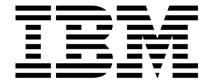
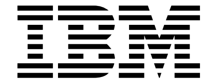
CICS Family



OO Programming in C++ for CICS Clients

SC33-1923-00

CICS Family



OO Programming in C++ for CICS Clients

SC33-1923-00

Note!

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

First edition (March 1997)

This edition applies to CICS Clients Version 2.0.1, program number 5639-001, and to all subsequent versions, releases, and modifications until otherwise indicated in new editions. Consult the latest edition of the applicable IBM system bibliography for current information on this product.

The previous book dealing with this subject, *Object Oriented Programming for CICS Clients* SC33-1639, has been split into two books. This book contains the C++ programming topics. *OO Programming in BASIC for CICS Clients* SC33-1924 contains the BASIC programming topics.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

At the back of this publication is a page entitled “Sending your comments to IBM”. If you want to make comments, but the methods described are not available to you, please address them to:

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

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1996,1997. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices . . . v
Trademarks and service marks . . . vi

Preface . . . vii
  Who this book is for . . . vii
  What this book is about . . . vii
  What you need to know before reading this book . . . vii
  Determining if a publication is current . . . vii
  Keeping up-to-date through the Internet . . . viii

Bibliography . . . ix

Part 1. Client Classes—Guidance . . . 1

Chapter 1. Introduction to OO programming . . . 3

Chapter 2. Establishing the working environment . . . 5

Chapter 3. Using the CICS client C++ classes . . . 7

Part 2. CICS Client C++ classes - reference . . . 31

Chapter 4. Ccl class . . . 33

Chapter 5. CclBuf class . . . 35

Chapter 6. CclConn class . . . 45

Chapter 7. CclECI class . . . 51

Chapter 8. CclEPI class . . . 55

Chapter 9. CclException class . . . 61

Chapter 10. CclField class . . . 65

Chapter 11. CclFlow class . . . 73

Chapter 12. CclMap class . . . 79

Chapter 13. CclScreen class . . . 83

Chapter 14. CclSession class . . . 87

© Copyright IBM Corp. 1996,1997

Chapter 15. CclTerminal class 91

Chapter 16. CclUOW class 97

Glossary 101

Index 103

Notices

The following paragraph does not apply to any 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 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.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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 of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

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 Laboratory Counsel, MP151, IBM United Kingdom Laboratories, 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.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, New York 10594, U.S.A.

© Copyright IBM Corp. 1996,1997

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	BookManager	CICS	CSet++
IBM	MVS/ESA	OS/2	VisualAge

Microsoft, Windows, Windows NT, the Windows 95 Logo, Visual Basic, and Visual C++ are trademarks of Microsoft Corporation.

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

Preface

Application programmers can use the object oriented (OO) classes supplied with CICS Clients Version 2.0.1 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 Programming 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.

Who this book is for

This book is for CICS application programmers who want to know how to use the OO classes provided in CICS Clients Version 2.0.1.

What this book is about

This book is divided into two parts.

Part 1 describes the programming environment and shows you, with examples, how to use the classes.

Part 2 contains the reference material.

Note: The previous book dealing with this subject,*Object Oriented Programming for CICS Clients* SC33-1639, has been split into two books. This book contains the C++ programming topics. *OO Programming in BASIC for CICS Clients* SC33-1924 contains the BASIC programming topics.

What you need to know before reading this book

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.

Determining if a publication is current

IBM regularly updates its publications with new and changed information. When first published, both hardcopy and BookManager softcopy versions of a publication are in step, but subsequent updates will probably be available in softcopy before they are available in hardcopy.

For CICS Transaction Server books, these softcopy updates appear regularly on the *Transaction Processing and Data Collection Kit* CD-ROM, SK2T-0730-xx. Each reissue of the collection kit is indicated by an updated order number suffix (the -xx part). For

© Copyright IBM Corp. 1996,1997

example, collection kit SK2T-0730-06 is more up-to-date than SK2T-0730-05. The collection kit is also clearly dated on the cover.

Here's how to determine if you are looking at the most current copy of a publication:

- A publication with a higher suffix number is more recent than one with a lower suffix number. For example, the publication with order number SC33-0667-02 is more recent than the publication with order number SC33-0667-01. (Note that suffix numbers are updated as a product moves from release to release, as well as for hardcopy updates within a given release.)
- When the softcopy version of a publication is updated for a new collection kit the order number it shares with the hardcopy version does not change. Also, the date in the edition notice remains that of the original publication. To compare softcopy with hardcopy, and softcopy with softcopy (on two editions of the collection kit, for example), check the last two characters of the publication's filename. The higher the number, the more recent the publication. For example, DFHPF104 is more recent than DFHPF103. Next to the publication titles in the CD-ROM booklet and the readme files, asterisks indicate publications that are new or changed.
- Updates to the softcopy are clearly marked by revision codes (usually a "#" character) to the left of the changes.

Keeping up-to-date through the Internet

IBM CICS Development maintain a World-Wide-Web (WWW) home page for all members of the CICS family of products. These pages are publicly available to anyone with internet access at the following URL:

CICS Home Page URL

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

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.

CICS family: The following books are published by IBM:

- *CICS Family: General Information*, GC33-0155
- *CICS Family: Client/Server Programming*, SC33-1435
- *CICS Clients Administration*, SC33-1436
- *ITSC CICS Clients Unmasked*, GG24-2534

CICS for MVS/ESA: The following book is published by IBM:

- *CICS for MVS/ESA Application Programming Guide*, SC33-1169

CICS on Open Systems: The following book is published by IBM:

- *CICS on Open Systems Application Programming Guide*, SC33-1568

CICS for OS/2: The following book is published by IBM:

- *CICS for OS/2 Application Programming*, SC33-1585

© Copyright IBM Corp. 1996,1997

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
Installation 6

Chapter 3. Using the CICS client C++ classes 7
C++ External call interface 8
C++ External presentation interface 18

© Copyright IBM Corp. 1996,1997

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.

© Copyright IBM Corp. 1996,1997

Programming language support

OO libraries are provided with CICS Clients Version 2.0.1 for C++ and BASIC programmers. This book covers the C++ programming topics. If you wish to program using BASIC, then please refer to *OO Programming in BASIC for CICS Clients* SC33-1924 where you will find a user guide based on the BASIC samples and a description of the BASIC interfaces.

Chapter 2. Establishing the working environment

You are provided with C++ (OO) support for CICS clients in OS/2 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 2.1 or later
IBM Visual Age C++ for OS/2 Version 3.0 or later

Windows

Microsoft Windows Version 3.11
Microsoft Visual C++ Version 1.51 or later

Windows NT

Windows NT Workstation Version 3.51
IBM Visual Age C++ for Windows Version 3.5
Microsoft Visual C++ Version 4.0

Windows 95

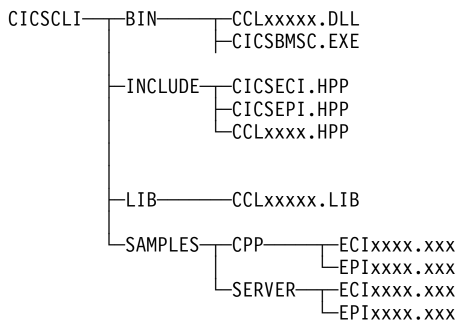
Windows 95
IBM Visual Age C++ for Windows Version 3.5
Microsoft Visual C++ Version 4.0

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

© Copyright IBM Corp. 1996,1997

Installation

OO support files for various programming languages are installed with the CICS Client. After installation, your directories will contain the following C++ files:



Sample programs

We have supplied sample programs for each compiler as follows:

ECICPO*n.ext* **and EPICPO***n.ext*
: for VisualAge on OS/2

ECICPN*n.ext* **and EPICPN***n.ext*
: for Visual C++ on Windows NT and Windows 95

ECICPW*n.ext* **and EPICPW***n.ext*
: for Visual C++ on Windows 3.11

ECIVAN*n.ext* **and EPIVAN***n.ext*
: for VisualAge C++ on Windows NT and Windows 95

Chapter 3. Using the CICS client C++ classes

For examples of programs that use the CICS client C++ classes, refer to the samples and the README file supplied in \CICSCLI\SAMPLES\CPP.

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

- Compiling and linking on OS/2 using IBM VisualAge C++ for 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 \CICSCLI\LIB.

- Compiling and linking on Windows 3.11 using Microsoft Visual C++

  The preprocessor definition CICS_WIN must be defined to the compiler using the /DCICS_WIN option.

  To support C++ subclassing, compiler options must be set to generate Windows prolog/epilog code for protected mode application functions, with the additional option set to generate code for __far functions (equivalent to the /GA and /GEf compiler options). Note that these options are not compatible with Windows "QuickWin" applications.

  For compile-time type checking of window handles (the HWND type), CICS client classes require the use of the /DSTRICT compiler option. For MFC applications this option will already be set.

  The application must be linked with the CCLCPWIN.LIB library supplied in \CICSCLI\LIB.

- Compiling and linking on Windows NT and Windows 95 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 \CICSCLI\LIB.

- Compiling and linking on Windows NT and Windows 95 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)

© Copyright IBM Corp. 1996,1997

C++ External call interface

The application must be linked with the `CCLICW32.LIB` library supplied in `\CICSCLI\LIB`

C++ 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
UOW	**CclUOW**	Corresponds with a Unit of Work in the server—used for managing updates to recoverable resources.

