3.1

Solaris

Solaris 2.6 or Solaris 7
Sun Workshop V3.0

Refer to *CICS Client Administration*, SC33-1436-00, for details of CICS server platforms supported by the CICS clients.

Chapter 3. Using the CICS client C++ classes

For examples of programs that use the CICS client C++ classes, refer to the separate samples documentation.

Compiling and Linking

Your C++ program source will need `#include` statements to include either CICSECI.HPP (for the ECI classes) or CICSEPI.HPP (for the EPI classes). These files are in the `include` subdirectory.

Using IBM VisualAge C++ OS/2

- The preprocessor macro CICS_OS2 must be defined to the compiler using the `/DCICS_OS2` option.
- The `/Gm+` compiler option must be specified so that multi-threaded versions of C++ run-time libraries are linked.
- The application must be linked with the CCLCPOS2.LIB library supplied in the subdirectory `LIB`.

Using Microsoft Visual C++ on Windows NT and Windows 98

Compiling and linking on Windows NT and Windows 98 using Microsoft Visual C++.

- The preprocessor definition CICS_W32 must be defined to the compiler using the `/DCICS_W32` option.
- Multithreaded DLL run-time libraries must be selected in the C++ compiler code generation options (equivalent to the `/MD` compiler option).
- The application must be linked with the CCLCPW32.LIB library supplied in the subdirectory `LIB`.

Using IBM VisualAge C++ for Windows

Compiling and linking on Windows NT using IBM VisualAge® C++ for Windows.

- The preprocessor macro CICS_W32 must be defined to the compiler (equivalent to the `/DCICS_W32` compiler option)
- Multithread libraries must be selected in the compiler object code options (equivalent to the `/Gm` compiler option)
- The application must be linked with the CCLICW32.LIB library supplied in subdirectory `LIB`

Using the CICS client C++ classes

Using IBM Compilers on AIX

Compiling and linking on AIX using IBM compilers

- Define the pre-processor macro `CICS_AIX`. You can do this in your code before including the header file or define it using a command line directive, e.g. `-DCICS_AIX`.
- When compiling specify:

  `xlC_r` to compile with the threadsafe compilers.

  `-lc_r` to use threadsafe C runtime libraries

  `-lcclcp`
  to link the shared library `libcclcp.a`

Using Sun Workshop

Compiling and linking on Solaris using Sun Workshop

- Define the pre-processor macro `CICS_SOL`. You can do this in your code before including the header file or define it using a command line directive, e.g. `-DCICS_SOL`.
-

  `-lc` to use C runtime libraries

  `-lcclcp`
  to link the shared library `libcclcp.so`
- Specify `-lcclcp` when compiling. This will link the shared library `libcclcp.so`.

Multi-Threading

The CICS Client C++ libraries are not totally 'thread safe', i.e. they do not have critical sections or semaphores to prevent two threads interfering if they both update the same instance of an object. However, the classes do not share data, so they are useable in a 'well behaved' multi-threaded program. The normal technique is for each thread to have its own instance of CclConn, CclFlow, CclBuf etc (they are light-weight objects).

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. Doing a throw within a destructor is a dangerous thing to do.

This routine is suitable for most needs when using synchronisation modes of dsync and sync. An example of this could be as follows :-

```
#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 synchronisation mode. With async mode a separate

thread controlled by the class library dll is created and an exception can occur on that thread. If an exception does occur on that thread, the default exception handler would throw the exception but there is no code in the class library to trap the throw. For unhandled exceptions, the default action of most compilers' runtimes is to terminate the application.

To create a new exception handler you do the following

```
class MyCclECI : public CclECI {
public:
   void handleException( CclException ex) {
         // Place whatever code you want here, for example set a
         // semaphore, or generate a Window Message
         }
   };
```

Once you have subclassed the ECI Class, you still can only create one object of this class for your application, however do not use the instance method, you must create the object either explicitly e.g.

```
MyCclECI myeci;
```

or by using the new operator

```
MyCclECI *pmyeci;
pmyeci = new MyCclECI;
```

External call interface

The ECI is one of two interfaces through which a non-CICS client program 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 client programming with the ECI. The following classes are included:

Table 1. 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

Table 1. C++ ECI classes. (continued)

Object	Classname	Description
UOW	CclUOW	Corresponds with a Unit of Work in the server—used for managing updates to recoverable resources.

Using Commareas

A CommArea is a block of storage allocated by the program. The client program uses the commarea to send data to the server and the server uses the same storage to return data to the client. Therefore, you must create a CommArea that is large enough to contain all the information to be sent to the server and large enough to contain all the information that can be returned from the server.

For example, you need to send a 12 byte serial number to the server, but you may receive 20 Kb back from the server. You must create a CommArea of size 20 Kb. Your code would look like this:
In the example, the serial number is stored in the new `commarea` which is then

```
 // 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
```

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 client software strips off the excess nulls and transmits 12 bytes to the server.

Finding potential servers

Information about the CICS servers that can be used by a client program is defined in the CICS Client Initialization file, CUC.INI (Client) or CTG.INI (Gateway). See *CICS Clients Administration Manual* for more information. The existence of such a definition doesn't guarantee availability of a server.

The ECI object—**CclECI** provides access to this server information through its **serverCount**, **serverDesc**, and **serverName** methods.

Unless the ECI class has been subclassed, its unique instance is found using the class method **instance** as in the following example:

```
CclECI* pECI = CclECI::instance();
printf( "Server Count = %d\n", pECI-> serverCount() );
printf( "Server1 Name = %s\n", pECI-> serverName( 1 ) );
  ...
```

Typical output produced:

```
Server Count = 2
Server1 Name = DEVTSERV
```

Server connection

A client program requires one connection object—**CclConn** for each CICS server with which it will interact. When a connection object is created, optional data can be specified which includes:

- The name of the server to be connected. This must be one of the server names defined in the CICS Client Initialization file. If this name is omitted, the default server, as defined by the ECI, will be used.
- A user ID. Some servers may require that a client 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 program 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 marshalling.

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 second example, an existing memory structure is used. This could, for example, correspond to a COBOL record used in the server program. In this case, the buffer object knows the record is fixed-length, externally-defined, and ensures it can not be extended in any subsequent processing. The link call requests execution of the program QVALUE on the CICS server defined by the serv2 connection object and passes data via the structure on which the buffer object comma2 is overlaid.

```
struct rec{
         short key;
         char name[8];
         char retval[70];
         };
rec record1 = { 1234,"Hilary" };
CclBuf comma2( sizeof(rec),&record1 );
serv2.link( sflow,"QVALUE",&comma2 );
  ...
```

The communications area returned from a server is also contained in a buffer object.

Controlling server interactions

A flow object—**CclFlow**—controls each interaction between the client program and a server and determines the synchronization of reply processing; synchronous, deferred synchronous or asynchronous. This example creates a synchronous flow object:
A flow object is referenced when a server request call is first issued and

```
CclFlow sflow( Ccl::sync );
```

remains active from that time until all client processing of the corresponding reply from the server has been completed. At that point it is set inactive and becomes available for reuse or deletion. During its active lifespan, a flow object maintains the state of the client/server interaction it is controlling.

The flow class should be subclassed to provide the implementation of a reply handler which will be called when a reply is received; this happens regardless of the synchronization type. The reply handler is passed a buffer object which contains the communication area returned by the server. A default reply handler is provided; it just returns to the caller without doing anything.

Separate flow subclasses could be needed to cater for different client/server communication area protocols. Many flows may be active at the same time. Many servers may be used simultaneously by the same client.

Synchronous reply handling

In the synchronous model, the client remains blocked at the server request call until a reply is eventually received from the server.

The following code 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 );
```

The program 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 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 program 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 the program ECICPO1 is called as follows:

```
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.

Asynchronous reply handling

In the asynchronous model, the client program 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 implementation of asynchronous reply handling uses multi-threading and is not supported on Windows 3.1.

The code which follows 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;
    }
  };
```

The program constructs a subclassed ECI object; then a connection object using the supplied server name, password and user ID. It constructs a buffer object 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 );
  ...
```

In the sample, there is nothing else for the main program 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 the program ECICPO2 is called as follows:

```
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 program 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();
```

Deferred synchronous reply handling

In the deferred synchronous model, the client program 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 program 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 program is now made to loop with a delay until poll indicates the reply has been received from the server. Note that in the deferred synchronous model, a buffer object to hold the returned communication area can be supplied as a parameter to the **poll** method. If, as in the example below, no buffer object is supplied on the **poll** method, one is allocated internally within the flow object, and is deleted after the reply handler has run.

```
  ...
Ccl::Bool reply = Ccl::no;
while ( reply == Ccl::no ) {
  cout << "DSync polling..." << endl;
  reply = dsflow.poll();
  if ( reply == Ccl::no ) DosSleep( msecs );
  }
  ...
```

Typical output on successful completion would look like this:

```
DSync polling...
DSync polling...
DSync polling...
Reply from CICS server: Hello World
```

As in the asynchronous model, the **wait** method can be used to make a deferred synchronous flow synchronous, blocking the client.

Monitoring server availability

The connection object—**CclConn** has 3 methods which can be used to determine the availability of the server connection that it represents:

status requests the status (that is, the availability) of the server.

changed
: requests notification of any change in this status.

cancel
: requests cancellation of a **changed** request.

The example described below shows how server availability can be monitored in a client program that is busy doing something else.

Here is a subclass of the flow class designed for use with server status calls. The reply handler implementation prints the server name and its newly-changed status; it ignores the returned communication area. Next, it issues a changed server request so that the next server status change will be

received. The reply handler will be called every time the availability of the server changes.

```
class ChgFlow : public CclFlow {
public:
       ChgFlow( Ccl::Sync stype ) : CclFlow( stype ) {}
  void handleReply( CclBuf* ) {
         CclConn* ccon = connection();
         cout << ccon-> serverName() << " is "
              << ccon-> serverStatusText() << endl;
         ChgFlow* sflow = new ChgFlow( Ccl::async );
         ccon-> changed( *sflow );
         }
  };
```

The main program iterates through all the servers listed in the CICS Client Initialization file. For each one, an asynchronous status request call is issued. The program 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 program is not running. It starts, and after a while makes contact with one of the servers.

Managing logical units of work

A logical unit of work is all the processing in a server that is needed to establish a set of updates to recoverable resources such as files or queues. If the unit of work finishes normally, ALL the changes can then be either committed or backed out. If it finishes abnormally, for example because a program abends, ALL the changes will be backed out.

A logical unit of work managed by a client can include many server link requests and many active units of work can be managed by a client at the same time, but some restrictions are imposed by the ECI. A given logical unit of work can include links to only one server. Only one link can be active at a time in a logical unit of work so care must be taken with non-synchronous requests.

A client program 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:
Any server link request which participates in a unit of work references the
`CclUOW uow;`

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 for a non zero value.

Security Management for ECI

You are now able to do security management on Servers that support Password Expiry Management.Please refer to the *CICS Family*: *Client/Server Programming* for more information on supported servers and protocols.

To use these features you first must have constructed a Connection object. The two methods available are **verifyPassword** which checks the userid and password within the connection object with the Server Security System, and **changePassword** which allows you to change the password at the server. If successful the connection object password is updated accordingly.

If either call is successful, you are returned a pointer to an internal object which provides information about the security, a CclSecAttr object. This object provides access to information such as last verified Date and Time, Expiry Date and Time and Last access Date and Time. If you query for example last verified Date, you get back a pointer to an object which allows you to get the information in various formats. The following is a sample of code to show the use of these various objects.

```
// Connection object already created called conn
CclSecAttr *pAttrblock;             // pointer to security attributes
CclSecTime *pDTinfo;                // pointer to Date/Time information
try {
     pAttrblock = conn->verifyPassword();
     pDTinfo = pAttrblock->lastVerifiedTime();
     cout << "last verified year  :" <<pDTinfo->year() << endl;
     cout << "last verified month :" <<pDTinfo->month() << endl;
     cout << "last verified day   :" <<pDTinfo->day() << endl;
     cout << "last verified hours :" <<pDTinfo->hours() << endl;
     cout << "last verified mins  :" <<pDTinfo->minutes() << endl;
     cout << "last verified secs  :" <<pDTinfo->seconds() << endl;
     cout << "last verified 100ths:" <<pDTinfo->hundredths() << endl;
// Use a tm structure to produce a single line text of information
     tm mytime;
     mytime = pDTinfo->get_tm();
     cout << "full info:" << asctime(&mytime) << endl;
}
catch (CclException &ex)
{

// Could check for expired password error and handle if required
  cout << "Exception occurred: " <<ex.diagnose()<< endl;
}
```

Note that the security attributes and date/time memory are all handled by the connection object. If you destroy the connection object, you destroy the security information being held by that object.

C++ External presentation interface

Many existing CICS server applications are written for 3270 terminal interfaces and CICS has some powerful capabilities for dealing with these datastreams, including Basic Mapping Support (BMS). It is useful for clients to be able to interface with these server programs.

In procedural programming, the External Presentation Interface (EPI) provides a mechanism for clients to communicate with transactions on a server and to handle 3270 datastreams.

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 datastreams

The classes also enhance the procedural CICS EPI support by providing higher level constructs for handling 3270 datastreams:

1. General purpose C++ classes for handling 3270 datastream, 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 support DBCS fields in 3270 datastreams.

The BMS utility is a tool for statically producing C++ class source code definitions and implementations from a CICS BMS mapset.

Here is a brief description of the supplied classes:

Table 2. 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 client.
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**.
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.

Table 2. C++ EPI classes. (continued)

Object	Classname	Description
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.

Support for Automatic Transaction Initiation (ATI)

The CICS server API call EXEC CICS START allows a server program to start a transaction on a particular terminal. This mechanism, called Automatic Transaction Initiation (ATI), requires additional programming at the client side to handle the interaction between these transactions and normal client-initiated transactions.

Firstly, 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 CICS client queues ATIs for a terminal while a transaction is in progress. The CclTerminal class will perform one of 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 synchronisation type is Synchronous, Asynchronous or Deferred synchronous.

Synchronous When you call the CclTerminal send() method,any outstanding ATIs will be run after the client-initiated transaction has completed. The CclTerminal class will wait for the ATI replies then update the CclScreen contents as part of the synchronous send() call. If you expect an ATI to occur before or between client-initiated transactions, you can call the CclTerminal receiveATI() method to wait synchronously for the ATI.

Asynchronous When the client application calls the **CclTerminal** send() method for an async session, the **CclTerminal** class starts a separate thread to handle replies. If ATIs are disabled, this thread finishes when the CICS transaction is complete. If ATIs

are enabled, the reply thread continues to run between transactions. When the CclTerminal state becomes idle, any outstanding ATIs are run and ATIs received subsequently are run immediately. The reply thread is not started until the first CclTerminal::send() call, so if you expect ATIs to occur before any client-initiated transactions, you can call the receiveATI() method to start the reply thread.

Deferred synchronous

After the **CclTerminal** send() method is called for a dsync session, the poll() method is used to receive the replies. Outstanding ATIs are started when the last reply has been received (i.e. on the final poll() call). You can also call the poll() method to start and receive replies for ATIs between client-initiated transactions. As the poll() method can be called before or between client-initiated transactions, the receiveATI() method is not needed (and is invalid) for deferred synchronous sessions. For any of the synchronisation 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.

Starting a 3270 terminal connection to CICS

The CICS client 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 client initialization file, CUC.INI (Client) or CTG.INI (Gateway). The following C++ sample shows the use of the CclEPI object:

```
#include <cicsepi.hpp> // CICS client EPI headers
  ...
CclEPI epi; // Initialize CICS client EPI
// List all CICS servers in Client initialization file
for ( int i=1; i<= EPI.serverCount(); i++ )
    cout << EPI.serverName(i) << "  "
         << EPI.serverDesc(i) << endl;
```

To establish a 3270 terminal connection to CICS, a CclTerminal object is created. The CICS server name used must be configured in the client's Client 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.

Accessing fields on CICS 3270 screens

Once a terminal connection to CICS has been established, the CclTerminal, CclSession, CclScreen and CclField objects are used to navigate through the screens presented by the CICS server application, reading and updating screen data as required.

The CclScreen object is created by the CclTerminal object and is obtained via the **screen** method on the CclTerminal object. It provides methods for obtaining general information about the 3270 screen (e.g. cursor position) and for accessing individual fields (by row/column screen position or by index). The following example prints out field contents, then ends the CESN transaction (started above) by returning **PF3**:

```
// Get access to the CclScreen object
CclScreen* screen = terminal.screen();
for ( int i=1; i ≤ screen->fieldCount(); i++ ) {
    CclField* field = screen->field(i); // get field by index
    if ( field->textLength > 0 )
        cout << "Field " << i << ": " << field->text();
}
// Return PF3 to CICS
screen->setAID( CclScreen::PF3 );
terminal.send( &session );
// Disconnect the terminal from CICS
terminal.disconnect();
```

The **CclField** class provides access to the text and attributes of an individual 3270 field. These can be used in a variety of ways to locate and manipulate information on a 3270 screen:

```
for ( int i=1; i ≤ screen->fieldCount(); i++ ) {
    CclField* field = screen->field(i); // get field by index
    // Find unprotected (i.e. input) fields
    if ( field->inputProt() == CclField::unprotect )
        ...
    // Find fields containing a specific text string
    if ( strstr(field->text(), "CICS Sign-on") )
        ...
    // Find red fields
    if ( field->foregroundColor() == CclField::red )
        ...
}
```

Note that the string “Sign-on” in the above sample may need to be changed to meet local conventions. For example, an AIX server may use the string “SIGNON”.

EPI call synchronization types

The CICS client EPI C++ classes support synchronous (“blocking”), and deferred synchronous (“polling”) and asynchronous (“callback”) protocols.

In the example above the CclSession object is created with the synchronization type of **Ccl::sync**. When this CclSession object is passed as the first parameter on a CclTerminal **send** method, a synchronous call is made to CICS. The C++ program 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 datastream 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:
The implementation of the **handleReply** method can process the screen data

```
class MySession : public CclSession {
public:
    MySession(Ccl::Sync protocol) : CclSession( protocol ) {}
    // Override reply handler method
    void handleReply( State state, CclScreen* screen );
};
```

available in the CclScreen object, which will have been updated in line with the 3270 datastream send from CICS:

```
void MySession::handleReply( State state, CclScreen* screen ) {
    // Check the state of the session
    switch( state ) {
    case CclSession::client:
    case CclSession::idle:
        // Output data from the screen
        for ( int i=1; i < screen->fieldCount(); i++ ) {
            cout << "Field " << i << ": " << screen->field->text();
        screen->setAID( CclScreen::PF3 );
        ...
    } // end switch
}
```

The **handleReply** method is called for each transmission received from CICS. Depending on the design of the CICS server program, a CclTerminal **send** call may result in one or more replies. The *state* parameter on the **handleReply** method indicates whether the server has finished sending replies:

CclSession::server
: indicates that the CICS server program is still running and has further data to send. The client application can process the current screen contents immediately, or simply wait for further replies.

CclSession::client
: indicates that the CICS server program is now waiting for a response. The client application should process the screen contents and send a reply.

CclSession::idle
: indicates that the CICS server program has completed. The client program should process the screen contents and either disconnect the terminal, or start a further transaction.

Most client applications will want to wait until the CICS server program has finished sending data (that is, the CclSession/CclTerminal state is client or idle) before processing the screen. However, some long-running server programs may send intermediate results or progress information that can usefully be accessed while the state is still server.

The implementation of the **handleReply** method can read and process data from the CclScreen object, update fields as required, and set the cursor position and AID key in preparation for the return transmission to CICS. The application main program should invoke further methods (**send** or **disconnect**) on the CclTerminal object to drive the server application:

```
try {
    // Connect to CICS server
    CclTerminal terminal( "CICS1234" );
    // Create asynchronous session
    MySession session(Ccl::async);
    // Start CESN transaction on CICS server
    terminal.send( &session, "CESN" );
    // Replies processed asynchronously in overridden
    // handleReply method
    ...
} catch ( CclException &exception ) {
    cout << "CclClass exception: " << exception.diagnose() << endl;
}
```

Note that the **handleReply** method is run on a separate thread. If the main program needs to know when the reply has been received, a message or semaphore could be used to communicate between the **handleReply** method and the main program.

To make deferred synchronous calls the CclSession object used on the CclTerminal **send** method is created with a synchronization type of **Ccl::dsync**. As in the asynchronous case, a call is made to CICS using the CclTerminal **send** method and control returns immediately to the client application without waiting for a reply from CICS. 3270 screen updates from CICS must be retrieved later using the poll method on the Terminal object:

```
try {
    // Connect to CICS server
    CclTerminal terminal( "CICS1234" );
    // Create deferred synchronous session
    MySession session(Ccl::dsync);
    // Start CESN transaction on CICS server
    terminal.send( &session, "CESN" );
    ...
    if ( terminal.poll())
        // reply processed in handleReply method
    else
        // no reply received yet
} catch ( CclException &exception ) {
    cout << "CclClass exception: " << exception.diagnose() << endl;
}
```

A CICS server transaction may send more than one reply in response to a CclTerminal **send** call. More than one CclTerminal **poll** call may therefore be needed to collect all the replies. Use the CclTerminal **state** method to find out if further replies are expected. If there are, the value returned will be server.

As in the synchronous and asynchronous cases, the **handleReply** method can conveniently be used to encapsulate the code processing the 3270 data returned from CICS from one or more transmissions.

EPI BMS conversion utility

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 datastream 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.

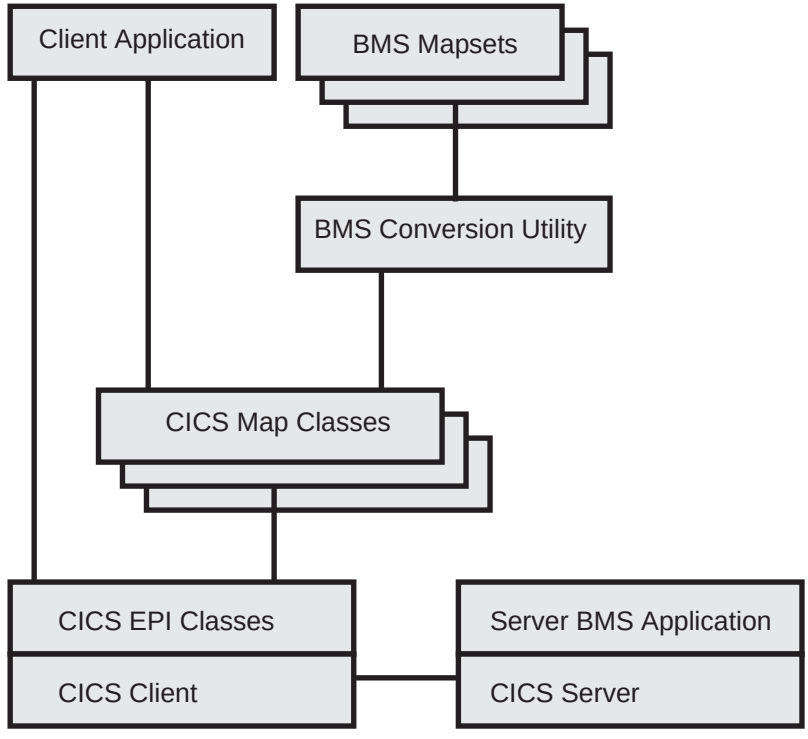

Figure 1. Use of BMS map classes

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 datastreams. 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
```

The utility generates .HPP and .CPP files containing the definition and implementation of the map classes.

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

Mapset containing a single map

The mapset listed in Figure 2 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 2. Sample Map Class—BMS Source

The BMS Conversion Utility generates the C++ class definition (shown in Figure 3 on page 30) 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 Client Classes ************************************
//
// FILE NAME:   epiinq.hpp
//
// DESCRIPTION: C++ header for epiinq.bms
//              Generated by CICS BMS Conversion Utility - Version 1.0
//
//*********************************************************************
#include <cicsepi.hpp>                // CICS Client 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 3. 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 NT and Windows 98 applications using pre-compiled headers, add `#include stdafx.h` to the `.cpp` file generated by CICSBMSC.

CclEPI, CclTerminal and CclSession objects are used in the normal way to start a CICS transaction:

```
try {
   // Initialize CICS client 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.

Security Management for EPI

You are now able to do security management on Servers that support Password Expiry Management. Please refer to the CICS Client General Programming Guide for more information on supported servers and protocols.

To use these features you first must have constructed a Terminal object which is signon incapable, in other words it must have a userid and password (even if they are null). The two methods available are **verifyPassword** which checks the userid and password within the terminal object with the Server Security System, and **changePassword** which allows you to change the password at the server. If successful the connection object password is updated accordingly.

If either call is successful, you are returned a pointer to an internal object which provides information about the security, a CclSecAttr object. This object provides access to information such as last verified Date and Time, Expiry Date and Time and Last access Date and Time. If you query for example last verified Date, you get back a pointer to an object which allows you to get the information in various formats. The following is sample code to show the use of these various objects.

```
// Terminal object already created called term
CclSecAttr *pAttrblock;             // pointer to security attributes
CclSecTime *pDTinfo;                // pointer to Date/Time information
try {
     pAttrblock = term->verifyPassword();
     pDTinfo = pAttrblock->lastVerifiedTime();
     cout << "last verified year  :" <<pDTinfo->year() << endl;
     cout << "last verified month :" <<pDTinfo->month() << endl;
     cout << "last verified day   :" <<pDTinfo->day() << endl;
     cout << "last verified hours :" <<pDTinfo->hours() << endl;
     cout << "last verified mins  :" <<pDTinfo->minutes() << endl;
     cout << "last verified secs  :" <<pDTinfo->seconds() << endl;
     cout << "last verified 100ths:" <<pDTinfo->hundredths() << endl;

// Use a tm structure to produce a single line text of information

     tm mytime;
     mytime = pDTinfo->get_tm();
     cout << "full info:" << asctime(&mytime) << endl;
}
catch (CclException &ex)
{

// Could check for expired password error and handle if required
  cout << "Exception occurred: " <<ex.diagnose()<< endl;
}
```

Note that the security attributes and date/time memory are all handled by the terminal object. If you destroy the terminal object, you destroy the security information being held by that object.

Part 2. CICS Client C++ classes - reference

Chapter 4. CclClass 37
Enumerations 37
Bool. 37
Sync 37
ExCode 37

Chapter 5. CclBuf class 39
CclBuf constructors 40
CclBuf (1) 40
CclBuf (2) 40
CclBuf (3) 40
CclBuf (4) 40
Public methods 41
assign 41
cut 41
dataArea 41
dataAreaLength 41
dataAreaOwner 41
dataAreaType 41
dataLength 42
insert 42
listState 42
operator= (1) 42
operator= (2) 42
operator+= (1) 43
operator+= (2) 43
operator== 43
operator!=. 43
replace 43
setDataLength 44
Enumerations 44
DataAreaOwner. 44
DataAreaType 44

Chapter 6. CclConn class 45
CclConn constructor 45
Public methods 46
alterSecurity 46
cancel 46
changed 46
changePassword 46
link 47
listState 47
makeSecurityDefault 48
password (1) 48
password (2) 48
serverName (1) 48
serverName (2) 48
status 48
serverStatus 49
serverStatusText. 49
userId (1) 49
userId (2) 49
verifyPassword 49
Enumerations 50
ServerStatus 50

Chapter 7. CclECI class 51
CclECI constructor (protected) 51
Public methods 51
exCode. 51
exCodeText 51
handleException 52
instance 52
listState 52
serverCount 52
serverDesc 52
serverName 52

Chapter 8. CclEPI class 55
CclEPI constructor 55
Public methods 55
diagnose 55
exCode. 55
exCodeText 56
handleException 56
serverCount 56
serverDesc 56
serverName 56
state 57
terminate 57
Enumerations 57
State 57

Chapter 9. CclException class 59
Public methods 59
abendCode 59
className 59
diagnose 59
exCode. 59

exCodeText 60
exObject 60
methodName 60

Chapter 10. CclField class 61
Public methods 61
appendText (1) 61
appendText (2) 61
backgroundColor 61
baseAttribute 61
column. 62
dataTag 62
foregroundColor 62
highlight 62
inputProt 62
inputType. 62
intensity 63
length 63
position 63
resetDataTag 63
row 63
setBaseAttribute. 63
setExtAttribute 63
setText (1). 64
setText (2). 64
text 64
textLength 64
transparency 64
Enumerations 65
BaseInts 65
BaseMDT 65
BaseProt 65
BaseType 65
Color 65
Highlight 65
Transparency. 65

Chapter 11. CclFlow class. 67
CclFlow constructor 67
CclFlow (1) 67
CclFlow (2) 67
Public methods 68
abendCode 68
callType 68
callTypeText 68
connection 68
diagnose 68
flowId 68
forceReset. 68
handleReply 69
listState 69
poll 69
setTimeout 69
syncType 70
timeout 70
uow. 70
wait. 70
Enumerations 70
CallType 70

Chapter 12. CclMap class 71
CclMap constructor 71
Public methods 71
exCode. 71
exCodeText 71
field (1) 72
field (2) 72
Protected methods 72
namedField 72
validate 72

Chapter 13. CclScreen class 75
Public methods 75
cursorCol 75
cursorRow 75
depth 75
field (1) 75
field (2) 75
fieldCount 76
mapName. 76
mapSetName. 76
setAID 76
setCursor 76
width 77
Enumerations 77
AID. 77

Chapter 14. CclSecAttr 79
Public Methods 79
expiryTime 79
invalidCount. 79
lastAccessTime 79
lastVerifiedTime. 79

Chapter 15. CclSecTime 81
Public Methods 81
day 81
get_time_t. 81
get_tm 81
hours 81

hundredths 81
minutes 81
month 82
seconds 82
year 82

Chapter 16. CclSession class 83
CclSession constructor 83
Public methods 83
diagnose 83
handleReply 83
state 84
terminal 84
transID 84
Enumerations 84
State 84

Chapter 17. CclTerminal class 85
CclTerminal constructor 85
CclTerminal 85
Public methods 86
alterSecurity 86
changePassword 86
CCSid 87
diagnose 87
disconnect 87
discReason 87
exCode 87
exCodeText 87
install 88
makeSecurityDefault 88
netName 88
password 88
poll 88
queryATI 89
readTimeout 89
receiveATI 89
screen 89
send (1) 89
send (2) 90
setATI 90
signonCapability 90
state 91
serverName 91
termID 91
transID 91
userId 91
verifyPassword 91
Enumerations 91
ATIState 91
signonType 92
State 92
EndTerminalReason 92

Chapter 18. CclUOW class 93
CclUOW constructor 93
Public methods 93
backout 93
commit 93
forceReset 94
listState 94
uowId 94

The following chapters contain descriptions of all the client classes, in alphabetic order. Within the interfaces, there are alphabetical lists of methods. For further information on how to use these interfaces, see Part 1.

Chapter 4. Ccl class

This class defines some enumerations which are used by other classes—both ECI and EPI.

Enumerations

Bool

There are two equivalent pairs of values:

no and yes
off and on

Sync

Possible values are:

async	asynchronous
dsync	deferred synchronous
sync	synchronous

ExCode

For possible values, refer to Table 3 on page 97.

Chapter 5. CclBuf class

A CclBuf object contains a data area in memory which can be used to hold information. A particular use for a CclBuf object is to hold a COMMAREA used to pass data to and from a CICS server.

The CclBuf object is primarily intended for use with byte (binary) data. Typically a COMMAREA will contain an application-specific data structure, often originating from a CICS server COBOL, PL/1 or C program. Methods such as **assign()** and **insert()** therefore provide a **void*** parameter type for application data input. There is limited support for SBCS null-terminated strings (some of the code samples make use of this), but there is no code-page conversion or DBCS support in the **CclBuf** class.

The maximum data length for a buffer is the maximum value for `unsigned long` (2^{32}) for 32 bit platforms. CICS imposes a limit of 32500 bytes in COMMAREA's. This may be reduced by setting the *MaxBufferSize* parameter in the CICS Client initialization file. See the *CICS Clients Administration* manual for more information. If a buffer object used as a COMMAREA is too long, a data length exception is raised.

When a CclBuf object is created it either uses an area of memory passed to it as its buffer, or allocates its own. The length of the data in this buffer can be reduced after the CclBuf object is created. The length of the data in this buffer can only be increased beyond the original length if the CclBuf object is created with a **DataAreaType** of extensible. The alternative to extensible is fixed.

If a buffer object has a **DataAreaType** of fixed and a method is called which would result in its data area length being exceeded, a buffer overflow exception is raised. If the exception is not handled, the buffer will contain the result of the call, truncated to the data area length.

If a method is called that results in a buffer object having a data length smaller than its data area length, the data is padded with nulls.

Many of the methods return object references. This makes it possible for users to chain calls to member functions. For example, the code:

```
CclBuf comma1;
comma1="Some text";
comma1.insert( 9,"inserted ",5) += " at the end";
```

would create the following string:

```
Some inserted text at the end
```

CclBuf constructors

CclBuf (1)

CclBuf(unsigned long *length* **= 0, DataAreaType** *type* **= extensible)**

length
The initial length of the data area, in bytes. The default length is 0.

type
An enumeration indicating whether the data area can be extended. Possible values are extensible or fixed. The default is extensible.

Creates a CclBuf object, allocating its own data area with the given length. All the bytes within it are set to null. The data length is set to zero and will remain zero until data is put in the buffer.

CclBuf (2)

CclBuf(unsigned long *length***, void*** *dataArea***)**

length
The length of the supplied data area, in bytes.

dataArea
The address of the first byte of the supplied data area.

Creates a CclBuf object which cannot be extended, adopting the given data area as its own. The DataAreaOwner is set external.

CclBuf (3)

CclBuf(const char* *text***, DataAreaType** *type* **= extensible)**

text
A string to be copied into the new CclBuf object.

type
An enumeration indicating whether the data area can be extended. Possible values are extensible or fixed. The default is extensible.

Creates a CclBuf object, allocating its own data area with the same length as the *text* string and copies the string into its data area.

CclBuf (4)

CclBuf(const CclBuf& *buffer***)**

buffer
A reference to the CclBuf object which is to be copied.

This copy constructor creates a new CclBuf object which is a copy of the given object. The data length, data area length and data area type of the new buffer are the same as the old buffer. The data area owner of the new buffer is internal.

Public methods

assign

CclBuf& assign(unsigned long *length***, const void*** *dataArea***)**

length
The length of the source data area, in bytes.

dataArea
The address of the source data area.
Overwrites the current contents of the data area with the source data and resets the data length.

cut

CclBuf& cut(unsigned long *length***, unsigned long** *offset* **= 0)**

length
The number of bytes to be cut from the data area.

offset
The offset into the data area. The default is zero.
Cuts the specified data from the data area. Data in the data area is padded with nulls.

dataArea

const void* dataArea(unsigned long *offset* **= 0) const**

offset
The offset into the data area. The default is zero.
Returns the address of the given offset into the data area.

dataAreaLength

unsigned long dataAreaLength() const
Returns the length of the data area in bytes.

dataAreaOwner

DataAreaOwner dataAreaOwner() const

Returns an enumeration value indicating whether the data area has been allocated by the **CclBuf** constructor or has been supplied from elsewhere. Possible values are internal and external.

dataAreaType

DataAreaType dataAreaType() const

Returns an enumeration value indicating whether the data area can be extended. Possible values are extensible and fixed.

dataLength

unsigned long dataLength() const
Returns the length of data in the data area. This cannot be greater than the value returned by **dataAreaLength**.

insert

CclBuf& insert(unsigned long *length***,**
const void* *dataArea***,**
unsigned long *offset* **= 0)**

length
The length of the data, in bytes, to be inserted into the CclBuf object.

dataArea
The start of the source data to be inserted into the CclBuf object.

offset
The offset into the data area where the data is to be inserted. The default is zero.

Inserts the source data into the data area at the given offset.

listState

const char* listState() const
Returns a formatted string containing the current state of the object. For example:

```
Buffer state..&CclBuf=000489B4  &CclBufI=00203A00
dataLength=8  &dataArea=002039C0
dataAreaLength=8  dataAreaOwner=0  dataAreaType=1
```

operator= (1)

CclBuf& operator=(const CclBuf& *buffer***)**

buffer
A reference to a CclBuf object.

Assigns data from another buffer object.

operator= (2)

CclBuf& operator=(const char* *text***)**

text
The string to be assigned to the CclBuf object.

Assigns data from a string.

operator+= (1)

CclBuf& operator+=(const CclBuf& *buffer***)**

buffer
A reference to a CclBuf object.

Appends data from another buffer object to the data in the data area.

operator+= (2)

CclBuf& operator+=(const char* *text***)**

text
The string to be appended to the CclBuf object.

Appends a string to the data in the data area.

operator==

Ccl::Bool operator==(const CclBuf& *buffer***) const**

buffer
A reference to a CclBuf object.

Returns an enumeration indicating whether the data contained in the buffers of the two CclBuf objects is the same. Possible values are yes or no. yes means that the data lengths are the same and the contents are the same.

operator!=

Ccl::Bool operator!=(const CclBuf& *buffer***) const**

buffer
A reference to a CclBuf object.

Returns an enumeration indicating whether the data contained in the buffers of the two CclBuf objects is different. Possible values are yes or no. no means that the data lengths are the same and the contents are the same.

replace

CclBuf& replace(unsigned long *length***,**
const void* *dataArea***,**
unsigned long *offset* **= 0)**

length
The length of the source data area, in bytes.

dataArea
The address of the start of the source data area.

offset
The position where the new data is to be written, relative to the start of the **CclBuf** data area. The default is zero.

Overwrites the current contents of the data area at the given offset with the source data. The data length remains the same.

setDataLength

unsigned long setDataLength(unsigned long *length***)**

length
The new length of the data area, in bytes.

Changes the current length of the data area and returns the new length. If the CclBuf object is not extensible, the data area length is set to either the original length of the data area, or *length*, whichever is less.

If *length* is greater than the data area length, the data is padded with nulls.

Enumerations

DataAreaOwner

Indicates whether the data area of a CclBuf object has been allocated outside the object. Possible values are:

internal
The data area has been allocated by the **CclBuf** constructor.

external
The data area has been allocated externally.

DataAreaType

Indicates whether the data area of a CclBuf object can be made longer than its original length. Possible values are:

extensible
The data area of a CclBuf object can be made longer than its original length.

fixed The data area of a CclBuf object cannot be made longer than its original length.

Chapter 6. CclConn class

An object of class **CclConn** is used to represent an ECI connection between a client and a named server. See “Server connection” on page 12. Access to the server is optionally controlled by a userId and password. It can call a program in the server or get information on the state of the connection. See “Passing data to a server program” on page 12 and “Monitoring server availability” on page 17 for more information.

The creation of a CclConn object does not cause any interaction with the CICS server, nor does it guarantee that the server is available to process requests.

Any interaction between client and server requires the use of a CclFlow object. See “Controlling server interactions” on page 13 for more information.

A CclConn object cannot be copied or assigned. An attempt to delete a CclConn object for which there are active CclFlow or CclUOW objects will raise an activeFlow or an activeUOW exception.

CclConn constructor

```
CclConn(const char* serverName = 0,
        const char* userId = 0,
        const char* password = 0,
        const char* runTran = 0,
        const char* attachTran = 0)
```

serverName
The name of the server. If no name is supplied the default server is used. You can discover this name, after the first call to the server by using the **serverName** method. The length is adjusted to 8 characters by padding with blanks or truncating, if necessary.

userId
The userId, if needed. The length is adjusted to 16 characters by padding with blanks or truncating, if necessary.

password
The password corresponding to the userId in *userID*, if needed. The length is adjusted to 16 characters by padding with blanks or truncating, if necessary.

runTran
The CICS transaction under which the called program will run. The

default is to use the default server transaction. The length is adjusted to 4 characters by padding with blanks or truncating, if necessary.

attachTran
The CICS transaction to which the called program is attached. The default is to use the default CPMI. The length is adjusted to 4 characters by padding with blanks or truncating, if necessary.

This constructor creates a CclConn object; it does not cause any interaction with the CICS server or guarantee that the server is available to process requests. The userId and password are not needed if the connection is only used for status calls or if the server has no security.

Public methods

alterSecurity

void alterSecurity(const char* *newUserid***, const char*** *newPassword***)**

newUserid
The new userid

newPassword
The new password corresponding to the new userid
Updates the Userid and Password to be used on the next link call

cancel

void cancel(CclFlow& *flow***)**

flow
A reference to the CclFlow object used to control the server request call.
Cancels any **changed** call which was previously issued to the server associated with this connection.

changed

void changed(CclFlow& *flow***)**

flow
A reference to the CclFlow object used to control the server request call.
Requests the server to notify the client when the current connection status changes. The call is ignored if there is already an outstanding **changed** call for this connection. Use **serverStatus** or **serverStatusText** to obtain server availability.

changePassword

CclSecAttr* changePassword(const char* *newPassword***)**

newPassword
the new password to be given

Allows a client application to change the password held in the terminal object and the password recorded by an external security manager for the userid held in the terminal object. The external security manager is assumed to be located in the server defined by the terminal object.

link

void link(CclFlow& *flow***,**
const char* *programName***,**
CclBuf* *commarea* **= 0,**
CclUOW* *unit* **= 0)**

flow
A reference to the CclFlow object used to control the server request call.

programName
The name of the server program which is being called. The length is adjusted to 8 characters by padding with blanks or truncating, if necessary.

commarea
A pointer to a CclBuf object which holds the data to be passed to the called program in a COMMAREA. The default is not to pass a COMMAREA.

unit
A pointer to the CclUOW object which identifies the unit of work (UOW) in which this call participates. The default is none. See “Managing logical units of work” on page 18.

Requests execution of the specified program on the server. The server program sees the incoming call as an EXEC CICS LINK call.

If the *commarea* buffer object is too long, a dataLength exception is raised and the request is denied. CICS imposes a limit of 32500 bytes which can be made smaller by using the MaxBufferSize parameter in the CICS Client Initialization file.

listState

const char* listState() const

Returns a formatted string containing the current state of the object. For example:

```
Connection state..&CclConn=000489AC  &CclConnI=00203A50
flowCount=0 &CclFlow(changed)=00000000  token(changed)=0
serverName="server  "  userId="userId          "  password="password        "
&CclUOWI=00000000 runTran="run "  attachTran="att "
```

makeSecurityDefault

void makeSecurityDefault()
Informs the client that the current userid and password for this object is to become the default for ECI and EPI requests passed to the server as specified in the construction of the connection object.

password (1)

const char* password() const
Returns the password held by the CclConn object, padded with spaces to 10 characters, or blanks if there is no password.

password (2)

void password(Ccl::Bool *unpadded***)**

unpadded

Ccl::Yes
returns a null terminated string of the stored password with no space padding in the string.

Ccl::No
returns the string padded with spaces — the same as invoking the password method with no parameters.

serverName (1)

const char* serverName() const
Returns the name of the server system held by the **CclConn** object, padded with spaces, or blanks if the default server is being used and no calls have yet been made.

serverName (2)

void serverName(Ccl::Bool *unpadded***)**

unpadded

Ccl::Yes
returns a null terminated string of the stored server name with no space padding in the string.

Ccl::No
returns the string padded with spaces — the same as invoking the serverName method with no parameters.

status

void status(CclFlow& *flow***)**

flow

A reference to the CclFlow object used to control the server request call. Requests the status of the server connection. When the reply has been received, use **serverStatus** or **serverStatusText** to obtain server availability.

serverStatus

ServerStatus serverStatus() const

Returns an enumeration value, set by an earlier **status** or **changed** request, indicating the availability of the server. Possible values are listed under Enumerations.

serverStatusText

const char* serverStatusText() const
Returns a string, set by an earlier **status** or **changed** request, indicating the availability of the server.

userId (1)

const char* userId() const
Returns the user ID held by the CclConn object, padded with spaces, or blanks if none.

userId (2)

void userId(Ccl::Bool *unpadded***)**

unpadded

Ccl::Yes
returns a null terminated string of the stored userid with no space padding in the string.

Ccl::No
returns the string padded with spaces exactly as invoking the userId method with no parameters.

verifyPassword

CclSecAttr* verifyPassword()
Allows a client application to verify that the password held in the CclConn object matches the password recorded by an external security manager for the userid held in the CclConn object. The external security manager is assumed to be located in the server defined by the CclConn object.

Enumerations

ServerStatus

Indicates the availability of the server. Possible values are:

unknown
: The server status is unknown.

available
: The server is available.

unavailable
: The server is not available.

Chapter 7. CclECI class

Only one instance of the **CclECI** class can exist. It is created by the **instance** class method. It controls the client interface to the available servers.

CclECI should be sub-classed to implement your own handleException method.

Only one instance of a **CclECI** subclass can exist. Any attempt to create more than one will raise the multipleInstance exception.

A CclECI object cannot be copied or assigned.

CclECI constructor (protected)

CclECI()

This constructor is protected and can only be accessed from a subclass.

Public methods

exCode

Deprecated method
Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

Ccl::ExCode exCode() const

Returns an enumeration indicating the most recent exception code. The possible values are listed under Table 3 on page 97.

exCodeText

Deprecated method
Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

const char* exCodeText() const

Returns a text string describing the most recent exception code.

handleException

virtual void handleException(CclException &*except*)

except
A CclException object that contains information about the exception just raised.

This method is called whenever an exception is raised. To deal with exceptions, you should always subclass **CclECI**, and provide your own implementation of **handleException**. See "Handling Exceptions" on page 9. The default implementation just throws the exception object.

instance

static CclECI* instance()
A class method that returns a pointer to the single CclECI object which exists on the client. Here is an example of its use:

```
CclECI* pmgr = CclECI::instance();
```

listState

const char* listState() const
Returns a formatted string containing the current state of the object. For example:

```
ECI state..&CclECI=00203AE0  &CclECII=00203B20
retCode=0  exCode=0
serverCount=0  &serverBuffer=00000000
```

serverCount

unsigned short serverCount() const

Returns the number of available servers to which the client may be connected, as configured in the Client initialization file. In practice, some or all of these servers may not be available. See "Finding potential servers" on page 11.

serverDesc

const char* serverDesc(unsigned short *index* **= 1) const**

index
The index of a connected server in the list. The default index is 1.

Returns the description of the *index*th server. See "Finding potential servers" on page 11.

serverName

const char* serverName(unsigned short *index* **= 1) const**

index

The index of a connected server in the list. The default index is 1. Returns the name of the *index*th server. See “Finding potential servers” on page 11.

Chapter 8. CclEPI class

The **CclEPI** class initializes and terminates the CICS client EPI function. It also has methods which allow you to obtain information about CICS servers configured in the Client Initialization File. You must create one object of this class for each application process before you create CclTerminal objects to connect to CICS servers.

CclEPI constructor

CclEPI()

This method initializes the CICS EPI interface on the client. An initEPI exception is raised if initialization fails. Initialization of the CICS client EPI is synchronous i.e. initialization is complete when the call to the **CclEPI** constructor returns.

Public methods

diagnose

const char* diagnose() const
Returns a character string which holds a description of the condition returned by the most recent server call.

exCode

Deprecated method
Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

Ccl::ExCode exCode() const

Returns an enumeration indicating the most recent exception code. The possible values are listed under Table 3 on page 97.

exCodeText

> **Deprecated method**
> Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

const char* exCodeText() const
Returns a text string describing the most recent exception code.

handleException

virtual void handleException(CclException &*except*)

except
A CclException object that contains information about the exception just raised.
This method is called whenever an exception is raised. To deal with exceptions, you can use try...catch or, you can subclass **CclEPI** and provide your own implementation of **handleException**. The default implementation just throws the exception object.

serverCount

unsigned short serverCount()

Returns the number of available servers to which the client may be connected, as configured in the Client initialization file.

serverDesc

const char* serverDesc(unsigned short *index* **= 1)**

index
The index of a configured server
Returns a description of the selected CICS server, or NULL if no information is available in the Client initialization file for the specified server. If the index exceeds the number of servers configured, a maxServers exception is raised.

serverName

const char* serverName(unsigned short *index* **= 1)**

index
The index of a configured server
Returns the name of the requested CICS server, or NULL if no information is available in the Client initialization file for the specified server. If the index exceeds the number of servers configured, a maxServers exception is raised.

state

State state() const

Returns an enumeration indicating the state of the EPI. Possible values are:
active EPI has been initialized successfully
discon
EPI has terminated
error EPI initialization has failed

terminate

> **Deprecated method**
> Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

void terminate()

Terminates the CICS client EPI in a controlled manner. The CclEPI object remains in being so that anything which occurs during the termination can be monitored by the application.

The terminate method is invoked during CclEPI object destruction, so it is no longer necessary to invoke this method yourself.

Enumerations

State

An enumeration indicating the state of the EPI. Possible values are:
active EPI has been initialized successfully
discon
EPI has terminated
error EPI initialization has failed

Chapter 9. CclException class

A CICS client object constructs an object of the **CclException** class if it encounters a problem.

To deal with such a problem, you should subclass the **CclECI** or **CclEPI** class and provide your own implementation of the **handleException** method. See "Handling Exceptions" on page 9. This method has access to the methods of the CclException object and can be coded to take whatever action is necessary. For example, it can stop the program or pop up a dialog box.

Alternatively, you can use a C++ `try...catch` block to handle exceptions.

A CclException object cannot be assigned and its constructors are intended for use by the CICS client class implementation only.

Public methods

abendCode

const char* abendCode()
Returns a null-terminated string containing the ECI abend code (returns blanks if no abend code available).

className

const char* className() const
Returns the name of the class in which the exception was raised.

diagnose

const char* diagnose() const
Returns text explaining the exception for use in diagnostic output, for example:

```
unknown server, classname=CclFlowI, methodName=afterReply, originCode=13
"link", flowId=2, retCode=-22, abendCode="    "
```

exCode

Ccl::ExCode exCode() const

Returns the exception code. See Table 3 on page 97.

exCodeText

const char* exCodeText() const
Returns a text string that describes the exception code.

exObject

void* exObject() const
This method is only relevant to the ECI.

exObject returns a pointer to the object controlling any server interaction at the time of the exception. If there was no such object, a null pointer is returned.

The pointer should be cast to a **CclFlow***. For example:

```
CclFlow* pflo = (CclFlow*) ex.exObject();
```

methodName

const char* methodName() const
Returns the name of the method in which the exception was raised.

Chapter 10. CclField class

An object of the **CclField** class is responsible for looking after a single field on a 3270 screen. **CclField** objects are created and deleted when 3270 data from the CICS server is processed by a CclScreen object.

Methods in this class allow field text and attributes to be read and updated. Modified fields are sent to the CICS server on the next **send**.

Public methods

appendText (1)

void appendText(const char* *text***, unsigned short** *length***)**

text
The text to be appended to the field

length
The number of characters to be appended to the field

Appends *length* characters from *text* to the end of the text already in the field.

appendText (2)

void appendText(const char* *text***)**

text
The null-terminated text string to be appended to the field

Appends the characters within the *text* string to the end of the text already in the field.

backgroundColor

Color backgroundColor() const

Returns an enumeration indicating the background color of the field. The possible values are shown under **Color** at the end of the description of this class.

baseAttribute

char baseAttribute() const

Returns the 3270 base attribute of the field.

column

unsigned short column() const

Returns the column number of the position of the start of the field on the screen, with the leftmost column being 1.

dataTag

BaseMDT dataTag() const

Returns an enumeration indicating whether the data in the field has been modified. Possible values are:

modified
unmodified

foregroundColor

Color foregroundColor() const

Returns an enumeration indicating the foreground color of the field. The possible values are shown under **Color** at the end of the description of this class.

highlight

Highlight highlight() const

Returns an enumeration indicating which type of highlight is being used. The possible values are shown under **Highlight** at the end of the description of this class.

inputProt

BaseProt inputProt() const

Returns an enumeration indicating whether the field is protected. Possible values are:

protect
unprotect

inputType

BaseType inputType() const

Returns an enumeration indicating the input data type for this field. Possible values are:

alphanumeric
numeric

intensity

BaseInts intensity() const

Returns an enumeration indicating the field intensity. Possible values are :
dark
normal
intense

length

unsigned short length() const

Returns the total length of the field. This includes one byte used to store the 3270 attribute byte information therefore the actual space for data is one less than the value returned by this method. See also the textLength method.

position

unsigned short position() const

Returns the position of the start of the field on the screen, given by position = column number + (*n* x row number), where *n* is the number of columns in a row (usually 80).

resetDataTag

void resetDataTag()

Resets the modified data tag (MDT) to *unmodified*.

row

unsigned short row() const

Returns the row number of the position of the start of the field on the screen. The top row is 1.

setBaseAttribute

void setBaseAttribute(char *attribute***)**

attribute
The value of the base 3270 attribute byte to be entered into the field
Sets the 3270 base attribute.

setExtAttribute

void setExtAttribute(char *attribute***, char** *value***)**

attribute
The type of extended attribute being set

value
The value of the extended attribute

Sets an extended 3270 attribute. If an invalid 3270 attribute type or value is supplied, a parameter exception is raised.

setText (1)

These methods update the field with the given text.

void setText(const char* *text***, unsigned short** *length***)**

text
The text to be entered into the field

length
The number of characters to be entered into the field

Copies *length* characters from *text* into the field.

setText (2)

void setText(const char* *text***)**

text
The null-terminated text to be entered into the field

Copies *text*, without the terminating null, into the field.

text

const char* text() const

Returns the text currently held in the field.

textLength

unsigned short textLength() const

Returns the number of characters currently held in the field.

transparency

Transparency transparency() const

Returns an enumeration indicating the background transparency of the field. Possible values are shown under **Transparency** at the end of the description of this class.

Enumerations

BaseInts

Indicates the field intensity. Possible values are:

normal
intense
dark

BaseMDT

Indicates whether data in the field has been modified. Possible values are:

unmodified
modified

BaseProt

Indicates whether the field is protected. Possible values are:

protect
unprotect

BaseType

Indicates field input data type. Possible values are:

alphanumeric
numeric

Color

Possible values are:

defaultColor	yellow	paleGreen
blue	neutral	paleCyan
red	black	gray
pink	darkBlue	white
green	orange	
cyan	purple	

Highlight

Indicates which type of highlight is being used. Possible values are:

defaultHlt	blinkHlt	underscoreHlt
normalHlt	reverseHlt	intenseHlt

Transparency

Indicates the background transparency of the field. Possible values are:

defaultTran
default transparency

orTran
OR with underlying color

xorTran
XOR with underlying color

opaqueTran
opaque

C++ Class: CclField

Chapter 11. CclFlow class

A CclFlow object is used to control ECI communications for a client/server pair and to determine the synchronization of reply processing. Refer to "Controlling server interactions" on page 13 for an explanation of synchronization. **CclFlow** automatically calls its **handleReply** method when a reply is available; this simplifies control of interleaved replies. **CclFlow** should be subclassed to implement your own **handleReply** method.

A CclFlow object is created for each client server interaction (request from client and response from server). CclFlow objects can be reused when they become inactive, that is, when reply processing is complete. An attempt to delete or reuse an active CclFlow object will raise an activeFlow exception.

CclFlow constructor

CclFlow (1)

CclFlow(Ccl::Sync *syncType***, unsigned long** *stackPages* **= 3)**

syncType
The type of synchronization

stackPages
If asynchronous, the number of 4kb stack pages. The default is 3. If not asynchronous, this parameter is ignored.

CclFlow (2)

CclFlow(Ccl::Sync *syncType***,**
unsigned long *stackPages***,**
const unsigned short &*timeout***)**

syncType
The type of synchronization

stackPages
If asynchronous, the number of 4kb stack pages. If not asynchronous, this parameter is ignored.

timeout
The defined timeout in seconds to wait for the ECI program to respond. If a timeout occurs, then HandleException method is called with a timeout CclException Object. valid values are 0-32767.

Public methods

abendCode

const char* abendCode() const
Returns the abend code from the most recently executed CICS transaction, or blank if there have been none.

callType

CallType callType() const

Returns an enumeration value indicating the most recent type of server request.

callTypeText

const char* callTypeText() const
Returns the name of the most recent server request.

connection

CclConn* connection() const
Returns a pointer to the CclConn object which represents the server being used, if any, or zeros.

diagnose

const char* diagnose() const
Returns text explaining the exception for use in diagnostic output; for example:

```
"link", flowId=2, retCode=-22, abendCode="    "
```

flowId

unsigned short flowId() const
Returns the unique identity of this CclFlow object.

forceReset

void forceReset()

Make the flow inactive and reset it. Typically used to prepare a CclFlow object for re-use or deletion after a flow has been abandoned, for example, when a C++ throw is used in a exception handler. This applies only to dsync and async flows. You cannot issue this on a sync call from another thread.

handleReply

virtual void handleReply(CclBuf* *commarea***)**

commarea
A pointer to the CclBuf object containing the returned COMMAREA or zero if none.

This method is called whenever a reply is received from a server, irrespective of the type of synchronization or the type of call. See "Controlling server interactions" on page 13. To deal with replies, you should subclass **CclFlow** and provide your own implementation of **handleReply**. The default implementation just returns to the caller.

listState

const char* listState() const

Returns a formatted string containing the current state of the object. For example:

```
Flow state..&CclFlow=000489A4  &CclFlowI=00203B70
syncType=2  threadId=0  stackPages=9  callType=0  flowId=0  commLength=0
retCode=0  systemRC=0  abendCode="    "  &CclConnI=00000000  &CclUOWI=00000000
```

poll

Ccl::Bool poll(CclBuf* *commarea* **= 0)**

commarea
An optional pointer to the CclBuf object which will be used to contain the returned COMMAREA.

Returns an enumeration, defined within the **Ccl** class indicating whether a reply has been received from a deferred synchronous **Backout**, **Cancel**, **Changed**, **Commit**, **Link**, or **Status** call request. If **poll** is used on a flow object which is not deferred synchronous, a syncType exception is raised. Possible values are:

yes A reply has been received. **handleReply** has been called synchronously.

no No reply has been received. The client process is not blocked.

setTimeout

void setTimeout(const unsigned short &*timeout***)**

timeout
the defined timeout in seconds to wait for the ECI program to respond. If a timeout occurs, then HandleException method is called with a timeout CclException Object. Valid values are 0-32767.

Sets the timeout value for the flow object for the next activation of the flow. This value can be set while a flow is active but doesn't affect the current active flow

syncType

Ccl::Sync syncType() const

Returns an enumeration, defined within the **Ccl** class indicating the type of synchronization being used. Possible values are shown in “Sync” on page 37.

timeout

short timeout()
Retrieves the current timeout value set for the flow object.

uow

CclUOW* uow() const
If there is CclUOW object which contains information on any unit of work (UOW) which is associated with this interaction, returns a pointer to it.

wait

void wait()

Waits for a reply from the server, blocking the client process in the meantime. If **wait** is used on a synchronous flow object, a syncType exception is raised.

Enumerations

CallType

The possible values for server requests in progress under the control of a CclFlow object are:

inactive
No server call is currently in progress

link A **CclConn::link** call to a server program

backout
A **CclUOW::backout** call to back out changes made to recoverable resources on the server

commit
A **CclUOW::commit** call to commit changes made to recoverable resources on the server

status A **CclConn::status** call to determine the status of a server connection

changed
A **CclConn::changed** call to request notification when the status of a connection to a server changes

cancel
A **CclConn::cancel** call to cancel an earlier **CclConn::changed** request.

Chapter 12. CclMap class

The **CclMap** class is a base class for map classes created by the CICS BMS Map Conversion Utility. The methods provided by **CclMap** class are inherited by the classes generated from BMS maps.

CclMap constructor

CclMap(CclScreen* *screen***)**

screen
A pointer to the matching CclScreen object.

Creates a CclMap object and checks (validates) that the map matches the content of the screen, defined by the CclScreen object. If validation was unsuccessful, an invalidMap exception is raised. If the supplied CclScreen object is invalid, an parameter exception is raised.

Public methods

exCode

Deprecated method
Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

Ccl::ExCode exCode() const
Returns an enumeration indicating the most recent exception code. The possible values are listed in Table 3 on page 97.

exCodeText

Deprecated method
Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

const char* exCodeText() const
Returns a text string describing the most recent exception code.

field (1)

CclField* field(unsigned short *index***)**

index
The index number of the required CclField object.
Returns a pointer to the CclField object identified by *index* in the BMS map.

field (2)

CclField* field(unsigned short *row***, unsigned short** *column***)**

row
The row number of the required CclField object within the map. 1 denotes the top row.

column
The column number of the required CclField object within the map. 1 denotes the left column.
Returns a pointer to the CclField object identified by position in the BMS map.

Protected methods

namedField

CclField* namedField(unsigned long *index***)**

index
The index number of the required CclField object.
Returns the address of the *index*th object.

validate

void validate(const MapData* *map***,**
const FieldIndex* *index***,**
const FieldData*
*fields***)**

map
A structure which contains information about the map. The structure is defined within this class and contains the following members which are all unsigned short integers:

row	Map row position on screen
col	Map column position on screen
width	Map width in columns
depth	Map depth in rows
fields	Number of fields

labels

Number of labeled fields

index

The index number of the required CclField object. **FieldIndex** is a typedef of this class and is equivalent to an unsigned short integer.

fields

A structure which contains information about a particular field. The structure is defined within this class and contains the following members which are all unsigned short integers:

row Field row (within map)

col Field column (within map)

len Field length

Validate map against the current screen.

Chapter 13. CclScreen class

The **CclScreen** EPI class maintains all data on the 3270 virtual screen and provides access to this data. It contains a collection of CclField objects which represent the fields on the current 3270 screen.

A single CclScreen object is created by the CclTerminal object, and should be obtained by using the **screen** method on the CclTerminal object. The CclScreen object is updated by the CclTerminal object when 3270 data is received from CICS. A dataStream exception is raised if an unsupported datastream is received.

Public methods

cursorCol

unsigned short cursorCol() const

Returns the column number of the current position of the cursor. Leftmost column = 1.

cursorRow

unsigned short cursorRow() const

Returns the row number of the current position of the cursor. Top row = 1.

depth

unsigned short depth() const

Returns the number of rows in the screen.

field (1)

These methods allow you to access fields on the current screen by returning a pointer to the relevant CclField object.

CclField* field(unsigned short *index***)**

index
The index number of the field of interest

field (2)

CclField* field(unsigned short *row***, unsigned short** *column***)**

row
The row number of the field

column
The column number of the field

fieldCount

unsigned short fieldCount() const

Returns the number of fields in the screen.

mapName

const char* mapName()
Returns a padded null terminated string specifying the name of the map that was most recently referenced in the MAP option of a SEND MAP command processed for the terminal resource. If the terminal resource is not supported by BMS, or the server has no record of any map being sent, the value returned is spaces.

mapSetName

const char* mapSetName()
Returns a padded null terminated string specifying the name of the mapset that was most recently referenced in the MAPSET option of a SEND MAP command processed for the terminal resource. If the MAPSET option was not specified on the most recent request, BMS used the map name as the mapset name. In both cases, the mapset name used may have been suffixed by a terminal suffix. If the terminal resource is not supported by BMS, or the server has no record of any mapset being sent, the value returned is spaces.

setAID

void setAID(const AID *key***)**

key
An AID key. See the AID enumerations at the end of this §.
Sets the AID key value to be passed to the server on the next transmission.

setCursor

void setCursor(unsigned short *row***, unsigned short** *col***)**

row
The required row number of the cursor. 1 denotes the top row.

col
The required column number of the cursor. 1 denotes the left column.

Requests that the cursor position be set. If the supplied row or column values are outside the screen boundaries, a parameter exception is raised.

width

unsigned short width() const

Returns the number of columns on the screen.

Enumerations

AID

Indicates an AID key. Possible values are:

enter
clear
PA1—PA3
PF1—PF24

Chapter 14. CclSecAttr

The CclSecAttr class provides information about passwords reported back by the external security manager when issuing verifyPassword or changePassword methods on CclConn or CclTerminal objects.

This object is created and owned by the CclConn or CclTerminal Object and access to this object is provided when invoking the verifyPassword or changePassword methods

Public Methods

expiryTime

CclSecTime* expiryTime() const
Returns a CclSecTime Object which contains the Date and Time at which the password will expire

invalidCount

unsigned short invalidCount() const
Returns the Number of times that an invalid password has been entered for the userid.

lastAccessTime

CclSecTime* lastAccessTime() const
Returns a CclSecTime Object which contains the Date and Time of the userid was last accessed

lastVerifiedTime

CclSecTime* lastVerifiedTime() const
Returns a CclSecTime Object which contains the Date and Time of the Last Verification

Chapter 15. CclSecTime

The CclSecTime class provides date and time information in the CclSecAttr object for various entries reported back by the external security manager when issuing verifyPassword or changePassword methods on CclConn or CclTerminal objects.

These objects are created and owned by the CclSecAttr object and access is obtained via the various methods available on this object. No Constructors or Destructors are available.

Public Methods

day

unsigned short day() const
Returns the day with a range from 1 to 31, 1 represents the 1st day of the month

get_time_t

time_t get_time_t() const
Returns the date and time in a time_t format

get_tm

tm get_tm() const
Returns the date and time in a tm structure

hours

unsigned short hours() const
Returns the hours with a range from 0 to 23

hundredths

unsigned short hundredths() const
Returns the hundredths of seconds with a range from 0 to 99

minutes

unsigned short minutes() const
Returns the minutes with a range from 0 to 59

month

unsigned short month() const
Returns the month with a range from 1 to 12, 1 represents January

seconds

unsigned short seconds() const
Returns the seconds with a range from 0 to 59

year

unsigned short year() const
Returns a 4 digit Year

Chapter 16. CclSession class

The **CclSession** class allows the programmer to implement reusable code to handle a segment (one or more transmissions) of a 3270 conversation. In multi-threaded environments, such as OS/2, it provides asynchronous handling of replies from CICS.

The **CclSession** class controls the flow of data to and from CICS within a single 3270 session. You should derive your own classes from **CclSession**.

CclSession constructor

CclSession(Ccl::Sync *syncType***)**

syncType
The protocol to be used on transmissions to the CICS server. Possible values are:
async asynchronous
dsync deferred synchronous
sync synchronous

Public methods

diagnose

const char* diagnose() const
Returns a text description of the last error.

handleReply

virtual void handleReply(State *state***, CclScreen*** *screen***)**

state
An enumeration indicating the state of the data flow. The scope of the values is within this class and they are shown under **State** at the end of the description of this class.

screen
A pointer to the CclScreen object
This is a virtual method which you can override when you develop your own class derived from **CclSession**. It is called when data is received from CICS.

state

State state() const

Returns an enumeration indicating the current state of the session. Possible values are shown under **State** at the end of the description of this class.

terminal

CclTerminal* terminal() const
Returns a pointer to the CclTerminal object for this session. Note that this method will return a NULL pointer until the CclSession object has been associated with a CclTerminal object (that is, until the CclSession object has been used as a parameter on a CclTerminal **send** method).

transID

const char* transID() const
Returns the 4-letter name of the current transaction.

Enumerations

State

Indicates the state of a session. Possible values are:

idle The terminal is connected and there is no CICS transaction in progress

server
There is a CICS transaction in progress in the server

client There is a CICS transaction in progress and the server is waiting for a response from the client

discon
The terminal is disconnected

error There is an error in the terminal

Chapter 17. CclTerminal class

An object of class **CclTerminal** represents a 3270 terminal connection to a CICS server. A CICS connection is established when the object is created. Methods can then be used to converse with a 3270 terminal application (often a BMS application) in the CICS server.

The EPI must be initialized (that is, a CclEPI object created) before a CclTerminal object can be created.

CclTerminal constructor

CclTerminal

CclTerminal(const char* *server***,**
const char* *devtype***,**
const char* *netname***,**
signonType *signonCapability*
const char* *userid*
const char* *password*
const unsigned short *&readTimeOut***,**
const unsigned short *CCSid*)

server
The name of the server with which you require to communicate. If no name is provided the default server system is assumed. The length is adjusted to 8 characters by padding with blanks.

devtype
The name of the model terminal definition which the server uses to generate a terminal resource definition. If no string is provided the default model is used. The length is adjusted to 16 characters by padding with blanks.

netname
The name of the terminal resource to be installed or reserved. The default is to use the contents of *devtype*. The length is adjusted to 8 characters by padding with blanks.

signonCapability

Sets the type of signon capability for the terminal.

Possible values are:

```
CclTerminal::SignonCapable
```

```
CclTerminal::SignonIncapable
```

userid
The name of the userid to associate with this terminal resource

password
The password to associate with the userid

readTimeOut
A value in the range 0 through 3600, specifying the maximum time in seconds between the time the classes go clientrepl state and the application program invokes the reply method.

CCSid
An unsigned short specifying the coded character set indentifier(CCSID) that identifies the coded graphic character set used by the client application for data passed between the terminal resource and CICS transactions. A zero string implies a default will be used.

Creates a Terminal object which doesn't do an implicit install terminal. You must run the install method to install the terminal.

Public methods

alterSecurity

void alterSecurity(const char* *userid***,const char*** *password***)**

userid
The new userid

password
The new password for *userid*

Allows you to re-define the userid and password for a terminal resource. You may call the method before you install a terminal. It will only change the terminal definition and the new userid and password will be used for the terminal when install is called.

changePassword

CclSecAttr* changePassword(const char* newPassword)

newPassword
The new password

Allows a client application to change the password held in the terminal object and the password recorded by an external security manager for the userid held in the terminal object. The external security manager is assumed to be located in the server defined by the terminal object.

CCSid

unsigned short CCSid()
Returns the selected code page as an unsigned short .

diagnose

const char* diagnose()

Returns a character string which holds a description of the error returned by the most recent server call.

disconnect

void disconnect()

This will disconnect the terminal and delete it. If a transaction was running at the time, the transaction will be purged and the terminal deleted.

discReason

void discReason(void)

Returns the reasons for a disconnection. See “EndTerminalReason” on page 92.

exCode

Deprecated method
Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

Ccl::ExCode exCode() const

Returns an enumeration indicating the most recent exception code. The possible values are listed in Table 3 on page 97.

exCodeText

Deprecated method
Do not use this method in new applications.The method has been deprecated and is only provided for backwards compatibility.

const char* exCodeText() const
Returns a text string describing the most recent exception code.

install

void install(CclSession *_session_,
const unsigned short &_installTimeOut_)

session
A pointer to the CclSession object which is to be used for the CICS server interaction.

installTimeOut
A value in the range 0 through 3600, specifying the maximum time in seconds that installation of the terminal resource is allowed to take. A value of 0 means that no limit is set

Connects a non-connected terminal resource. Throws an invalidState error if already connected, or a timeout error if a timeout occurs.

makeSecurityDefault

void makeSecurityDefault()
Informs the client that the current userid and password for this object is to become the default for ECI and EPI requests passed to the server as specified in the construction of the Terminal object.

netName

const char* netName() const
Returns the network name of the terminal as a null terminated string .

password

const char* password()
Returns a null terminated string containing the current password setting for the terminal, Null if none.

poll

Ccl::Bool poll()

Polls for data from the CICS server.

For deferred synchronous transmissions (that is, if a deferred synchronous CclSession object was used on a previous send call) the **poll** method is called by the application when it wishes to receive data from the CICS server. If a reply from CICS is ready, the CclTerminal object updates the CclScreen object with the contents of the 3270 datastream received from CICS, the **handleReply** virtual function on the CclSession object is called, and the **poll** method returns `Ccl::yes`. If no reply has been received from CICS yet, the **poll** method returns `Ccl::no`.

The **poll** method is only used for deferred synchronous transmissions, a syncType exception is raised if the poll method is called when a synchronous or asynchronous session is in use. An invalidState exception is raised if the **poll** method is called when there was no previous **send** call. The CclTerminal object should be in server state for poll to be called.

A CICS server transaction may send more than one reply in response to a CclTerminal **send** call. More than one CclTerminal **poll** call may therefore be needed to collect all the replies. Use the CclTerminal **state** method to find out if further replies are expected. If there are, the value returned will be server. See "EPI call synchronization types" on page 25.

queryATI

ATIState queryATI()

Returns an enumeration indicating whether the "Automatic Transaction Initiation" (ATI) is enabled or disabled. Possible values are:

disabled
enabled

readTimeout

const char* readTimeout()
Returns the read timout value for the terminal as a null terminated string .

receiveATI

void receiveATI(CclSession* *session***)**

session
pointer to the CclSession object which is to be used for the CICS server interaction.

Waits for and receives 3270 datastream for a CICS ATI transaction. The CclSession object supplied as a parameter determines whether the call is synchronous or asynchronous, and can be subclassed to provide a reply handler

screen

CclScreen* screen() const
Returns a pointer to the CclScreen object that is handling the 3270 screen associated with this terminal session.

send (1)

void send(CclSession* *session***,**
const char* *transid***,**
const char* *startdata* **= NULL)**

session
A pointer to the CclSession object which controls the session which is to be used. If no valid CclSession object is supplied, a parameter exception is raised.

transid
The name of the transaction which is to be started

startdata
start transaction data. The default is to have no data for the transaction being started.

Formats and sends a 3270 datastream, starting the named transaction. The CclTerminal object must be in idle state (that is, connected to a CICS server but with no transaction in progress). If the object is not in idle state, an invalidState exception is raised.

send (2)

void send(CclSession* *session***)**

The *session* parameter is described above.

Formats and sends a 3270 datastream. The CclTerminal object must be idle state (see above) or in client state (that is, with a transaction in progress and the CICS server is waiting for a response). If the object is not in idle or client state, an invalidState exception is raised.

setATI

void setATI(ATIState *newstate***)**

newstate
An enumeration indicating whether the ATI is to be enabled or disabled. The scope of the values is within this class and the possible values are disabled and enabled.

signonCapability

signonType signonCapability()
Returns the type of signon capability applied to the terminal at installation.

Possible values are:

```
CclTerminal::signonCapable
CclTerminal::signonIncapable
CclTerminal::signonUnknown
```

state

State state() const

Returns an enumeration indicating the current state of the session. Possible values are shown at the end of the description of this class.

serverName

const char* serverName() const
Returns the name of the CICS server to which this terminal session is connected.

termID

const char* termID() const
Returns the 4-character terminal ID.

transID

const char* transID() const
Returns the 4-character name of the current CICS transaction.Note that if a RETURN IMMEDIATE is run from the current transaction, TransId will not provide the name of the new transaction, it will still contain the name of the first transaction.

userId

const char* userId()
Returns a null terminated string containing the current userid setting for the terminal, Null if none.

verifyPassword

CclSecAttr* verifyPassword()
Allows a client application to verify that the password held in the terminal object matches the password recorded by an external security manager for the userid held in the terminal object. The external security manager is assumed to be located in the server defined by the terminal object.

Enumerations

ATIState

Indicates whether “Automatic Transaction Initiation” (ATI) is enabled or disabled. Possible values are:

enabled
disabled

signonType

Indicates the signon capability of a terminal. Possible values are:

signonCapable
Signon Capable

signonIncapable
Signon Incapable

signonUnknown
Signon Unknown

State

Indicates the state of the CclTerminal object. Possible values are:

client There is a CICS transaction in progress and the server is waiting for a response from the client

discon
The terminal is disconnected

error There is an error in the terminal

idle The terminal is connected and there is no CICS transaction in progress

server
There is a CICS transaction in progress in the server

termDefined
A terminal has been defined but not installed.

EndTerminalReason

Indicates the EndTerminalReason of the CclTerminal object. Possible values are:

signoff
A disconnect was requested or the user has signed off the terminal.

shutdown
The CICS server has been shutdown.

outofService
The terminal has been switched to out of service.

unknown
An unknown situation has occurred.

failed The terminal failed to disconnect.

notDiscon
The terminal is not disconnected.

Chapter 18. CclUOW class

Use this ECI class when you make updates to recoverable resources in the server within a “unit of work” (UOW). Each update in a UOW is identified at the client by a reference to its **CclUOW**—see **link** in **CclConn** (“link” on page 47).

A CclUOW object cannot be copied or assigned. An attempt to delete a CclUOW object for which there is an active CclFlow object will raise an activeFlow exception. An attempt to delete an active CclUOW object, that is one which has not been committed or backed out, will raise an activeUOW exception.

CclUOW constructor

CclUOW()

Creates a CclUOW object.

Public methods

backout

void backout(CclFlow& *flow***)**

flow
A reference to the CclFlow object which is used to control the client-server call

Terminate this UOW and back out all changes made to recoverable resources in the server.

commit

void commit(CclFlow& *flow***)**

flow
A reference to the CclFlow object which is used to control the client-server call

Terminate this UOW and commit all changes made to recoverable resources in the server.

forceReset

void forceReset()

Make this UOW inactive and reset it.

listState

const char* listState() const
Returns a zero-terminated formatted string containing the current state of the object. For example:

```
UOW state..&CclUOW=0004899C  &CclUOWI=00203BD0
&CclConnI=00000000  uowId=0  &CclFlowI=00000000
```

uowId

unsigned long uowId() const

Returns the identifier of the UOW. 0 indicates that the UOW is either complete or has not yet started. In other words, it is inactive.

Part 3. Appendixes

Appendix. Exception Objects

All exception objects provide the following information

- Class Name
- Method Name
- Exception Code
- Exception Text
- Abend Code (ECI Only)
- Origin Point

The Class name can contain a trailing 'I' for example CclFlowI which implies it is a internally contained class for the well known class, e.g. CclFlowI is contained by CclFlow. if an internal class is reported the method reported may be an internal method, not an external one.

The Origin Point is a unique value which defines the exact point within the class library where the exception was generated. These are mainly useful for service.

The more important items of information are the Exception Code, Exception Text and Abend Code (ECI only). The following is a Summary of these Exception Codes and Text and whether they are relevent to ECI or EPI or both.Enumeration

Table 3. Exception codes

Enumeration	Text	Description	ECI	EPI
Ccl::noError	no error	No error occurred	Yes	Yes
Ccl::bufferOverflow	buffer overflow	Attempted to increase a CclBuf object which isn't Extensible	Yes	
Ccl::multipleInstance	multiple instance	Attempted to create more than one ECI object	Yes	
Ccl::activeFlow	flow is active	Current Flow is still active, you cannot use this flow until it is inactive	Yes	
Ccl::activeUOW	UOW is active	Current UOW is still active, you need to backout or commit.	Yes	

Table 3. Exception codes (continued)

Enumeration	Text	Description	ECI	EPI
Ccl::syncType	sync error	Incorrect synchronisation type for method call.	Yes	Yes
Ccl::threadCreate	thread create error	Internal thread creation error	Yes	Yes
Ccl::threadWait	thread wait error	Internal thread wait error	Yes	
Ccl::threadKill	thread kill error	Internal thread kill error	Yes	
Ccl::dataLength	data length invalid	CommArea > 32768 Bytes or inbound 3270 data stream too large for Terminal Buffer size.	Yes	Yes
Ccl::noCICS	no CICS	The client is unavailable, or the server implementation is unavailable, or a logical unit of work was to be begun, but the CICS server specified is not available. No resources have been updated	Yes	Yes
Ccl::CICSDied	CICS died	A logical unit of work was to be begun or continued, but the CICS server was no longer available. If this is a link call with an active UOW, the changes are backed out. If This was a UOW Commit or the application cannot determine whether the changes have been committed or backed out, and must log this condition to aid future manual recovery	Yes	
Ccl::noReply	no reply	There was no outstanding reply	Yes	

Table 3. Exception codes (continued)

Enumeration	Text	Description	ECI	EPI
Ccl::transaction	transaction abend	ECI Program Abended	Yes	
Ccl::systemError	system error	Unknown internal error occurred	Yes	Yes
Ccl::resource	resource shortage	The server implementation or the client did not have enough resources to complete the request e.g. insufficient SNA sessions.	Yes	Yes
Ccl::maxUOWs	exceeded max UOWs	A new logical unit of work was being created, but the application already has as many outstanding logical units of work as the configuration will support.	Yes	
Ccl::unknownServer	unknown server	The requested server could not be located	Yes	Yes
Ccl::security	security error	You did not supply a valid combination of user ID and password, though the server expects it.	Yes	Yes
Ccl::maxServers	exceeded max servers	You attempted to start requests to more servers than your configuration allows. You should consult the documentation for your client or server to see how to control the number of servers you can use.	Yes	Yes

Table 3. Exception codes (continued)

Enumeration	Text	Description	ECI	EPI
Ccl::maxRequests	exceeded max requests	There were not enough communication resources to satisfy the request. You should consult the documentation for your client or server to see how to control communication resources	Yes	Yes
Ccl::rolledBack	rolled back	An attempt was made to commit a logical unit of work, but the server was unable to commit the changes, and backed them out instead	Yes	
Ccl::parameter	parameter error	Incorrect parameter supplied	Yes	Yes
Ccl::invalidState	invalid object state	The Object is not in the correct state to invoke the method, e.g. terminal object still in server state and an attempt to send data is made.	Yes	Yes
Ccl::transId	invalid transaction	Null transid supplied or returned for a Pseudo Conversional transaction		Yes
Ccl::initEPI	EPI not initialised	No EPI object or EPI failed to initiliase correctly		Yes
Ccl::connect	connection failed	Unexpected error trying to add the terminal		Yes
Ccl::dataStream	3270 datastream error	Unsupported Data Stream		Yes
Ccl::invalidMap	map/screen mismatch	Map definition and Screen do not match		Yes
Ccl::cclClass	CICS class error	Unknown internal Class error occurred.	Yes	Yes

Table 3. Exception codes (continued)

Enumeration	Text	Description	ECI	EPI
Ccl::startTranFailure	Start Transaction Failure	Transaction failed to start		Yes
Ccl::timeout	Timeout Occurred	Timeout occurred before response from Server	Yes	Yes
Ccl::noPassword	Password is Null	The object's password is null.	Yes	Yes
Ccl::noUserid	Userid is Null	The object's userid is null	Yes	Yes
Ccl::nullNewPassword	A NULL new password was supplied	The provided password is null	Yes	Yes
Ccl::pemNotSupported	PEM is not supported on the server	The CICS Server doesn't support the Password Expiry Management facilities. The method cannot be used	Yes	Yes
Ccl::pemNotActive	PEM is not active on the server	Password Expiry Management is not active	Yes	Yes
Ccl::passwordExpired	Password has expired	The password has expired. No information has been returned	Yes	Yes
Ccl::passwordInvalid	Password is invalid	The password is invalid.	Yes	Yes
Ccl::passwordRejected	New password was rejected	Change password failed because the password doesn't conform to standards defined	Yes	Yes
Ccl::useridInvalid	Userid unknown at server	The userid is unknown	Yes	Yes
Ccl:invalidTermid	Termid is invalid	The terminal ID is invalid		Yes
Ccl:invalidModelid	Modelid is invalid	Invalid Model/Device Type		Yes
Ccl:not3270	Not a 3270 device	Not a 3270 device		Yes

Table 3. Exception codes (continued)

Enumeration	Text	Description	ECI	EPI
Ccl:invalidCCSid	Codepage (CCSid value) is invalid	Invalid CCSid		Yes
Ccl:serverBusy	Server is too busy	CICS server is busy		Yes
Ccl:signonNotPossible	Signon Capable terminal is not possible	The server does not allow the terminal to be installed as signon capable.		Yes

Glossary

AID. Attention Identifier

ATI. Automatic Transaction Initiation

BMS. Basic Mapping Support

COMMAREA. CICS Communications Area

ECI. External Call Interface

EPI. External Presentation Interface

ESI. External Security Interface

ISC. Inter-Systems Communication

OLE. Object Linking and Embedding

UOW. Unit of Work

Bibliography

Here are some books that you may find useful.

C++ Programming

You should read the books supplied with your C++ compiler.

The following are some non-IBM publications that are generally available. This is not an exhaustive list. IBM does not specifically recommend these books, and other publications may be available in your local library or bookstore.

- Lippman, Stanley B., *C++ Primer*: Addison-Wesley Publishing Company.
- Stroustrup, Bjarne, *The C++ Programming Language*: Addison-Wesley Publishing Company.
- Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++Reference Manual*: Addison-Wesley Publishing Company.

The CICS Transaction Gateway and CICS Universal Clients library

This chapter lists all the CICS Transaction Gateway, CICS Universal Clients, and related books, and discusses the various forms in which they are available.

The headings in this chapter are:

- "CICS Transaction Gateway books"
- "CICS Universal Clients books" on page 106
- "CICS Family publications" on page 106
- "Book filenames" on page 106
- "Sample configuration documents" on page 107
- "Other publications" on page 107
- "Viewing the online documentation" on page 107

CICS Transaction Gateway books

- *CICS Transaction Gateway for OS/2 Administration*, SC34-5590

  This book describes the administration of the CICS Transaction Gateway for OS/2.
- *CICS Transaction Gateway for Windows Administration*, SC34-5589

  This book describes the administration of CICS Transaction Gateway for Windows 98 and CICS Transaction Gateway for Windows NT.
- *CICS Transaction Gateway for AIX Administration*, SC34-5591

  This book describes the administration of the CICS Transaction Gateway for AIX.
- *CICS Transaction Gateway for Solaris Administration*, SC34-5592

  This book describes the administration of the CICS Transaction Gateway for Solaris.
- *CICS Transaction Gateway for OS/390 Administration*, SC34-5528

  This book describes the administration of the CICS Transaction Gateway for OS/390.
- *CICS Transaction Gateway Messages*

  This online book lists and explains the error messages that can be generated by CICS Transaction Gateway.

  You cannot order this book.
- *CICS Transaction Gateway Programming*, SC34-5594

This book provides an introduction to Java programming with the CICS Transaction Gateway.

There are also additional HTML pages that contain programming reference information.

CICS Universal Clients books

- *CICS Universal Client for OS/2 Administration*, SC34-5450

  This book describes the administration of the CICS Universal Client for OS/2.

- *CICS Universal Client for Windows Administration*, SC34-5449

  This book describes the administration of the CICS Universal Client for Windows 98 and CICS Universal Client for Windows NT.

- *CICS Universal Client for AIX Administration*, SC34-5348

  This book describes the administration of the CICS Universal Client for AIX.

- *CICS Universal Client for Solaris Administration*, SC34-5451

  This book describes the administration of the CICS Universal Client for Solaris.

- *IBM CICS Universal Clients Messages*

  This online book lists and explains the error and trace messages that can be generated by CICS Universal Clients.

  You cannot order this book.

- *IBM CICS Universal Clients C++ Programming*, SC33-1923

  This book describes how to write object oriented programs for the ECI and EPI in the C++ language.

- *IBM CICS Universal Clients COM Automation Programming*, SC33-1924

  This book describes how to write object oriented programs for the ECI and EPI according to the Component Object Model (COM) standard.

CICS Family publications

- *CICS Family: Client/Server Programming*, SC33-1435

  This book describes the programming interfaces associated with CICS client/server Programming— the External Call Interface (ECI), the External Presentation Interface (EPI), and the External Security Interface (ESI). It is intended for application designers and programmers who wish to develop client applications to communicate with CICS server systems.

Book filenames

Table 4 on page 107 show the softcopy filenames of the CICS Transaction Gateway and CICS Universal Client books.

Table 4. CICS Transaction Gateway and CICS Universal Clients books and file names

Book title	File name
IBM CICS Universal Clients Messages	CCLHAB
CICS Universal Client for AIX Administration	CCLHAD
CICS Universal Client for OS/2 Administration	CCLHAE
CICS Universal Client for Windows Administration	CCLHAF
CICS Universal Client for Solaris Administration	CCLHAG
CICS Transaction Gateway for OS/390 Administration	CCLHAI
CICS Transaction Gateway Messages	CCLHAJ
CICS Transaction Gateway Programming	CCLHAK
CICS Transaction Gateway for Windows Administration	CCLHAL
CICS Transaction Gateway for OS/2 Administration	CCLHAM
CICS Transaction Gateway for AIX Administration	CCLHAN
CICS Transaction Gateway for Solaris Administration	CCLHAO
IBM CICS Universal Clients C++ Programming	CCLHAP
IBM CICS Universal Clients COM Automation Programming	CCLHAQ
CICS Family: Client/Server Programming	DFHZAD
Note: The File names in this table do not include the 2-digit suffix.	

Sample configuration documents

A number of sample configuration documents are available in the Portable Document Format (PDF) format.

These documents provide step-by-step guidance to help you, for example, in configuring your CICS Universal Clients for communication with CICS servers, using various protocols. They provide detailed instructions that extend the information in the CICS Transaction Gateway and CICS Universal Client libraries.

As more sample configuration documents become available, you can download them from our Web site; go to:

`http://www.ibm.com/software/ts/cics/`

and follow the **Library** link.

Other publications

The following International Technical Support Organization (ITSO) Redbook publication contains many examples of client/server configurations:

- *Revealed! CICS Transaction Gateway with more CICS Clients Unmasked, SG24-5277*

  This book supersedes the following book:

- *CICS Clients Unmasked, GG24-2534*

You can obtain ITSO Redbooks from a number of sources. For the latest information, see:

`http://www.ibm.com/redbooks/`

You can find information on CICS products at:

`http://www.ibm.com/software/ts/cics/`

Viewing the online documentation

You can access all of the documentation provided with CICS Transaction Gateway and CICS Universal Client in our online library. You need Adobe Acrobat Reader and a suitable Web browser to use the online library (and you may need to configure these).

Viewing the online documentation

To get to the online library:

- On Windows and OS/2, select the **Documentation** icon
- On AIX and Solaris, run the **ctgdoc** script

and the library home page is displayed.

The online library allows you to link to:

- CICS Transaction Gateway and CICS Universal Clients books in PDF format.
- Programming reference documentation in HyperText Markup Language (HTML) files (provided for CICS Transaction Gateway only).
- README files.
- Sample configuration documents in PDF format.
- Translated books in PDF format. (You may find that not all books are translated for your language.)
- The CICS Web site.

Guidance information on using Acrobat Reader is also provided.

Updated versions of the books may be provided from time to time, check our Web site at:

`http://www.ibm.com/software/ts/cics/`

and follow the **Library** link.

Viewing PDF books

The PDF information provides powerful functions for:

- Navigating through the information. There are hypertext links within PDF documents, and to other PDF documents and Web pages.
- Searching for specific information.
- Printing all or part of PDF documents on a PostScript printer.

You can find out more about Acrobat Reader at the Adobe Web site:

`http://www.adobe.com/acrobat/`

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply in the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Trademarks and service marks

The following terms, used in this publication, are trademarks or service marks of IBM Corporation in the United States or other countries:

AIX	IBM	VisualAge
CICS	OS/2	

Microsoft, Windows, Windows NT and the Windows logo are trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Index

Special Characters

-DCICS_AIX 8
-DCICS_SOL 8
-lc 8
-lc_r 8
-lcclcp 8
(parameter)
  in changed 46

Numerics

3270 datastreams 21

A

abendCode
  in CclException class 59
  in CclFlow class 68
  in Public methods 59, 68
Accessing fields on CICS 3270 screens
  in C++ External presentation interface 24
  in Using the CICS client C++ classes 24
active
  in state 57
  in State 57
activeFlow
  in CclConn class 45
  in CclFlow class 67
  in CclUOW class 93
activeUOW
  in CclConn class 45
  in CclUOW class 93
AID
  in CclScreen class 77
  in Enumerations 77
AIX
  in Environments supported 5
alphanumeric
  in BaseType 65
  in inputType 62
alterSecurity
  in CclConn class 46
  in CclTerminal class 86
  in Public methods 46, 86
alterSecurity (parameter)
  in alterSecurity 46
appendText
  in CclField class 61
  in Public methods 61
assign
  in CclBuf class 41
  in Public methods 41
async
  in CclSession constructor 83
  in Sync 37
Async Exception Handling 9
Asynchronous call synchronisation
  in ATI 22
Asynchronous reply handling
  in C++ External call interface 15
  in Controlling server interactions 15
ATI 22
ATIState
  in CclTerminal class 91
  in Enumerations 91
attachTran (parameter)
  in CclConn constructor 45, 46
attribute (parameter)
  in setBaseAttribute 63
  in setExtAttribute 64
Automatic Transaction Initiation 22
available
  in ServerStatus 50

B

backgroundColor
  in CclField class 61
  in Public methods 61
backout
  in CallType 70
  in CclUOW class 93
  in Managing logical units of work 19
  in Public methods 93
  in Server connection 12
baseAttribute
  in CclField class 61
  in Public methods 61
BaseInts
  in CclField class 65
  in Enumerations 65
BaseMDT
  in CclField class 65
  in Enumerations 65
BaseProt
  in CclField class 65
  in Enumerations 65
BaseType
  in CclField class 65
  in Enumerations 65
Basic Mapping Support
  in C++ External presentation interface 20
  in Glossary 103
black
  in Color 65
blinkHlt
  in Highlight 65
blue
  in Color 65
BMS
  map source files 21
  utility 21
books 105
  CICS Transaction Gateway and CICS Universal Clients library 105
  online 107
  PDF 108
  printed 108
Bool
  in Ccl class 37
  in Enumerations 37
buffer (parameter)
  in CclBuf 40
  in operator= 42
  in operator!= 43
  in operator+= 43
  in operator== 43

C

C++ External call interface
  Asynchronous reply handling 15
  Controlling server interactions 13
  Deferred synchronous reply handling 16
  Finding potential servers 11
  Managing logical units of work 18
  Monitoring server availability 17
  Passing data to a server program 12
  Server connection 12
  Synchronous reply handling 14

C++ External presentation interface
Accessing fields on CICS 3270 screens 24
EPI BMS conversion utility 28
EPI call synchronization types 25
in Using the CICS client C++ classes 20
Mapset containing a single map 29
Starting a 3270 terminal connection to CICS 23
Using EPI BMS Map Classes 30
C++ Programming
in Bibliography 105
callType
in CclFlow class 68
in Public methods 68
CallType
in CclFlow class 70
in Enumerations 70
callTypeText
in CclFlow class 68
in Public methods 68
cancel
in CallType 70
in CclConn class 46
in Monitoring server availability 17
in Public methods 46
in Server connection 12
Ccl::async
in EPI call synchronization types 25
Ccl::dsync
in EPI call synchronization types 27
Ccl::sync
in EPI call synchronization types 25
Ccl class
Bool 37
Sync 37
CclBuf
in CclBuf class 40
in CclBuf constructors 40
in Passing data to a server program 12
CclBuf class
assign 41
CclBuf 40
cut 41
dataArea 41
dataAreaLength 41
dataAreaOwner 41
CclBuf class *(continued)*
DataAreaOwner 44
dataAreaType 41
DataAreaType 44
dataLength 42
insert 42
listState 42
operator= 42
operator!= 43
operator+= 43
operator== 43
replace 43
setDataLength 44
CclBuf constructors
CclBuf 40
in CclBuf class 40
CclConn
in Monitoring server availability 17
in Server connection 12
CclConn class
alterSecurity 46
cancel 46
change password 46
changed 46
link 47
listState 47
makeSecurityDefault 48
password 48
serverName 48
serverStatus 49
ServerStatus 50
serverStatusText 49
status 48
userId 49
verifyPassword 49
CclConn constructor
in CclConn class 45
CCLCPOS2.LIB
in Using the CICS client C++ classes 7
CCLCPW32.LIB
in Using the CICS client C++ classes 7
CclECI
in Finding potential servers 11
CclECI class
exCode 51
exCodeText 51
handleException 52
instance 52
listState 52
serverCount 52
serverDesc 52
serverName 52
CclECI constructor (protected)
in CclECI class 51
CclEPI class
diagnose 55
exCode 55
exCodeText 56
handleException 56
serverCount 56
serverDesc 56
serverName 56
state 57
State 57
terminate 57
CclEPI constructor
in CclEPI class 55
CclException class
abendCode 59
className 59
diagnose 59
exCode 59
exCodeText 60
exObject 60
methodName 60
CclField
in Accessing fields on CICS 3270 screens 24
in C++ External presentation interface 22
CclField class
appendText 61
backgroundColor 61
baseAttribute 61
BaseInts 65
BaseMDT 65
BaseProt 65
BaseType 65
Color 65
column 62
dataTag 62
foregroundColor 62
highlight 62
Highlight 65
inputProt 62
inputType 62
intensity 63
length 63
position 63
resetDataTag 63
row 63
setBaseAttribute 63
setExtAttribute 63
setText 64
text 64
textLength 64
transparency 64

CclField class *(continued)*
Transparency 65
CclFlow
in CclFlow class 67
in CclFlow constructor 67
in Controlling server interactions 13
CclFlow class
abendCode 68
callType 68
CallType 70
callTypeText 68
CclFlow 67
connection 68
diagnose 68
flowId 68
forceReset 68
handleReply 69
listState 69
poll 69