Finding potential servers

Information about the CICS servers that can be used by a client program is defined in the CICS Client Initialization file, `CICSCLI.INI`. 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.

Handling exceptions

When a problem is detected by the ECI class implementation, the details are encapsulated in a newly created exception object—**CclException** which is immediately passed to the **handleException** method of the ECI object.

The ECI object should be subclassed to provide the implementation of an exception handler which will be automatically called whenever any problem is detected. The handler can take whatever action is appropriate. For example, it could display a diagnostic string in a popup window and/or throw the exception object to a C++ catch handler before the program continues. A default exception handler is provided which throws the exception object; except on Windows 3.1 where it just returns to the caller.

A wide range of problems can be successfully trapped with a single simple exception handler, for example:

- an invalid password rejected by a server
- the ECI client implementation losing contact with the server
- the truncation of a string assigned to an inextensible buffer object

Here is a simple subclass of the ECI class with an exception handler implementation which just prints text describing the exception:

```
class MyCclECI : public CclECI {
public:
  void handleException( CclException ex ) {
    cout << "Exception: " << ex.exCodeText() << endl;
    }
  };
```

If an instance of the ECI subclass is created, this new implementation of handleException will be executed whenever an exception is raised. In the following code, a connection object is created with a bad server name. When the link call requests the server program to be executed, the exception handler is called and the output text is printed.

```
MyCclECI myeci;
  ...
CclConn serbad( "BadName","user1","password" );
serbad.link( sflow,"PROGNAME",&comma2 );
  ...
```

Output produced:

```
Exception: unknown server
```

The ECI class is a single instance class. Only one object of the ECI class or an ECI subclass can exist.

The exception class has some useful methods which can be used to diagnose the problem being reported and determine its context; for example, the **diagnose** method returns a text string containing object state information which could be used in an exception handler.

Most of the ECI classes include a **listState** method which can be freely called to generate a trace of the state of selected objects to aid in problem diagnosis.

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:

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

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

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

Separate flow subclasses could be needed to cater for different client/server communication area protocols. Many flows may be active at the same time. Many servers may be used simultaneously by the same 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 code which follows is based on one of the sample programs, `ECIxxx1.CPP`. (See “Sample programs” on page 6 for the file appropriate to your compiler.) This 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 is based on the sample program, ECIxxx2.CPP. (See “Sample programs” on page 6 for the file appropriate to your compiler.) This 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.

On Windows 3.11, **CclFlow** provides an optional message notification mechanism to indicate when a reply has been received from the host. To use message notification, pass the optional parameters, *windowHandle* and *messageId*, when calling the

CclFlow class constructor. The application can then be designed to poll the flow for the host reply when message *messageId* is received.

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. A more sophisticated example can be found in the sample program `ECIxxx4.CPP` which shows availability of servers in a listbox displayed and updated in real time as part of an OS/2 PM application implemented using IBM OpenClass. (See “Sample programs” on page 6 for the file appropriate to your compiler.)

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:

```
CclUOW uow;
```

Any server link request which participates in a unit of work references the corresponding unit of work object. When all the links participating in a unit of work have successfully completed, the unit of work can be committed by the **commit** method of the unit of work object or backed out by **backout**:

```
serv1.link( sflow, "ECITSQ", &( comma1="1st link in UOW" ), &uow );
serv1.link( sflow, "ECITSQ", &( comma1="2nd link in UOW" ), &uow );
  ...
uow.backout( sflow );
```

If no UOW object is used, each link call becomes a complete unit of work (equivalent to LINK SYNCONRETURN in the CIS server).

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.

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.
Session	**CclSession**	Controls communication with the server in synchronous, asynchronous and deferred synchronous modes. Applications can use **CclSession** to derive their own classes to encapsulate specific CICS transactions.
Terminal	**CclTerminal**	Controls a 3270 terminal connection to CICS. The **CclTerminal** class handles CICS conversational, pseudo-conversational, and ATI transactions. One application can create many **CclTerminal** objects.

C++ External presentation interface

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, CICSCLI.INI. 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 CICSCLI.INI
for ( int i=1; i<= EPI.serverCount(); i++ )
    cout << EPI.serverName(i) << "  "
         << EPI.serverDesc(i) << endl;
```

In the Windows 3.1 environment additional considerations apply when initializing the EPI; see “Using EPI under Windows 3.1” on page 29.

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 CICSCLI.INI 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.

In the Windows 3.1 environment additional considerations apply when connecting a terminal to CICS, see “Using EPI under Windows 3.1” on page 29.

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.”

C++ External presentation interface

EPI call synchronization types

The CICS client EPI C++ classes in OS/2, Windows 95 and Windows NT environments support synchronous ("blocking"), and deferred synchronous ("polling") and asynchronous ("callback") protocols. In the Windows 3.1 environment the EPI C++ classes only support the deferred synchronous protocol, and there are some additional considerations when initializing the EPI and when connecting a 3270 terminal. See "Using EPI under Windows 3.1" on page 29.

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:

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

The implementation of the **handleReply** method can process the screen data available in the CclScreen object, which will have been updated in line with the 3270 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.

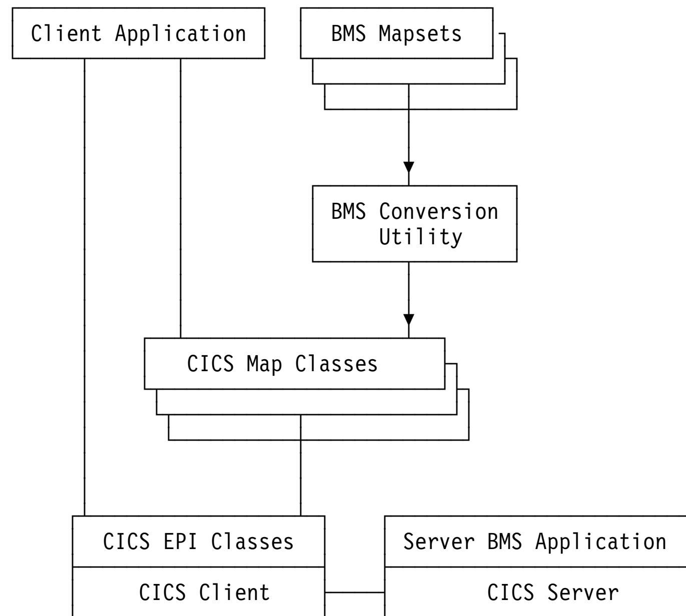


Figure 1. Use of BMS map classes

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 on page 26 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 27) 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

C++ External presentation interface

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 3.11, Windows NT and Windows 95 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.

Using EPI under Windows 3.1

In the Windows 3.1 environment, the CICS client EPI C++ classes only support deferred synchronous calls. The **send** methods on the CclTerminal object therefore have to be followed by **poll** method calls to receive the CICS replies. An additional message notification mechanism is provided to indicate when the application should poll for a reply.

In addition, EPI initialization and terminal connection are asynchronous. On creation of CclEPI and CclTerminal object, control is returned to the client application immediately, with message notification provided to indicate when the operation is complete.

To support the message notification mechanism, in the Windows 3.1 environment only, the CclEPI constructor has two parameters:

Type	Parameter	Description
HWND	hWnd	Window handle of the window to which CICS client messages are to be sent
UINT	message	Message id for messages sent by the CICS client. This parameter defaults to a supplied value CICS_NOTIFY.

The client application will need to handle the CICS client messages, for example by defining a message handler in the applications **view** class:

```
#include <cicsepi.hpp> // CclClass definitions

class EpiView : public CFormView {
...
// Message map functions
protected:
    afx_msg LRESULT OnNotify(WPARAM wParam,LPARAM lParam);
};
```

C++ External presentation interface

The implementation of the view class contains code handling the messages received on EPI initialization, terminal connection and on each 3270 reply from CICS:

```
BEGIN_MESSAGE_MAP(EpiView, CFormView)
    ...
    ON_MESSAGE(CICS_NOTIFY, OnNotify)
END_MESSAGE_MAP()
...
// CICS Client EPI Class Message Handler
LRESULT EpiView::OnNotify(WPARAM wParam,LPARAM lParam) {
    switch((CclEPI::Message)wParam) {
    case CclEPI::init:
        // CICS EPI initialized
        ....
    case CclEPI::connect:
        // CICS EPI terminal connected
        ....
    case CclEPI::reply:
        // Reply waiting, poll the CclTerminal object
        terminal->poll();
        ....
    } // end switch (message)
    return(0);
}
```

The main application program supplies the window handle of the view class on the CclEPI constructor. After creating an CclEPI object, creating a CclTerminal object or invoking the CclTerminal **send** method, the application then waits for the **OnNotify** function to receive the CICS_NOTIFY message.

Note that, depending on the CICS server program, more that one reply message may be received for a single send call. The client application will need to poll for each reply. The terminal state should be checked to find out whether the CICS server program has more data to send. This check can be done using the **state** method on the CclTerminal or CclSession object. Alternatively it can be made inside the CclSession **handleReply** method if a CclSession subclass is used.

Part 2. CICS Client C++ classes - reference

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 . . . 33
Enumerations . . . 33

Chapter 5. CclBuf class . . . 35
CclBuf constructors . . . 36
Public methods . . . 37
Enumerations . . . 43

Chapter 6. CclConn class . . . 45
CclConn constructor . . . 45
Public methods . . . 46
Enumerations . . . 50

Chapter 7. CclECI class . . . 51
CclECI constructor (protected) . . . 51
Public methods . . . 51

Chapter 8. CclEPI class . . . 55
CclEPI constructor . . . 55
Public methods . . . 56
Enumerations . . . 59

Chapter 9. CclException class . . . 61
Public methods . . . 61

Chapter 10. CclField class . . . 65
Public methods . . . 65
Enumerations . . . 70

Chapter 11. CclFlow class . . . 73
CclFlow constructor . . . 73
Public methods . . . 74
Enumerations . . . 78

Chapter 12. CclMap class . . . 79
CclMap constructor . . . 79
Public methods . . . 79
Protected methods . . . 81

Chapter 13. CclScreen class . . . 83
Public methods . . . 83

© Copyright IBM Corp. 1996,1997

Enumerations 85

Chapter 14. CclSession class 87
CclSession constructor 87
Public methods 87
Enumerations 89

Chapter 15. CclTerminal class 91
CclTerminal constructor 91
Public methods 92
Enumerations 96

Chapter 16. CclUOW class 97
CclUOW constructor 97
Public methods 97

Chapter 4. Ccl class

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

Enumerations

Bool

There are three equivalent pairs of values:

`no` and `yes`
`off` and `on`
`false` and `true`

Sync

Possible values are:

`async`	asynchronous (not Windows 3.1)
`dsync`	deferred synchronous
`sync`	synchronous (not Windows 3.1 EPI applications)

ExCode

Possible values are:

`noError`	currently, there is no error to report
`bufferOverflow`	a fixed-length Buffer object overflowed
`multipleInstance`	attempted to construct more than one CclECI or CclEPI object or subclass instance
`activeFlow`	attempted an invalid operation on an active CclFlow object
`activeUOW`	attempted an invalid operation on an active CclUOW object
`syncType`	a CclFlow or CclSession synchronization type was used incorrectly
`threadCreate`	an error occurred during thread creation
`threadWait`	an error occurred while waiting for a thread to terminate
`threadKill`	an error occurred while attempting to terminate a thread
`dataLength`	the length of a commarea buffer or a 3270 transmission was out of range
`noCICS`	the CICS client or server was not available
`CICSDied`	the CICS server became unavailable while a UOW was active
`noReply`	no reply was available from the server
`transaction`	the CICS transaction that executed the requested program abended. See **CclFlow** method “abendCode” on page 74
`systemError`	an internal CICS client system error occurred. See **CclFlow** method “listState” on page 76
`resource`	the client or server had insufficient resources to complete the request

© Copyright IBM Corp. 1996,1997

maxUOWs	exceeded the maximum number of UOWs the configuration will support
unknownServer	could not locate the requested server
security	did not supply a valid userId/password for the server
maxServers	exceeded the maximum number of servers the configuration will support
maxRequests	exceeded the maximum number of simultaneous server requests the configuration will support
rolledBack	the server could not commit the changes in a UOW, so backed them out instead
parameter	input parameter invalid
invalidState	the object is in an invalid state for the requested operation
transId	invalid transaction identifier
initEPI	EPI not initialized
connect	unable to connect to requested CICS server
dataStream	unsupported 3270 datastream received from CICS
invalidMap	BMS map object does not match current screen
cclClass	an internal error occurred in the Client Classes

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 (OS/2, Windows NT and Windows 95), or the maximum value for `unsigned short` (2^{16}) for 16 bit platforms (Windows 3.11). 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 X'00''s.

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
```

© Copyright IBM Corp. 1996,1997

CclBuf constructors

1.

CclBuf

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 X'00'. The data length is set to zero and will remain zero until data is put in the buffer.

2.

CclBuf ——— OS/2, Windows NT and Windows 95

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

CclBuf ——— Windows 3.11

CclBuf(unsigned long *length***, void __far*** *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.

3.

CclBuf ——— OS/2, Windows NT and Windows 95

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

CclBuf ——— Windows 3.11

CclBuf(const char __far* *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.

4.

CclBuf ——— OS/2, Windows NT and Windows 95

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

CclBuf ——— Windows 3.11

CclBuf(const CclBuf __far& *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

assign ——— OS/2, Windows NT and Windows 95

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

assign ——— Windows 3.11

CclBuf __far& assign(unsigned short *length***, const void __far*** *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

cut ——— OS/2, Windows NT and Windows 95

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

cut ——— Windows 3.11

CclBuf __far& cut(unsigned short *length***, unsigned short** *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 X'00's.

dataArea

dataArea — OS/2, Windows NT and Windows 95

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

dataArea — Windows 3.11

const void __far* dataArea(unsigned short *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

dataAreaLength — OS/2, Windows NT and Windows 95

unsigned long dataAreaLength() const

dataAreaLength — Windows 3.11

unsigned short dataAreaLength() const

Returns the length of the data area in bytes.

dataAreaOwner

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 dataAreaType() const

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

dataLength

dataLength ——— OS/2, Windows NT and Windows 95

unsigned long dataLength() const

dataLength ——— Windows 3.11

unsigned short dataLength() const

Returns the length of data in the data area. This cannot be greater than the value returned by **dataAreaLength**.

insert

insert ——— OS/2, Windows NT and Windows 95

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

insert ——— Windows 3.11

CclBuf __far& insert(unsigned short *length*,
const void __far* *dataArea*,
unsigned short *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

listState — OS/2, Windows NT and Windows 95

const char* listState() const

listState — Windows 3.11

const char __far* 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.

operator= — OS/2, Windows NT and Windows 95

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

operator= — Windows 3.11

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

buffer A reference to a CclBuf object.

Assigns data from another buffer object.

2.

operator= — OS/2, Windows NT and Windows 95

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

operator= — Windows 3.11

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

text The string to be assigned to the CclBuf object.

Assigns data from a string.

operator+=

1.

operator+= — OS/2, Windows NT and Windows 95

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

operator+= — Windows 3.11

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

buffer A reference to a CclBuf object.

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

2.

operator+= — OS/2, Windows NT and Windows 95

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

operator+= — Windows 3.11

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

text The string to be appended to the CclBuf object.

Appends a string to the data in the data area.

operator==

operator== — OS/2, Windows NT and Windows 95

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

operator== — Windows 3.11

Ccl::Bool operator==(const CclBuf __far& *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 `true` or `false`. `true` means that the data lengths are the same and the contents are the same.

operator!=

operator!= — OS/2, Windows NT and Windows 95

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

operator!= — Windows 3.11

```
Ccl::Bool operator!=(const CclBuf __far& 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 `true` or `false`. `false` means that the data lengths are the same and the contents are the same.

replace

replace — OS/2, Windows NT and Windows 95

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

replace — Windows 3.11

```
CclBuf __far& replace(unsigned short length,
                      const void __far* dataArea,
                      unsigned short 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

setDataLength — OS/2, Windows NT and Windows 95

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

setDataLength — Windows 3.11

unsigned short setDataLength(unsigned short *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 less than the data area length, the data is padded with X'00''s.

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.

C++ Class: CclBuf

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 9. 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 9 and “Monitoring server availability” on page 16 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 11 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 — OS/2, Windows NT and Windows 95

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

CclConn — Windows 3.11

CclConn(const char __far* *serverName* **= 0,**
const char __far* *userId* **= 0,**
const char __far* *password* **= 0,**
const char __far* *runTran* **= 0,**
const char __far* *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.

© Copyright IBM Corp. 1996,1997

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

cancel

cancel — OS/2, Windows NT and Windows 95

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

cancel — Windows 3.11

void cancel(CclFlow __far& *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

changed — OS/2, Windows NT and Windows 95

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

changed — Windows 3.11

void changed(CclFlow __far& *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.

link

link — OS/2, Windows NT and Windows 95

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

link — Windows 3.11

void link(CclFlow __far& *flow*,
const char __far* *programName*,
CclBuf __far* *commarea* **= 0,**
CclUOW __far* *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 17.

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

listState ——— OS/2, Windows NT and Windows 95

const char* listState() const

listState ——— Windows 3.11

const char __far* 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 "
```

password

password ——— OS/2, Windows NT and Windows 95

const char* password() const

password ——— Windows 3.11

const char __far* password() const

Returns the password held by the CclConn object, or blanks, if there is none.

serverName

serverName ——— OS/2, Windows NT and Windows 95

const char* serverName() const

serverName ——— Windows 3.11

const char __far* serverName() const

Returns the name of the server system held by the **CclConn** object, or blanks if the default server is being used and no calls have yet been made.

status

status ——— OS/2, Windows NT and Windows 95

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

status ——— Windows 3.11

void status(CclFlow __far& *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 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

serverStatusText ——— OS/2, Windows NT and Windows 95

const char* serverStatusText() const

serverStatusText ——— Windows 3.11

const char __far* serverStatusText() const

Returns a string, set by an earlier **status** or **changed** request, indicating the availability of the server.

userId

userId — OS/2, Windows NT and Windows 95

const char* userId() const

userId — Windows 3.11

const char __far* userId() const

Returns the user ID held by the CclConn object, or blanks if none.

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

CclECI()

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

Public methods

exCode

exCode

Ccl::ExCode exCode() const

Returns an enumeration indicating the most recent exception code. The possible values are listed under ExCode in the description of class **Ccl** on page 33.

exCodeText

exCodeText — OS/2, Windows NT and Windows 95

const char* exCodeText() const

exCodeText — Windows 3.11

const char __far* exCodeText() const

Returns a text string describing the most recent exception code.

© Copyright IBM Corp. 1996,1997

handleException

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 10. The default implementation just throws the exception object—except on Windows 3.11 where it just returns to the caller.

instance

instance — OS/2, Windows NT and Windows 95

static CclECI* instance()

instance — Windows 3.11

static CclECI __far* __pascal 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

listState — OS/2, Windows NT and Windows 95

const char* listState() const

listState — Windows 3.11

const char __far* 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

serverCount

unsigned short serverCount() const

Returns the number of available servers to which the client may be connected, as configured in the CICSCLI.INI file. In practice, some or all of these servers may not be available. See “Finding potential servers” on page 8.

serverDesc

serverDesc — OS/2, Windows NT and Windows 95

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

serverDesc — Windows 3.11

const char __far* 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 8.

serverName

serverName — OS/2, Windows NT and Windows 95

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

serverName — Windows 3.11

const char __far* 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 8.

C++ Class: CclECI

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 CICSCLI.INI. You must create one object of this class for each application process before you create CclTerminal objects to connect to CICS servers. The **diagnose**, **exCode**, and **state** methods provide information on error conditions.

CclEPI constructor

CclEPI — OS/2, Windows NT and Windows 95

CclEPI()

CclEPI — Windows 3.11

CclEPI(HWND *hWnd*, UINT *message*=CICS_NOTIFY)

hWnd The handle of the window to which messages are to be sent

message The message to be transmitted when the interface has been initialized.

Defaults to CICS_NOTIFY (WM_USER+1), supplied in include file CCLDEFS.HPP.

This method initializes the CICS EPI interface on the client. An initEPI exception is raised if initialization fails.

In OS/2, Windows NT and Windows 95 environments, initialization of the CICS client EPI is synchronous i.e. initialization is complete when the call to the **CclEPI** constructor returns.

In the Windows 3.11 environment, initialization of the CICS client EPI is deferred synchronous. The **CclEPI** constructor returns immediately, but initialization is not complete until a CICS_NOTIFY message has been received by the application.

The window handle and message id supplied on the CclEPI constructor are also used by the CclTerminal object for initialization and reply notification. See “Using EPI under Windows 3.1” on page 29.

© Copyright IBM Corp. 1996,1997

Public methods

diagnose

diagnose — OS/2, Windows NT and Windows 95

```
const char* diagnose() const
```

diagnose — Windows 3.11

```
const char __far* diagnose() const
```

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

exCode

exCode

```
Ccl::ExCode exCode() const
```

Returns an enumeration indicating the most recent exception code. The possible values are listed under ExCode in the description of class **Ccl** on page 33.

exCodeText

exCodeText — OS/2, Windows NT and Windows 95

```
const char* exCodeText() const
```

exCodeText — Windows 3.11

```
const char __far* exCodeText() const
```

Returns a text string describing the most recent exception code.

handleException

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—except on Windows 3.11 where it just returns to the caller.

serverCount

serverCount

unsigned short serverCount()

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

serverDesc

serverDesc — OS/2, Windows NT and Windows 95

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

serverDesc — Windows 3.11

const char __far* 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 CICSCLI.INI file for the specified server. If the index exceeds the number of servers configured, a maxServers exception is raised.

serverName

serverName ——— OS/2, Windows NT and Windows 95

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

serverName ——— Windows 3.11

const char __far* 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 CICSCLI.INI file for the specified server. If the index exceeds the number of servers configured, a maxServers exception is raised.

state

state

State state() const

Returns an enumeration indicating the state of the EPI. Possible values are:

start	(Windows 3.11 only) EPI initialization is in progress
active	EPI has been initialized successfully
discon	EPI has terminated
error	EPI initialization has failed

terminate

terminate

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.

Enumerations

Message

(Windows 3.11) An enumeration defining the notification message *wParam* parameter. See "Using EPI under Windows 3.1" on page 29.

Possible values are:

`init`	CclEPI initialization is complete.
`connect`	CclTerminal connection is complete.
`reply`	CclTerminal reply has been received.

State

An enumeration indicating the state of the EPI. Possible values are:

`start`	(Windows 3.11 only) EPI initialization is in progress
`active`	EPI has been initialized successfully
`discon`	EPI has terminated
`error`	EPI initialization has failed

C++ Class: CclEPI

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 10. 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, on OS/2, Windows NT and Windows 95, 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

className

className — OS/2, Windows NT and Windows 95

const char* className() const

className — Windows 3.11

const char __far* className() const

Returns the name of the class in which the exception was raised.

diagnose

diagnose — OS/2, Windows NT and Windows 95

const char* diagnose() const

diagnose — Windows 3.11

const char __far* diagnose() const

Returns text explaining the exception for use in diagnostic output, for example:

© Copyright IBM Corp. 1996,1997

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

exCode

exCode

Ccl::ExCode exCode() const

Returns the exception code. See "ExCode" on page 33.

exCodeText

exCodeText ——— OS/2, Windows NT and Windows 95

const char* exCodeText() const

exCodeText ——— Windows 3.11

const char __far* exCodeText() const

Returns a text string that describes the exception code.

exObject

exObject ——— OS/2, Windows NT and Windows 95

void* exObject() const

exObject ——— Windows 3.11

void __far* exObject() const

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.

For the ECI, the pointer should be cast to a **CclFlow***. For example:

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

methodName

methodName —— OS/2, Windows NT and Windows 95

const char* methodName() const

methodName —— Windows 3.11

const char __far* methodName() const

Returns the name of the method in which the exception was raised.

C++ Class: CclException

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.

appendText

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.

2.

appendText

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

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.

© Copyright IBM Corp. 1996,1997

C++ Class: CclField

baseAttribute

baseAttribute

char baseAttribute() const

Returns the 3270 base attribute of the field.

column

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

dataTag

BaseMDT dataTag() const

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

```
modified
unmodified
```

foregroundColor

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 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

inputProt

BaseProt inputProt() const

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

```
protect
unprotect
```

inputType

inputType

BaseType inputType() const

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

```
alphanumeric
numeric
```

intensity

intensity

BaseInts intensity() const

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

```
dark
normal
intense
```

length

length

unsigned short length() const

Returns the total length of the field. (See also the **textLength** method.)

position

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

resetDataTag

void resetDataTag()

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

row

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

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

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

These methods update the field with the given text.

1. **setText**

   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.

2. **setText**

   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

text — OS/2, Windows NT and Windows 95

const char* text() const

text — Windows 3.11

const char __far* text() const

Returns the text currently held in the field.

textLength

textLength

unsigned short textLength() const

Returns the number of characters currently held in the field.

transparency

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

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 11 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

1.

CclFlow — OS/2, Windows NT and Windows 95

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.

2.

CclFlow — Windows 3.11

CclFlow(Ccl::Sync *syncType***)**

syncType The type of synchronization

3.

CclFlow — Windows 3.11

CclFlow(Ccl::Sync *syncType***,**
HWND *windowHandle***,**
UINT *messageId* **= CICS_NOTIFY)**

syncType The type of synchronization

windowHandle The handle of the window to which messages are sent, if deferred synchronous

messageId The message for deferred synchronous notification

© Copyright IBM Corp. 1996,1997

C++ Class: CclFlow

Public methods

abendCode

abendCode — OS/2, Windows NT and Windows 95

```
const char* abendCode() const
```

abendCode — Windows 3.11

```
const char __far* abendCode() const
```

Returns the abend code from the most recently executed CICS transaction, or blank if there have been none.

callType

callType

```
CallType callType() const
```

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

callTypeText

callTypeText — OS/2, Windows NT and Windows 95

```
const char* callTypeText() const
```

callTypeText — Windows 3.11

```
const char __far* callTypeText() const
```

Returns the name of the most recent server request.

connection

connection — OS/2, Windows NT and Windows 95