setTimeout 69
syncType 70
timeout 70
uow 70
wait 70
CclFlow constructor
CclFlow 67
in CclFlow class 67
CCLICW32.LIB
in Using the CICS client C++ classes 7
CclMap
in C++ External presentation interface 22
in EPI BMS conversion utility 28
in Mapset containing a single map 29
CclMap class
exCode 71
exCodeText 71
field 72
namedField 72
validate 72
CclMap constructor
in CclMap class 71
CclScreen 23
in C++ External presentation interface 22
in EPI BMS conversion utility 29
CclScreen class
AID 77
cursorCol 75
cursorRow 75
CclScreen class *(continued)*
depth 75
field 75
fieldCount 76
mapName 76
mapSetName 76
setAID 76
setCursor 76
width 77
CclSecAttr 19, 31, 79
CclSession 23
in C++ External presentation interface 22
in EPI call synchronization types 25
CclSession::client
in EPI call synchronization types 26
CclSession::idle
in EPI call synchronization types 26
CclSession::server
in EPI call synchronization types 26
CclSession class
diagnose 83
handleReply 83
state 84
State 84
terminal 84
transID 84
CclSession constructor
in CclSession class 83
CclTerminal 22, 23
in C++ External presentation interface 22
CclTerminal class
alterSecurity 86
ATIState 91
CCSid 87
changePassword 86
diagnose 87
disconnect 87
discReason 87
EndTerminalReason 92
exCode 87
exCodeText 87
install 88
makeSecurityDefault 88
netName 88
password 88
poll 88
queryATI 89
readTimeout 89
receiveATI 89
CclTerminal class *(continued)*
screen 89
send 89, 90
serverName 91
setATI 90
signonCapability 90
signonType 92
state 91
State 92
termID 91
transID 91
userId 91
verifyPassword 91
CclTerminal constructor
in CclTerminal class 85
CclUOW
in Managing logical units of work 19
CclUOW class
backout 93
commit 93
forceReset 94
listState 94
uowId 94
CclUOW constructor
in CclUOW class 93
CCSid
in CclTerminal class 87
in Public methods 87
CCSid (parameter)
in CclTerminal constructor 85
change password
in CclConn class 46
in Public methods 46
changed
in CallType 70
in CclConn class 46
in Monitoring server availability 17
in Public methods 46
in Server connection 12
changePassword 19, 31
in CclTerminal class 86
in Public methods 86
CICS_AIX 8
CICS_OS2
in Using the CICS client C++ classes 7
CICS_SOL 8
CICS_W32
in Using the CICS client C++ classes 7
cicscli
in Starting a 3270 terminal connection to CICS 23

CICSECI.HPP
in Using the CICS client C++ classes 7
CICSEPI.HPP
in Using the CICS client C++ classes 7
className
in CclException class 59
in Public methods 59
clear
in AID 77
client
in EPI call synchronization types 26
in send 90
in State 84, 92
Client initialization file
in Finding potential servers 11
in serverCount 52, 56
in serverDesc 56
in serverName 56
in Starting a 3270 terminal connection to CICS 23
col
in validate 72, 73
col (parameter)
in setCursor 76
Color
in CclField class 65
in Enumerations 65
column
in CclField class 62
in Public methods 62
column (parameter)
in field 72, 75, 76
CommArea 11
commarea (parameter)
in handleReply 69
in link 47
in poll 69
commit
in CallType 70
in CclUOW class 93
in Managing logical units of work 19
in Public methods 93
in Server connection 12
communication, synchronous 14
connection
in CclFlow class 68
in Public methods 68
Connection object 19
Controlling server interactions
Asynchronous reply handling 15
Controlling server interactions *(continued)*
Deferred synchronous reply handling 16
in C++ External call interface 13
in Using the CICS client C++ classes 13
Synchronous reply handling 14
cursorCol
in CclScreen class 75
in Public methods 75
cursorRow
in CclScreen class 75
in Public methods 75
cut
in CclBuf class 41
in Public methods 41
cyan
in Color 65

D

dark
in BaseInts 65
in intensity 63
darkBlue
in Color 65
dataArea
in CclBuf class 41
in Public methods 41
dataArea (parameter)
in assign 41
in CclBuf 40
in insert 42
in replace 43
dataAreaLength
in CclBuf class 41
in Public methods 41
dataAreaOwner
in CclBuf class 41
in Public methods 41
DataAreaOwner
in CclBuf class 44
in Enumerations 44
dataAreaType
in CclBuf class 41
in Public methods 41
DataAreaType
in CclBuf class 44
in Enumerations 44
dataLength
in CclBuf class 42
in link 47
in Public methods 42
dataStream
in CclScreen class 75
dataTag
in CclField class 62
in Public methods 62
DBCS 21
default exception handler 10
defaultColor
in Color 65
defaultHlt
in Highlight 65
defaultTran
in Transparency 65
Deferred synchronous call synchronisation
in ATI 23
Deferred synchronous reply handling
in C++ External call interface 16
in Controlling server interactions 16
depth
in CclScreen class 75
in Public methods 75
in validate 72
devtype (parameter)
in CclTerminal constructor 85
diagnose
in CclEPI class 55
in CclException class 59
in CclFlow class 68
in CclSession class 83
in CclTerminal class 87
in Public methods 55, 59, 68, 83, 87
disabled
in ATIState 91
in queryATI 89
in setATI 90
discon
in state 57
in State 57, 84, 92
disconnect
in CclTerminal class 87
in EPI call synchronization types 26
in Public methods 87
discReason
in CclTerminal class 87
in Public methods 87
documentation 105
HTML 107
PDF 108
dsync
in CclSession constructor 83
in Sync 37

E

enabled
in ATIState 91
in queryATI 89
in setATI 90
EndTerminalReason
in CclTerminal class 92
in Enumerations 92
enter
in AID 77
Enumerations
AID 77
ATIState 91
BaseInts 65
BaseMDT 65
BaseProt 65
BaseType 65
Bool 37
CallType 70
Color 65
DataAreaOwner 44
DataAreaType 44
EndTerminalReason 92
Highlight 65
in Ccl class 37
in CclBuf class 44
in CclConn class 50
in CclEPI class 57
in CclField class 65
in CclFlow class 70
in CclScreen class 77
in CclSession class 84
in CclTerminal class 91
ServerStatus 50
signonType 92
State 57, 84, 92
Sync 37
Transparency 65
Environments supported
in Establishing the working environment 5
EPI BMS conversion utility
in C++ External presentation interface 28
in Using the CICS client C++ classes 28
Mapset containing a single map 29
EPI call synchronization types
in C++ External presentation interface 25
in Using the CICS client C++ classes 25
error
in state 57
error *(continued)*
in State 57, 84, 92
except (parameter)
in handleException 52, 56
exception handler, default 10
Exception Handling, Async 9
exceptions, handling 9
exCode
in CclECI class 51
in CclEPI class 55
in CclException class 59
in CclMap class 71
in CclTerminal class 87
in Public methods 51, 55, 59, 71, 87
exCodeText
in CclECI class 51
in CclEPI class 56
in CclException class 60
in CclMap class 71
in CclTerminal class 87
in Public methods 51, 56, 60, 71, 87
exObject
in CclException class 60
in Public methods 60
Expiry Management, Password 19, 31
extensible
in CclBuf 40
in CclBuf class 39
in dataAreaType 42
in DataAreaType 44
in setDataLength 44
external
in dataAreaOwner 41
in DataAreaOwner 44
External call interface
in Using the CICS client C++ classes 10

F

failed
in EndTerminalReason 92
field
in CclMap class 72
in CclScreen class 75
in Mapset containing a single map 29
in Public methods 72, 75
field()
in Mapset containing a single map 29
fieldCount
in CclScreen class 76
fieldCount *(continued)*
in Public methods 76
fields
in validate 72
fields (parameter)
in validate 72, 73
Finding potential servers
in C++ External call interface 11
in Using the CICS client C++ classes 11
fixed
in CclBuf 40
in CclBuf class 39
in dataAreaType 42
in DataAreaType 44
flow (parameter)
in backout 93
in cancel 46
in changed 46
in commit 93
in link 47
in status 48, 49
flowId
in CclFlow class 68
in Public methods 68
forceReset
in CclFlow class 68
in CclUOW class 94
in Public methods 68, 94
foregroundColor
in CclField class 62
in Public methods 62

G

gray
in Color 65
green
in Color 65

H

handleException 9
in CclECI class 52
in CclEPI class 56
in Public methods 52, 56
handleReply 23
in CclFlow class 69
in CclSession class 83
in EPI call synchronization types 25, 26, 27
in Public methods 69, 83
in Using EPI BMS Map Classes 31
handling exceptions 9
hardcopy books 108

highlight
in CclField class 62
in Public methods 62
Highlight
in CclField class 65
in Enumerations 65
HTML (HyperText Markup Language) 107
HTML documentation, viewing 107
HyperText Markup Language (HTML) 107

I

IBM VisualAge C++ 7
idle
in EPI call synchronization types 26
in send 90
in State 84, 92
inactive
in CallType 70
index (parameter)
in field 72, 75
in namedField 72
in serverDesc 52, 56
in serverName 52, 53, 56
in validate 72, 73
initEPI
in CclEPI constructor 55
Initiation, Automatic Transaction 22
inputProt
in CclField class 62
in Public methods 62
inputType
in CclField class 62
in Public methods 62
insert
in CclBuf class 42
in Public methods 42
install
in CclTerminal class 88
in Public methods 88
instance
in CclECI class 52
in Finding potential servers 11
in Public methods 52
intense
in BaseInts 65
in intensity 63
intenseHlt
in Highlight 65
intensity
in CclField class 63
in Public methods 63
internal
in CclBuf 40
internal *(continued)*
in dataAreaOwner 41
in DataAreaOwner 44
invalidMap
in CclMap constructor 71
invalidState
in poll 89
in send 90

K

key (parameter)
in setAID 76

L

labels
in validate 73
len
in validate 73
length
in CclField class 63
in Public methods 63
length (parameter)
in appendText 61
in assign 41
in CclBuf 40
in cut 41
in insert 42
in replace 43
in setDataLength 44
in setText 64
libcclcp.a 8
link
in CallType 70
in CclConn class 47
in Public methods 47
in Server connection 12
listState
in CclBuf class 42
in CclConn class 47
in CclECI class 52
in CclFlow class 69
in CclUOW class 94
in Public methods 42, 47, 52, 69, 94

M

makeSecurityDefault
in CclConn class 48
in CclTerminal class 88
in Public methods 48, 88
Management, Password Expiry 19, 31
Managing logical units of work
in C++ External call interface 18
in Using the CICS client C++ classes 18
map (parameter)
in validate 72
MAPINQ1Map
in Mapset containing a single map 29
mapName
in CclScreen class 76
in Public methods 76
Mapset containing a single map
in C++ External presentation interface 29
in EPI BMS conversion utility 29
mapSetName
in CclScreen class 76
in Public methods 76
MaxBufferSize (parameter)
in CclBuf class 39
maxServers
in serverDesc 56
in serverName 56
methodName
in CclException class 60
in Public methods 60
Microsoft Visual C++ 7
modified
in BaseMDT 65
in dataTag 62
Monitoring server availability
in C++ External call interface 17
in Using the CICS client C++ classes 17
multi-threading 8
multipleInstance
in CclECI class 51

N

n (parameter)
in position 63
namedField
in CclMap class 72
in Protected methods 72
netName
in CclTerminal class 88
in Public methods 88
netname (parameter)
in CclTerminal constructor 85
neutral
in Color 65
newPassword (parameter)
in alterSecurity 46
in changed 47
in changePassword method 86
newstate (parameter)
in setATI 90

newUserid (parameter)
in alterSecurity 46
no
in Bool 37
in operator!= 43
in operator== 43
in poll 69
normal
in BaseInts 65
in intensity 63
normalHlt
in Highlight 65
notDiscon
in EndTerminalReason 92
numeric
in BaseType 65
in inputType 62

O

off
in Bool 37
offset (parameter)
in cut 41
in dataArea 41
in insert 42
in replace 43
on
in Bool 37
online books, PDF 108
online documenatation, HTML 107
OO support in CICS Clients
in Introduction to OO programming 3
opaqueTran
in Transparency 65
operator=
in CclBuf class 42
in Public methods 42
operator!=
in CclBuf class 43
in Public methods 43
operator+=
in CclBuf class 43
in Public methods 43
operator==
in CclBuf class 43
in Public methods 43
orange
in Color 65
orTran
in Transparency 65
OS/2
in Environments supported 5
outofService
in EndTerminalReason 92

P

PA1
in AID 77
PA3
in AID 77
paleCyan
in Color 65
paleGreen
in Color 65
parameter
in CclMap constructor 71
in send 90
in setCursor 77
in setExtAttribute 64
Passing data to a server program
in C++ External call interface 12
in Using the CICS client C++ classes 12
password 19, 47
in CclConn class 48
in CclTerminal class 88
in Public methods 48, 88
in verifyPassword method 49
password (parameter)
in alterSecurity method 86
in CclConn constructor 45
in CclTerminal constructor 85
Password Expiry Management 19, 31
PDF (Portable Document Format) 108
PDF books, viewing 108
PF1
in AID 77
PF24
in AID 77
PF3
in Accessing fields on CICS 3270 screens 24
pink
in Color 65
poll
in CclFlow class 69
in CclTerminal class 88
in Deferred synchronous reply handling 16, 17
in EPI call synchronization types 27
in Public methods 69, 88
poll method 23
Portable Document Format (PDF) 108
position
in CclField class 63
in Public methods 63
PostScript books 108
Programming language support
in Introduction to OO programming 4
programName (parameter)
in link 47
protect
in BaseProt 65
in inputProt 62
Protected methods
in CclMap class 72
namedField 72
validate 72
Public methods
abendCode 59, 68
alterSecurity 46, 86
appendText 61
assign 41
backgroundColor 61
backout 93
baseAttribute 61
callType 68
callTypeText 68
cancel 46
CCSid 87
change password 46
changed 46
changePassword 86
className 59
column 62
commit 93
connection 68
cursorCol 75
cursorRow 75
cut 41
dataArea 41
dataAreaLength 41
dataAreaOwner 41
dataAreaType 41
dataLength 42
dataTag 62
depth 75
diagnose 55, 59, 68, 83, 87
disconnect 87
discReason 87
exCode 51, 55, 59, 71, 87
exCodeText 51, 56, 60, 71, 87
exObject 60
field 72, 75
fieldCount 76
flowId 68
forceReset 68, 94
foregroundColor 62
handleException 52, 56
handleReply 69, 83

Public methods *(continued)*
highlight 62
in CclBuf class 41
in CclConn class 46
in CclECI class 51
in CclEPI class 55
in CclException class 59
in CclField class 61
in CclFlow class 68
in CclMap class 71
in CclScreen class 75
in CclSession class 83
in CclTerminal class 86
in CclUOW class 93
inputProt 62
inputType 62
insert 42
install 88
instance 52
intensity 63
length 63
link 47
listState 42, 47, 52, 69, 94
makeSecurityDefault 48, 88
mapName 76
mapSetName 76
methodName 60
netName 88
operator= 42
operator!= 43
operator+= 43
operator== 43
password 48, 88
poll 69, 88
position 63
queryATI 89
readTimeout 89
receiveATI 89
replace 43
resetDataTag 63
row 63
screen 89
send 89, 90
serverCount 52, 56
serverDesc 52, 56
serverName 48, 52, 56, 91
serverStatus 49
serverStatusText 49
setAID 76
setATI 90
setBaseAttribute 63
setCursor 76
setDataLength 44
setExtAttribute 63
setText 64
Public methods *(continued)*
setTimeout 69
signonCapability 90
state 57, 84, 91
status 48
syncType 70
termID 91
terminal 84
terminate 57
text 64
textLength 64
timeout 70
transID 84, 91
transparency 64
uow 70
uowId 94
userId 49, 91
verifyPassword 49, 91
wait 70
width 77
publications, CICS Transaction Gateway and CICS Universal Clients library 105
purple
in Color 65

Q

queryATI 22
in CclTerminal class 89
in Public methods 89

R

readTimeout
in CclTerminal class 89
in Public methods 89
readTimeOut (parameter)
in CclTerminal constructor 85
receiveATI 23
in CclTerminal class 89
in Public methods 89
red
in Color 65
replace
in CclBuf class 43
in Public methods 43
reply handling, asynchronous 15
resetDataTag
in CclField class 63
in Public methods 63
reverseHlt
in Highlight 65
row
in CclField class 63
in Public methods 63
in validate 72, 73
row (parameter)
in field 72, 75, 76
in setCursor 76
runTran (parameter)
in CclConn constructor 45

S

screen
in Accessing fields on CICS 3270 screens 24
in CclTerminal class 89
in Public methods 89
screen (parameter)
in CclMap constructor 71
in handleReply 83
Security Management
ECI 19
EPI 31
send
in CclTerminal class 89, 90
in EPI call synchronization types 25, 26, 27
in Public methods 89, 90
in Starting a 3270 terminal connection to CICS 23
send method
in ATI 22
server
in EPI call synchronization types 26, 27
in poll 89
in State 84, 92
server (parameter)
in CclTerminal constructor 85
Server connection
in C++ External call interface 12
in Using the CICS client C++ classes 12
serverCount
in CclECI class 52
in CclEPI class 56
in Finding potential servers 11
in Public methods 52, 56
serverDesc
in CclECI class 52
in CclEPI class 56
in Finding potential servers 11
in Public methods 52, 56
serverName
in CclConn class 48
in CclECI class 52
in CclEPI class 56
in CclTerminal class 91
in Finding potential servers 11
in Public methods 48, 52, 56, 91

serverName (parameter)
in CclConn constructor 45
serverStatus
in CclConn class 49
in Public methods 49
ServerStatus
in CclConn class 50
in Enumerations 50
serverStatusText
in CclConn class 49
in Public methods 49
session (parameter)
in receiveATI method 89
in send 89, 90
setAID
in CclScreen class 76
in Public methods 76
setATI 22
in CclTerminal class 90
in Public methods 90
setBaseAttribute
in CclField class 63
in Public methods 63
setCursor
in CclScreen class 76
in Public methods 76
setDataLength
in CclBuf class 44
in Public methods 44
setExtAttribute
in CclField class 63
in Public methods 63
setText
in CclField class 64
in Public methods 64
setTimeout
in CclFlow class 69
in Public methods 69
shutdown
in EndTerminalReason 92
signoff
in EndTerminalReason 92
signonCapability
in CclTerminal class 90
in Public methods 90
signonCapability (parameter)
in CclTerminal constructor 85
signonCapable
in signonType 92
signonIncapable
in signonType 92
signonType
in CclTerminal class 92
in Enumerations 92
signonUnknown
in signonType 92
softcopy books, PDF 108
Solaris 8
in Environments supported 5
stackPages (parameter)
in CclFlow 67
startdata (parameter)
in send 89, 90
Starting a 3270 terminal connection to CICS
in C++ External presentation interface 23
in Using the CICS client C++ classes 23
state
in CclEPI class 57
in CclSession class 84
in CclTerminal class 91
in EPI call synchronization types 27
in Public methods 57, 84, 91
State
in CclEPI class 57
in CclSession class 84
in CclTerminal class 92
in Enumerations 57, 84, 92
state (parameter)
in EPI call synchronization types 26
in handleReply 83
status
in CallType 70
in CclConn class 48
in Monitoring server availability 17
in Public methods 48
in Server connection 12
Sun Workshop 8
sync
in CclSession constructor 83
in Sync 37
Sync
in Ccl class 37
in Enumerations 37
synchronous, Deferred 23
Synchronous call synchronisation
in ATI 22
Synchronous reply handling
in C++ External call interface 14
in Controlling server interactions 14
syncType
in CclFlow class 70
in poll 69, 89
syncType *(continued)*
in Public methods 70
in wait 70
syncType (parameter)
in CclFlow 67
in CclSession constructor 83

T

termDefined
in State 92
termID
in CclTerminal class 91
in Public methods 91
terminal
in CclSession class 84
in Public methods 84
terminate
in CclEPI class 57
in Public methods 57
text
in CclField class 64
in Public methods 64
text (parameter)
in appendText 61
in CclBuf 40
in operator= 42
in operator+= 43
in setText 64
textLength
in CclField class 64
in Public methods 64
threads, multiple 8
timeout
in CclFlow class 70
in Public methods 70
timeout (parameter)
in CclFlow 67
in setTimeout 69
Trademarks and service marks
in Notices 110
Transaction Initiation, Automatic 22
transID 23
in CclSession class 84
in CclTerminal class 91
in Public methods 84, 91
transid (parameter)
in send 89, 90
transparency
in CclField class 64
in Public methods 64
Transparency
in CclField class 65
in Enumerations 65
type (parameter)
in CclBuf 40

U

unavailable
in ServerStatus 50
underscoreHlt
in Highlight 65
unit (parameter)
in link 47
unknown
in EndTerminalReason 92
in ServerStatus 50
unmodified
in BaseMDT 65
in dataTag 62
unmodified (parameter)
in resetDataTag 63
unpadded (parameter)
in status 48, 49
unprotect
in BaseProt 65
in inputProt 62
uow
in CclFlow class 70
in Public methods 70
uowId 19
in CclUOW class 94
in Public methods 94
userId
in CclConn class 49
in CclTerminal class 91
in Public methods 49, 91
userid (parameter)
in alterSecurity method 86
in CclTerminal constructor 85
userId (parameter)
in CclConn constructor 45
userID (parameter)
in CclConn constructor 45
Using EPI BMS Map Classes
in C++ External presentation interface 30
in Using the CICS client C++ classes 30
Using the CICS client C++ classes
Accessing fields on CICS 3270 screens 24
Controlling server interactions 13
EPI BMS conversion utility 28
EPI call synchronization types 25
Finding potential servers 11
Managing logical units of work 18
Monitoring server availability 17
Using the CICS client C++ classes *(continued)*
Passing data to a server program 12
Server connection 12
Starting a 3270 terminal connection to CICS 23
Using EPI BMS Map Classes 30

V

validate
in CclMap class 72
in Protected methods 72
value (parameter)
in setExtAttribute 64
verifyPassword 19, 31
in CclConn class 49
in CclTerminal class 91
in Public methods 49, 91
viewing online documentation 107

W

wait
in Asynchronous reply handling 16
in CclFlow class 70
in Deferred synchronous reply handling 17
in Public methods 70
white
in Color 65
width
in CclScreen class 77
in Public methods 77
in validate 72
Windows 98
in Environments supported 5
Windows NT
in Environments supported 5
Workshop, Sun 8

X

xlC_r 8
xorTran
in Transparency 65

Y

yellow
in Color 65
yes
in Bool 37
in operator!= 43
in operator== 43
in poll 69

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC33-1923-01