```
CclConn* connection() const
```

connection — Windows 3.11

CclConn __far* connection() const

Returns a pointer to the CclConn object which represents the server being used, if any, or zeros.

diagnose

diagnose — OS/2, Windows NT and Windows 95

const CclBuf& diagnose() const

diagnose — Windows 3.11

const CclBuf __far& diagnose() const

Returns text explaining the exception for use in diagnostic output; for example:

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

flowId

flowId

unsigned short flowId() const

Returns the unique identity of this CclFlow object.

forceReset

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.

handleReply

handleReply ——— OS/2, Windows NT and Windows 95

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

handleReply ——— Windows 3.11

virtual void handleReply(CclBuf __far* *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 11. To deal with replies, you should subclass **CclFlow** and provide your own implementation of **handleReply**. The default implementation just returns to the caller.

listState

listState ——— OS/2, Windows NT and Windows 95

const char* listState() const

listState ——— Windows 3.11

const char __far* 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

poll ——— OS/2, Windows NT and Windows 95

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

poll ——— Windows 3.11

Ccl::Bool poll(CclBuf __far* *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.

syncType

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 33.

uow

uow ——— OS/2, Windows NT and Windows 95

CclUOW* uow() const

uow ——— Windows 3.11

CclUOW __far* 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

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.

C++ Class: CclFlow

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 — OS/2, Windows NT and Windows 95

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

CclMap — Windows 3.11

CclMap(CclScreen __far* *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

exCode

Ccl::ExCode exCode() const

Returns an enumeration indicating the most recent exception code. The possible values are listed under ExCode in the description of class **Ccl** on page 33.

exCodeText

exCodeText — OS/2, Windows NT and Windows 95

const char* exCodeText() const

© Copyright IBM Corp. 1996,1997

exCodeText ——— Windows 3.11

const char __far* exCodeText() const

Returns a text string describing the most recent exception code.

field

1.

field ——— OS/2, Windows NT and Windows 95

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

field ——— Windows 3.11

CclField __far* 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.

2.

field ——— OS/2, Windows NT and Windows 95

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

field ——— Windows 3.11

CclField __far* 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

namedField — OS/2, Windows NT and Windows 95

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

namedField — Windows 3.11

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

index The index number of the required CclField object.

Returns the address of the *index*th object.

validate

validate — OS/2, Windows NT and Windows 95

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

validate — Windows 3.11

void validate(const MapData __far* *map***,**
const FieldIndex __far* *index***,**
const FieldData __far*
*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

cursorCol

unsigned short cursorCol() const

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

cursorRow

cursorRow

unsigned short cursorRow() const

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

depth

depth

unsigned short depth() const

Returns the number of rows in the screen.

© Copyright IBM Corp. 1996,1997

field

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

1.

field — OS/2, Windows NT and Windows 95

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

field — Windows 3.11

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

index The index number of the field of interest

2.

field — OS/2, Windows NT and Windows 95

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

field — Windows 3.11

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

row The row number of the field
column The column number of the field

fieldCount

fieldCount

unsigned short fieldCount() const

Returns the number of fields in the screen.

setAID

setAID

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

key An AID key. See the AID enumerations at the end of this section.

Sets the AID key value to be passed to the server on the next transmission.

setCursor

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

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
```

C++ Class: CclScreen

Chapter 14. 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

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

syncType The protocol to be used on transmissions to the CICS server. Possible values are:

- `async` asynchronous (not Windows 3.1)
- `dsync` deferred synchronous
- `sync` synchronous (not Windows 3.1 EPI applications)

Public methods

diagnose

diagnose — OS/2, Windows NT and Windows 95

const char* diagnose() const

diagnose — Windows 3.11

const char __far* diagnose() const

Returns a text description of the last error.

handleReply

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 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

terminal — OS/2, Windows NT and Windows 95

CclTerminal* terminal() const

terminal — Windows 3.11

CclTerminal __far* 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

transId — OS/2, Windows NT and Windows 95

const char* transId() const

transId — Windows 3.11

const char __far* 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

C++ Class: CclSession

Chapter 15. 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 — OS/2, Windows NT and Windows 95

CclTerminal(const char* *server* **= NULL,**
const char* *devtype* **= NULL,**
const char* *netname* **= NULL)**

CclTerminal — Windows 3.11

CclTerminal(const char __far* *server* **= NULL,**
const char __far* *devtype* **= NULL,**
const char __far* *netname* **= NULL)**

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.

Creates the CclTerminal object which is used for EPI communication between the client and server.

If the named server is not configured in the `CICSCLI.INI` file, an `unknownServer` exception is raised.

If invalid values are supplied for *server*, *devtype* or *netname*, a `parameter` exception is raised.

If a CclEPI object has not been created, an `initEPI` exception is raised.

© Copyright IBM Corp. 1996,1997

If the maximum number of supported terminal connections has been exceeded, a `maxRequests` exception is raised.

Public methods

diagnose

diagnose

const char* diagnose()

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

disconnect

disconnect

void disconnect()

Disconnect session from CICS.

exCode

exCode

Ccl::ExCode exCode() const

Returns an enumeration indicating the most recent exception code. The possible values are listed under ExCode in the description of class **Ccl** on page 33.

exCodeText

exCodeText — OS/2, Windows NT and Windows 95

const char* exCodeText() const

exCodeText — Windows 3.11

const char __far* exCodeText() const

Returns a text string describing the most recent exception code.

netName

netName ——— OS/2, Windows NT and Windows 95

const char* netName() const

netName ——— Windows 3.11

const char __far* netName() const

Returns the network name of the terminal.

poll

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::`true`. If no reply has been received from CICS yet, the **poll** method returns Ccl::`false`.

On Windows 3.1 a `CICS_NOTIFY` message sent to the application indicates when the **poll** method should be called to receive data from CICS.

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 22.

queryATI

queryATI

ATIState queryATI()

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

```
disabled
enabled
```

screen

screen — OS/2, Windows NT and Windows 95

CclScreen* screen() const

screen — Windows 3.11

CclScreen __far* screen() const

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

send

1\.

send — OS/2, Windows NT and Windows 95

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

send — Windows 3.11

void send(CclSession* *session***,**
const char __far* *transid***,**
const char __far* *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.

2.

send

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

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`.

state

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

serverName — OS/2, Windows NT and Windows 95

const char* serverName() const

serverName — Windows 3.11

const char __far* serverName() const

Returns the name of the CICS server to which this terminal session is connected.

transID

transID — OS/2, Windows NT and Windows 95

```
const char* transID() const
```

transID — Windows 3.11

```
const char __far* transID() const
```

Returns the 4-character name of the current CICS transaction.

Enumerations

ATIState

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

`enabled`
`disabled`

State

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

`start`	(Windows 3.11 environment only) Terminal initialization is in progress
`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 16. 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

CclUOW()

Creates a CclUOW object.

Public methods

backout

backout — OS/2, Windows NT and Windows 95

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

backout — Windows 3.11

void backout(CclFlow __far& ***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.

© Copyright IBM Corp. 1996,1997

C++ Class: CclUOW

commit

commit — OS/2, Windows NT and Windows 95

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

commit — Windows 3.11

void commit(CclFlow __far& *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

forceReset

void forceReset()

Make this UOW inactive and reset it.

listState

listState — OS/2, Windows NT and Windows 95

const char* listState() const

listState — Windows 3.11

const char __far* 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

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.

C++ Class: CclUOW

Glossary

AID. Attention Identifier

ATI. Automatic Transaction Initiation

BMS. Basic Mapping Support

COMMAREA. CICS Communications Area

ECI. External Call Interface

EPI. External Presentation Interface

ISC. Inter-Systems Communication

OLE. Object Linking and Embedding

UOW. Unit of Work

© Copyright IBM Corp. 1996,1997

Index

A

abendCode
  in Public methods
    in CclFlow class 74
Accessing fields on CICS 3270 screens
  in C++ External presentation interface
    in Using the CICS client C++ classes 21
active
  in state 58, 59
activeFlow
  in CclConn class 45
  in CclFlow class 73
  in CclUOW class 97
  in ExCode 33
activeUOW
  in CclConn class 45
  in CclUOW class 97
  in ExCode 33
AID
  in Enumerations
    in CclScreen class 85
alphanumeric
  in BaseType 70
  in inputType 67
appendText
  in Public methods
    in CclField class 65
assign
  in Public methods
    in CclBuf class 37
async
  in CclSession constructor 87
  in Sync 33
Asynchronous reply handling
  in Controlling server interactions
    in C++ External call interface 13
ATIState
  in Enumerations
    in CclTerminal class 96
attachTran (parameter)
  in CclConn constructor 45, 46
attribute (parameter)
  in setBaseAttribute 68
  in setExtAttribute 68
available
available *(continued)*
  in ServerStatus 50

B

backgroundColor
  in Public methods
    in CclField class 65
backout
  in CallType 78
  in Managing logical units of work 17
  in poll 77
  in Public methods
    in CclUOW class 97
  in Server connection 9
baseAttribute
  in Public methods
    in CclField class 66
BaseInts
  in Enumerations
    in CclField class 70
BaseMDT
  in Enumerations
    in CclField class 70
BaseProt
  in Enumerations
    in CclField class 70
BaseType
  in Enumerations
    in CclField class 70
Basic Mapping Support
  in C++ External presentation interface 18
  in Glossary 101
black
  in Color 70
blinkHlt
  in Highlight 71
blue
  in Color 70
Bool
  in Enumerations
    in Ccl class 33
buffer (parameter)
  in CclBuf constructors 37
  in operator= 40
  in operator== 41

© Copyright IBM Corp. 1996,1997

buffer (parameter) *(continued)*
in operator!= 42
in operator+= 41
buffer size 35
bufferOverflow
in ExCode 33

C

C++ External call interface
Controlling server interactions 11
Asynchronous reply handling 13
Deferred synchronous reply handling 14
Synchronous reply handling 12
Finding potential servers 8
Handling exceptions 10
in Using the CICS client C++ classes 8
Managing logical units of work 17
Monitoring server availability 16
Passing data to a server program 9
Server connection 9
C++ External presentation interface
Accessing fields on CICS 3270 screens 21
EPI BMS conversion utility 24
Mapset containing a single map 25
EPI call synchronization types 22
in Using the CICS client C++ classes 18
Starting a 3270 terminal connection to CICS 20
Using EPI BMS Map Classes 28
Using EPI under Windows 3.1 29
callType
in Enumerations
in CclFlow class 78
in Public methods
in CclFlow class 74
callTypeText
in Public methods
in CclFlow class 74
cancel
in CallType 78
in Monitoring server availability 16
in poll 77
in Public methods
in CclConn class 46
in Server connection 9
Ccl
in exCode 51, 56, 79, 92
in poll 77
in syncType 77
Ccl class
Enumerations
Bool 33
ExCode 33
Sync 33
Ccl::async
in EPI call synchronization types 22
Ccl::dsync
in EPI call synchronization types 24
Ccl::sync
in EPI call synchronization types 22
CclBuf
in dataAreaOwner 38, 43
in Passing data to a server program 9
in replace 42
CclBuf class
Enumerations
DataAreaOwner 43
DataAreaType 43
Public methods
assign 37
cut 37
dataArea 38
dataAreaLength 38
dataAreaOwner 38
dataAreaType 39
dataLength 39
insert 39
listState 40
operator= 40
operator== 41
operator!= 42
operator+= 41
replace 42
setDataLength 43
CclBuf constructors
in CclBuf class 36
cclClass
in ExCode 34
CclConn
in CclConn class 45
in CclUOW class 97
in Monitoring server availability 16
in Server connection 9
in serverName 48
CclConn class
Enumerations
ServerStatus 50
Public methods
cancel 46
changed 46

CclConn class *(continued)*
  Public methods *(continued)*
    link 47
    listState 48
    password 48
    serverName 48
    serverStatus 49
    serverStatusText 49
    status 49
    userId 50
CclConn constructor
  in CclConn class 45
CclConn::cancel
  in CallType 78
CclConn::changed
  in CallType 78
CclConn::link
  in CallType 78
CclConn::status
  in CallType 78
CCLCPOS2.LIB
  in Using the CICS client C++ classes 7
CCLCPW32.LIB
  in Using the CICS client C++ classes 7
CCLCPWIN.LIB
  in Using the CICS client C++ classes 7
CclECI
  in CclECI class 51
  in CclECI constructor (protected) 51
  in CclException class 61
  in Finding potential servers 8
  in handleException 52
CclECI class
  Public methods
    exCode 51
    exCodeText 51
    handleException 52
    instance 52
    listState 52
    serverCount 53
    serverDesc 53
    serverName 53
CclECI constructor (protected)
  in CclECI class 51
CclEPI
  in CclEPI class 55
  in CclEPI constructor 55
  in CclException class 61
  in handleException 57
CclEPI class
  Enumerations
    Message 59
    State 59
  Public methods
    diagnose 56
    exCode 56
    exCodeText 56
    handleException 57
    serverCount 57
    serverDesc 57
    serverName 58
    state 58
    terminate 58
CclEPI constructor
  in CclEPI class 55
CclException
  in CclException class 61
  in Handling exceptions 10
CclException class
  Public methods
    className 61
    diagnose 61
    exCode 62
    exCodeText 62
    exObject 62
    methodName 63
CclField
  in Accessing fields on CICS 3270 screens 21
  in C++ External presentation interface 19
  in CclField class 65
CclField class
  Enumerations
    BaseInts 70
    BaseMDT 70
    BaseProt 70
    BaseType 70
    Color 70
    Highlight 71
    Transparency 71
  Public methods
    appendText 65
    backgroundColor 65
    baseAttribute 66
    column 66
    dataTag 66
    foregroundColor 66
    highlight 66
    inputProt 67
    inputType 67
    intensity 67

CclField class *(continued)*
Public methods *(continued)*
length 67
position 68
resetDataTag 68
row 68
setBaseAttribute 68
setExtAttribute 68
setText 69
text 69
textLength 69
transparency 70
CclFlow
in CclFlow class 73
in Controlling server interactions 11
in Deferred synchronous reply handling 16
in ExCode 33
in handleReply 76
CclFlow class
Enumerations
CallType 78
Public methods
abendCode 74
callType 74
callTypeText 74
connection 74
diagnose 75
flowId 75
forceReset 75
handleReply 76
listState 76
poll 76
syncType 77
uow 77
wait 77
CclFlow constructor
in CclFlow class 73
CclFlow*
in exObject 62
CCLICW32.LIB
in Using the CICS client C++ classes 8
CclMap
in C++ External presentation interface 19
in CclMap class 79
in EPI BMS conversion utility 25
in Mapset containing a single map 26
CclMap class
Protected methods
namedField 81
validate 81
CclMap class *(continued)*
Public methods
exCode 79
exCodeText 79
field 80
CclMap constructor
in CclMap class 79
CclScreen
in C++ External presentation interface 19
in CclScreen class 83
in EPI BMS conversion utility 25
CclScreen class
Enumerations
AID 85
Public methods
cursorCol 83
cursorRow 83
depth 83
field 84
fieldCount 84
setAID 84
setCursor 85
width 85
CclSession
in C++ External presentation interface 19
in CclSession class 87
in EPI call synchronization types 22
in handleReply 88
CclSession class
Enumerations
State 89
Public methods
diagnose 87
handleReply 88
state 88
terminal 88
transId 89
CclSession constructor
in CclSession class 87
CclSession::client
in EPI call synchronization types 23
CclSession::idle
in EPI call synchronization types 23
CclSession::server
in EPI call synchronization types 23
CclTerminal
in C++ External presentation interface 19
in CclTerminal class 91
CclTerminal class
Enumerations
ATIState 96

CclTerminal class *(continued)*
  Enumerations *(continued)*
    State 96
  Public methods
    diagnose 92
    disconnect 92
    exCode 92
    exCodeText 92
    netName 93
    poll 93
    queryATI 94
    screen 94
    send 94
    serverName 95
    setATI 95
    state 95
    transID 96
CclTerminal constructor
  in CclTerminal class 91
CclUOW
  in CclUOW class 97
  in CclUOW constructor 97
  in Managing logical units of work 17
CclUOW class
  Public methods
    backout 97
    commit 98
    forceReset 98
    listState 98
    uowId 99
CclUOW constructor
  in CclUOW class 97
CclUOW::backout
  in CallType 78
CclUOW::commit
  in CallType 78
changed
  in CallType 78
  in cancel 46
  in changed 47
  in Monitoring server availability 16
  in poll 77
  in Public methods
    in CclConn class 46
  in Server connection 9
  in serverStatus 49
  in serverStatusText 49
CICS for MVS/ESA
  in Bibliography ix
CICS for OS/2
  in Bibliography ix
CICS on Open Systems
  in Bibliography ix
CICS WWW home page viii
CICS_OS2
  in Using the CICS client C++ classes 7
CICS_W32
  in Using the CICS client C++ classes 7
CICS_WIN
  in Using the CICS client C++ classes 7
CICSCLI.INI
  in CclEPI class 55
  in CclTerminal constructor 91
  in Finding potential servers 8
  in serverCount 53, 57
  in serverDesc 57
  in serverName 58
  in Starting a 3270 terminal connection to CICS 20
CICSDied
  in ExCode 33
CICSECI.HPP
  in Using the CICS client C++ classes 7
CICSEPI.HPP
  in Using the CICS client C++ classes 7
className
  in Public methods
    in CclException class 61
clear
  in AID 85
client
  in EPI call synchronization types 23
  in send 95
  in State 89, 96
col
  in validate 81, 82
col (parameter)
  in setCursor 85
Color
  in backgroundColor 65
  in Enumerations
    in CclField class 70
  in foregroundColor 66
column
  in Public methods
    in CclField class 66
column (parameter)
  in field 80, 84
COMMAREA 35

commarea (parameter)
  in handleReply 76
  in link 47
  in poll 76, 77
commit
  in CallType 78
  in Managing logical units of work 17
  in poll 77
  in Public methods
    in CclUOW class 98
  in Server connection 9
connect
  in ExCode 34
  in Message 59
connection
  in Public methods
    in CclFlow class 74
Controlling server interactions
  Asynchronous reply handling 13
  Deferred synchronous reply handling 14
  in C++ External call interface
    in Using the CICS client C++ classes 11
  Synchronous reply handling 12
cursorCol
  in Public methods
    in CclScreen class 83
cursorRow
  in Public methods
    in CclScreen class 83
cut
  in Public methods
    in CclBuf class 37
cyan
  in Color 70

D

dark
  in BaseInts 70
  in intensity 67
darkBlue
  in Color 70
dataArea
  in Public methods
    in CclBuf class 38
dataArea (parameter)
  in assign 37
  in CclBuf constructors 36
  in insert 39
  in replace 42
dataAreaLength
  in dataLength 39
  in Public methods
    in CclBuf class 38
dataAreaOwner
  in Enumerations
    in CclBuf class 43
  in Public methods
    in CclBuf class 38
DataAreaType
  in CclBuf class 35
  in Enumerations
    in CclBuf class 43
  in Public methods
    in CclBuf class 39
dataLength
  in ExCode 33
  in link 47
  in Public methods
    in CclBuf class 39
dataStream
  in CclScreen class 83
  in ExCode 34
dataTag
  in Public methods
    in CclField class 66
DBCS 35
defaultColor
  in Color 70
defaultHlt
  in Highlight 71
defaultTran
  in Transparency 71
Deferred synchronous reply handling
  in Controlling server interactions
    in C++ External call interface 14
depth
  in Public methods
    in CclScreen class 83
  in validate 81
Determining if a publication is current
  in Preface vii
devtype (parameter)
  in CclTerminal constructor 91
diagnose
  in CclEPI class 55
  in Handling exceptions 11
  in Public methods
    in CclEPI class 56
    in CclException class 61
    in CclFlow class 75

diagnose *(continued)*
in Public methods *(continued)*
in CclSession class 87
in CclTerminal class 92
disabled
in ATIState 96
in queryATI 94
in setATI 95
discon
in state 58, 59, 89, 96
disconnect
in EPI call synchronization types 23
in Public methods
in CclTerminal class 92
dsync
in CclSession constructor 87
in Sync 33

E

ECICPNn.ext
in Sample programs 6
ECICPOn.ext
in Sample programs 6
ECICPWn.ext
in Sample programs 6
ECIVANn.ext
in Sample programs 6
enabled
in ATIState 96
in queryATI 94
in setATI 95
enter
in AID 85
Enumerations
AID 85
ATIState 96
BaseInts 70
BaseMDT 70
BaseProt 70
BaseType 70
Bool 33
CallType 78
Color 70
DataAreaOwner 43
DataAreaType 43
ExCode 33
Highlight 71
in Ccl class 33
in CclBuf class 43
Enumerations *(continued)*
in CclConn class 50
in CclEPI class 59
in CclField class 70
in CclFlow class 78
in CclScreen class 85
in CclSession class 89
in CclTerminal class 96
Message 59
ServerStatus 50
State 59, 89, 96
Sync 33
Transparency 71
Environments supported
in Establishing the working environment 5
EPI BMS conversion utility
in C++ External presentation interface
in Using the CICS client C++ classes 24
Mapset containing a single map 25
EPI call synchronization types
in C++ External presentation interface
in Using the CICS client C++ classes 22
EPICPNn.ext
in Sample programs 6
EPICPOn.ext
in Sample programs 6
EPICPWn.ext
in Sample programs 6
EPIVANn.ext
in Sample programs 6
error
in state 58, 59, 89, 96
Establishing the working environment
Installation
Sample programs 6
except (parameter)
in handleException 52, 57
ExCode
in CclEPI class 55
in Enumerations
in Ccl class 33
in Public methods
in CclECI class 51
in CclEPI class 56
in CclException class 62
in CclMap class 79
in CclTerminal class 92
exCodeText
in Public methods
in CclECI class 51
in CclEPI class 56

exCodeText *(continued)*
in Public methods *(continued)*
in CclException class 62
in CclMap class 79
in CclTerminal class 92
exObject
in Public methods
in CclException class 62
extensible
in CclBuf class 35
in CclBuf constructors 36
in dataAreaType 39, 43
in setDataLength 43
external
in dataAreaOwner 38, 43

F

false
in Bool 33
in operator== 41
in operator!= 42
in poll 93
field
in Mapset containing a single map 26
in Public methods
in CclMap class 80
in CclScreen class 84
fieldCount
in Public methods
in CclScreen class 84
FieldIndex
in validate 82
fields
in validate 81
fields (parameter)
in validate 81, 82
Finding potential servers
in C++ External call interface
in Using the CICS client C++ classes 8
fixed
in CclBuf class 35
in CclBuf constructors 36
in dataAreaType 39, 43
flow (parameter)
in backout 97
in cancel 46
in changed 46, 47
in commit 98
in link 47
flow (parameter) *(continued)*
in status 49
flowId
in Public methods
in CclFlow class 75
forceReset
in Public methods
in CclFlow class 75
in CclUOW class 98
foregroundColor
in Public methods
in CclField class 66

G

gray
in Color 70
green
in Color 70

H

handleException
in CclException class 61
in handleException 52, 57
in Handling exceptions 10
in Public methods
in CclECI class 52
in CclEPI class 57
handleReply
in CclFlow class 73
in EPI call synchronization types 22, 23, 24
in handleReply 76
in poll 77, 93
in Public methods
in CclFlow class 76
in CclSession class 88
in Using EPI BMS Map Classes 28
in Using EPI under Windows 3.1 30
Handling exceptions
in C++ External call interface
in Using the CICS client C++ classes 10
highlight
in Enumerations
in CclField class 71
in highlight 66
in Public methods
in CclField class 66
Home page viii

hWnd (parameter)
in CclEPI constructor 55

I

idle
in EPI call synchronization types 23
in send 95
in State 89, 96
inactive
in CallType 78
index (parameter)
in field 80, 84
in namedField 81
in serverDesc 53, 57
in serverName 53, 58
in validate 81, 82
init
in Message 59
initEPI
in CclEPI constructor 55
in CclTerminal constructor 91
in ExCode 34
inputProt
in Public methods
in CclField class 67
inputType
in Public methods
in CclField class 67
insert
in Public methods
in CclBuf class 39
Installation 6
in Establishing the working environment 6
Sample programs 6
instance
in CclECI class 51
in Finding potential servers 8
in Public methods
in CclECI class 52
intense
in BaseInts 70
in intensity 67
intenseHlt
in Highlight 71
intensity
in Public methods
in CclField class 67
internal
in CclBuf constructors 37
internal *(continued)*
in dataAreaOwner 38, 43
Internet viii
invalidMap
in CclMap constructor 79
in ExCode 34
invalidState
in ExCode 34
in poll 93
in send 95

K

Keeping up-to-date through the Internet
in Preface viii
key (parameter)
in setAID 84

L

labels
in validate 81
len
in validate 82
length
in Public methods
in CclField class 67
length (parameter)
in appendText 65
in assign 37
in CclBuf constructors 36
in cut 37
in insert 39
in replace 42
in setDataLength 43
in setText 69
link
in CallType 78
in CclUOW class 97
in poll 77
in Public methods
in CclConn class 47
in Server connection 9
listState
in Handling exceptions 11
in Public methods
in CclBuf class 40
in CclConn class 48
in CclECI class 52
in CclFlow class 76
in CclUOW class 98

M

Managing logical units of work
 in C++ External call interface
  in Using the CICS client C++ classes 17
map (parameter)
 in validate 81
MAPINQ1Map
 in Mapset containing a single map 26
Mapset containing a single map
 in EPI BMS conversion utility
  in C++ External presentation interface 25
MaxBufferSize (parameter)
 in CclBuf class 35
maxRequests
 in CclTerminal constructor 92
 in ExCode 34
maxServers
 in ExCode 34
 in serverDesc 57
 in serverName 58
maxUOWs
 in ExCode 34
Message
 in Enumerations
  in CclEPI class 59
message (parameter)
 in CclEPI constructor 55
messageId (parameter)
 in CclFlow constructor 73
 in Deferred synchronous reply handling 16
methodName
 in Public methods
  in CclException class 63
modified
 in BaseMDT 70
 in dataTag 66
Monitoring server availability
 in C++ External call interface
  in Using the CICS client C++ classes 16
multipleInstance
 in CclECI class 51
 in ExCode 33

N

n (parameter)
 in position 68
namedField
 in Protected methods
  in CclMap class 81
netName
 in Public methods
  in CclTerminal class 93
netname (parameter)
 in CclTerminal constructor 91
neutral
 in Color 70
newstate (parameter)
 in setATI 95
no
 in Bool 33
 in poll 77
noCICS
 in ExCode 33
noError
 in ExCode 33
noReply
 in ExCode 33
normal
 in BaseInts 70
 in intensity 67
normalHlt
 in Highlight 71
numeric
 in BaseType 70
 in inputType 67

O

off
 in Bool 33
offset (parameter)
 in cut 37, 38
 in dataArea 38
 in insert 39
 in replace 42
on
 in Bool 33
on page
 in exCode 51, 56, 79, 92
OnNotify
 in Using EPI under Windows 3.1 30
OO support in CICS Clients
 in Introduction to OO programming 3
opaqueTran
 in Transparency 71
operator=
 in Public methods
  in CclBuf class 40

operator==
in Public methods
in CclBuf class 41
operator!=
in Public methods
in CclBuf class 42
operator+=
in Public methods
in CclBuf class 41
orange
in Color 70
orTran
in Transparency 71
OS/2
in Environments supported 5

P

PA1
in AID 85
PA3
in AID 85
paleCyan
in Color 70
paleGreen
in Color 70
parameter
in CclMap constructor 79
in CclTerminal constructor 91
in ExCode 34
in send 94
in setCursor 85
in setExtAttribute 69
Passing data to a server program
in C++ External call interface
in Using the CICS client C++ classes 9
password
in Public methods
in CclConn class 48
password (parameter)
in CclConn constructor 45
PF1
in AID 85
PF24
in AID 85
PF3
in Accessing fields on CICS 3270 screens 21
pink
in Color 70
poll
in Deferred synchronous reply handling 15
in poll 77, 93
in Public methods
in CclFlow class 76
in CclTerminal class 93
in Using EPI under Windows 3.1 29
position
in Public methods
in CclField class 68
Preface
Determining if a publication is current vii
Keeping up-to-date through the Internet viii
What this book is about vii
What you need to know before reading this book vii
Who this book is for vii
programName (parameter)
in link 47
protect
in BaseProt 70
in inputProt 67
Protected methods
in CclMap class 81
namedField 81
validate 81
Public methods
abendCode 74
appendText 65
assign 37
backgroundColor 65
backout 97
baseAttribute 66
callType 74
callTypeText 74
cancel 46
changed 46
className 61
column 66
commit 98
connection 74
cursorCol 83
cursorRow 83
cut 37
dataArea 38
dataAreaLength 38
dataAreaOwner 38
dataAreaType 39
dataLength 39
dataTag 66
depth 83

Public methods *(continued)*
diagnose 56, 61, 75, 87, 92
disconnect 92
exCode 51, 56, 62, 79, 92
exCodeText 51, 56, 62, 79, 92
exObject 62
field 80, 84
fieldCount 84
flowId 75
forceReset 75, 98
foregroundColor 66
handleException 52, 57
handleReply 76, 88
highlight 66
in CclBuf class 37
in CclConn class 46
in CclECI class 51
in CclEPI class 56
in CclException class 61
in CclField class 65
in CclFlow class 74
in CclMap class 79
in CclScreen class 83
in CclSession class 87
in CclTerminal class 92
in CclUOW class 97
inputProt 67
inputType 67
insert 39
instance 52
intensity 67
length 67
link 47
listState 40, 48, 52, 76, 98
methodName 63
netName 93
operator= 40
operator== 41
operator!= 42
operator+= 41
password 48
poll 76, 93
position 68
queryATI 94
replace 42
resetDataTag 68
row 68
screen 94
send 94
serverCount 53, 57
Public methods *(continued)*
serverDesc 53, 57
serverName 48, 53, 58, 95
serverStatus 49
serverStatusText 49
setAID 84
setATI 95
setBaseAttribute 68
setCursor 85
setDataLength 43
setExtAttribute 68
setText 69
state 58, 88, 95
status 49
syncType 77
terminal 88
terminate 58
text 69
textLength 69
transId 89, 96
transparency 70
uow 77
uowId 99
userId 50
wait 77
width 85
purple
in Color 70

Q

queryATI
in Public methods
in CclTerminal class 94

R

red
in Color 70
replace
in Public methods
in CclBuf class 42
reply
in Message 59
resetDataTag
in Public methods
in CclField class 68
resource
in ExCode 33

reverseHlt
in Highlight 71
rolledBack
in ExCode 34
row
in Public methods
in CclField class 68
in validate 81, 82
row (parameter)
in field 80, 84
in setCursor 85
runTran (parameter)
in CclConn constructor 45, 46

S

Sample programs
in Installation
in Establishing the working environment 6
screen
in Accessing fields on CICS 3270 screens 21
in CclScreen class 83
in Public methods
in CclTerminal class 94
screen (parameter)
in CclMap constructor 79
in handleReply 88
section
in setAID 84
security
in ExCode 34
send
in CclField class 65
in EPI call synchronization types 22, 23, 24
in poll 93
in Public methods
in CclTerminal class 94
in Starting a 3270 terminal connection to CICS 20
in terminal 88
in Using EPI under Windows 3.1 29, 30
server
in EPI call synchronization types 23
in poll 93
in State 89, 96
server (parameter)
in CclTerminal constructor 91
Server connection
in C++ External call interface
in Using the CICS client C++ classes 9
serverCount
in Finding potential servers 8
in Public methods
in CclECI class 53
in CclEPI class 57
serverDesc
in Finding potential servers 8
in Public methods
in CclECI class 53
in CclEPI class 57
serverName
in CclConn constructor 45
in Finding potential servers 8
in Public methods
in CclConn class 48
in CclECI class 53
in CclEPI class 58
in CclTerminal class 95
serverName (parameter)
in CclConn constructor 45
serverStatus
in changed 47
in Enumerations
in CclConn class 50
in Public methods
in CclConn class 49
in status 49
serverStatusText
in changed 47
in Public methods
in CclConn class 49
in status 49
session (parameter)
in send 94, 95
setAID
in Public methods
in CclScreen class 84
setATI
in Public methods
in CclTerminal class 95
setBaseAttribute
in Public methods
in CclField class 68
setCursor
in Public methods
in CclScreen class 85
setDataLength
in Public methods
in CclBuf class 43

setExtAttribute
  in Public methods
    in CclField class 68
setText
  in Public methods
    in CclField class 69
size, buffer 35
stackPages (parameter)
  in CclFlow constructor 73
start
  in state 58, 59, 96
startdata (parameter)
  in send 94
Starting a 3270 terminal connection to CICS
  in C++ External presentation interface
    in Using the CICS client C++ classes 20
state
  in CclEPI class 55
  in Enumerations
    in CclEPI class 59
    in CclSession class 89
    in CclTerminal class 96
  in handleReply 88
  in Public methods
    in CclEPI class 58
    in CclSession class 88
    in CclTerminal class 95
  in state 88
  in Using EPI under Windows 3.1 30
state (parameter)
  in EPI call synchronization types 23
  in handleReply 88
status
  in CallType 78
  in Monitoring server availability 16
  in poll 77
  in Public methods
    in CclConn class 49
  in Server connection 9
  in serverStatus 49
  in serverStatusText 49
Sync
  in CclSession constructor 87
  in Enumerations
    in Ccl class 33
  in Sync 33
Synchronous reply handling
  in Controlling server interactions
    in C++ External call interface 12
syncType
  in ExCode 33
  in poll 77, 93
  in Public methods
    in CclFlow class 77
  in wait 77
syncType (parameter)
  in CclFlow constructor 73
  in CclSession constructor 87
systemError
  in ExCode 33

T

terminal
  in Public methods
    in CclSession class 88
terminate
  in Public methods
    in CclEPI class 58
text
  in Public methods
    in CclField class 69
text (parameter)
  in appendText 65
  in CclBuf constructors 36
  in operator= 40
  in operator+= 41
  in setText 69
textLength
  in length 67
  in Public methods
    in CclField class 69
threadCreate
  in ExCode 33
threadKill
  in ExCode 33
threadWait
  in ExCode 33
Trademarks and service marks
  in Notices vi
transaction
  in ExCode 33
transId
  in ExCode 34
  in Public methods
    in CclSession class 89
    in CclTerminal class 96
transid (parameter)
  in send 94

transparency
  in Enumerations
    in CclField class 71
  in Public methods
    in CclField class 70
  in transparency 70
true
  in Bool 33
  in operator== 41
  in operator!= 42
  in poll 93
type (parameter)
  in CclBuf constructors 36

U

unavailable
  in ServerStatus 50
underscoreHlt
  in Highlight 71
unit (parameter)
  in link 47
unknown
  in ServerStatus 50
unknownServer
  in CclTerminal constructor 91
  in ExCode 34
unmodified
  in BaseMDT 70
  in dataTag 66
unmodified (parameter)
  in resetDataTag 68
unprotect
  in BaseProt 70
  in inputProt 67
uow
  in Public methods
    in CclFlow class 77
uowId
  in Public methods
    in CclUOW class 99
userId
  in Public methods
    in CclConn class 50
userId (parameter)
  in CclConn constructor 45
Using EPI BMS Map Classes
  in C++ External presentation interface
    in Using the CICS client C++ classes 28
Using EPI under Windows 3.1
  in C++ External presentation interface
    in Using the CICS client C++ classes 29
Using the CICS client C++ classes
  C++ External call interface
    Controlling server interactions 11
    Finding potential servers 8
    Handling exceptions 10
    Managing logical units of work 17
    Monitoring server availability 16
    Passing data to a server program 9
    Server connection 9
  C++ External presentation interface
    Accessing fields on CICS 3270 screens 21
    EPI BMS conversion utility 24
    EPI call synchronization types 22
    Starting a 3270 terminal connection to CICS 20
    Using EPI BMS Map Classes 28
    Using EPI under Windows 3.1 29

V

validate
  in Protected methods
    in CclMap class 81
value (parameter)
  in setExtAttribute 68, 69
view
  in Using EPI under Windows 3.1 29

W

wait
  in Asynchronous reply handling 14
  in Deferred synchronous reply handling 15
  in Public methods
    in CclFlow class 77
  in wait 77
What this book is about
  in Preface vii
What you need to know before reading this book
  in Preface vii
white
  in Color 70
Who this book is for
  in Preface vii
width
  in Public methods
    in CclScreen class 85
  in validate 81

windowHandle (parameter)
in CclFlow constructor 73
in Deferred synchronous reply handling 16
Windows
in Environments supported 5
Windows 95
in Environments supported 5
Windows NT
in Environments supported 5
wParam (parameter)
in Message 59
WWW viii

X

xorTran
in Transparency 71

Y

yellow
in Color 70
yes
in Bool 33
in poll 77

Sending your comments to IBM

CICS Family

OO Programming in C++
for CICS Clients

SC33-1923-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, use the Readers' Comment Form (RCF)
- By fax:
  - From outside the U.K., after your international access code use 44 1962 870229
  - From within the U.K., use 01962 870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: WINVMD(IDRCF)
  - Internet: idrcf@winvmd.vnet.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic to which your comment applies
- Your name/address/telephone number/fax number/network ID.

Readers' Comments

CICS Family

OO Programming in C++
for CICS Clients

SC33-1923-00

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name

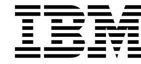
Address

Company or Organization

Telephone

Email

CICS Family
OO Programming in C++ for CICS Clients SC33-1923-00



You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States of America
Belgium	France	Israel	New Zealand	Spain	
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

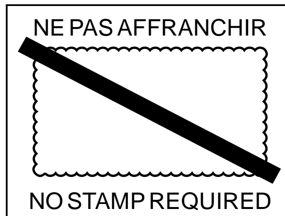
If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO



REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories
Information Development Department (MP095)
Hursley Park,
WINCHESTER, Hants
SO21 2ZZ United Kingdom

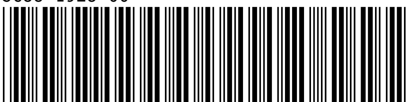
3 Fold along this line

From: Name ____________________
Company or Organization ____________________
Address ____________________
____
EMAIL ____________________
Telephone ____________________

1 Cut along this line

4 Fasten here with adhesive tape

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



Spine information:

CICS Family

OO Programming in C++
for CICS Clients