CICS Transaction Server for z/OS

C++ OO Class Libraries

Version 3 Release 1

SC34-6437-03

CICS Transaction Server for z/OS

C++ OO Class Libraries

Version 3 Release 1

SC34-6437-03

Note!

Before using this information and the product it supports, be sure to read the general information under Notices” on page 345.

This edition applies to Version 3 Release 1 of CICS Transaction Server for z/OS, program number 5655-M15, and to all subsequent versions, releases, and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

Preface

The CICS® family provides robust transaction processing capabilities across the major hardware platforms that IBM® offers, and also across key non-IBM platforms. 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 that allow you to 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 create methods (member functions) that define the behavior associated with objects of a certain type, capturing more of the meaning of the underlying data.

The CICS foundation classes software is a set of facilities that IBM has added to CICS to make it easier for application programmers to develop object oriented programs. It is not intended to be a product in its own right.

The CICS C++ foundation classes, as described here, allow an application programmer to access many of the CICS services that are available via the EXEC CICS procedural application programming interface (API). They also provide an object model, making OO application development simpler and more intuitive.

Who this book is for

This book is for CICS application programmers who want to know how to use the CICS foundation classes.

What this book is about

This book is divided into three parts and three appendixes:

- Part 1, Installation and setup," on page 1 describes how to install the product and check that the installation is complete.
- Part 2, Using the CICS foundation classes," on page 13 describes the classes and how to use them.
- Part 3, Foundation Classes—reference," on page 65 contains the reference material: the class descriptions and their methods.
- For those of you familiar with the EXEC CICS calls, Appendix A, Mapping EXEC CICS calls to Foundation Class methods," on page 279 maps EXEC CICS calls to the foundation class methods detailed in this book...
- ... and Appendix B, Mapping Foundation Class methods to EXEC CICS calls," on page 285 maps them the other way — foundation class methods to EXEC CICS calls.
- Appendix C, Output from sample programs," on page 291 contains the output from the sample programs.

What you need to know before reading this book

Chapter 1, Getting ready for object oriented CICS," on page 3 describes what you need to know to understand this book.

Notes® on terminology

CICS˝ is used throughout this book to mean the CICS element of the IBM CICS Transaction Server for z/OS®, Version 3 Release 1.

RACF˝ is used throughout this book to mean the MVS™ Resource Access Control Facility (RACF®) or any other external security manager that provides equivalent function.

In the programming examples in this book, the dollar symbol ($) is used as a national currency symbol. In countries where the dollar is not the national currency, the local currency symbol should be used.

Part 1. Installation and setup

This part of the book describes the CICS foundation classes installed on your CICS server.

Chapter 1. Getting ready for object oriented CICS

This book makes several assumptions about you, the reader. It assumes you are familiar with:

- Object oriented concepts and technology
- C++ language
- CICS.

This book is not intended to be an introduction to any of these subjects. If the terms in the Glossary˝ on page 299 are not familiar to you, then please consult other sources before going any further. A selection of appropriate books may be found in the bibliography on page Bibliography" on page 301, but you may find other books useful too.

Chapter 2. Installed contents

The CICS foundation classes package consists of several files or datasets. These contain the:

- header files
- executables (DLL's)
- samples
- other CICS Transaction Server for z/OS files

This section describes the files that comprise the CICS C++ Foundation Classes and explains where you can find them on your CICS server.

Header files

The header files are the C++ class definitions needed to compile CICS C++ Foundation Class programs.

C++ Header File	Classes Defined in this Header
ICCABDEH	IccAbendData
ICCBASEH	IccBase
ICCBUFEH	IccBuf
ICCCLKEH	IccClock
ICCCNDEH	IccCondition (struct)
ICCCONEH	IccConsole
ICCCTLEH	IccControl
ICCDATEH	IccDataQueue
ICCEH	see 1
ICCEVTEH	IccEvent
ICCEXCEH	IccException
ICCFILEH	IccFile
ICCFLIEH	IccFileIterator
ICCGLBEH	Icc (struct) (global functions)
ICCJRNEH	IccJournal
ICCMSGEH	IccMessage
ICCPRGEH	IccProgram
ICCRECEH	IccRecordIndex, IccKey, IccRBA and IccRRN
ICCRESEH	IccResource
ICCRIDEH	IccResourceId + subclasses (such as IccConvId)
ICCSEMEH	IccSemaphore
ICCSESEH	IccSession
ICCSRQEH	IccStartRequestQ
ICCSYSEH	IccSystem
ICCTIMEH	IccTime, IccAbsTime, IccTimeInterval, IccTimeOfDay
ICCTMDEH	IccTerminalData
ICCTMPEH	IccTempStore
ICCTRMEH	IccTerminal
ICCTSKEH	IccTask
ICCUSREH	IccUser
ICCVALEH	IccValue (struct)

Notes:

1. A single header that #includes all the above header files is supplied as ICCEH

2. The file ICCMAIN is also supplied with the C++ header files. This contains the **main** function stub that should be used when you build a Foundation Class program.

Location

PDS: CICSTS31.CICS.SDFHC370

Dynamic link library

The Dynamic Link Library is the runtime that is needed to support a CICS C++ Foundation Class program.

Location

ICCFCDLL module in PDS: CICSTS31.CICS.SDFHLOAD

Sample source code

The samples are provided to help you understand how to use the classes to build object oriented applications.

Location

PDS: CICSTS31.CICS.SDFHSAMP

Running the sample applications.

If you have installed the resources defined in the member DFHCURDS, you should be ready to run some of the sample applications.

The sample programs are supplied as source code in library CICSTS31.CICS.SDFHSAMP and before you can run the sample programs, you need to compile, pre-link and link them.To do this, use the procedure ICCFCCL in dataset CICSTS31.CICS.SDFHPROC.

ICCFCCL contains the Job Control Language needed to compile, pre-link and link a CICS user application. Before using ICCFCCL you may find it necessary to perform some customization to conform to your installation standards. See also Compiling Programs” on page 49.

Sample programs such as ICCBUF, ICCCLK and ICC$HEL require no additional CICS resource definitions, and should now execute successfully.

Other sample programs, in particular the DTP samples named ICC$SES1 and ICC$SES2, require additional CICS resource definitions. Refer to the prologues in the source of the sample programs for information about these additional requirements.

Other datasets for CICS Transaction Server for z/OS

CICSTS31.CICS.SDFHSDCK contains the member
    ICCFCIMP - 'sidedeck' containing import control statements

CICSTS31.CICS.SDFHPROC contains the members
    ICCFCC - JCL to compile a CFC user program
    ICCFCCL - JCL to compile, prelink and link a CFC user program
    ICCFCGL - JCL to compile and link an XPLINK program that uses CFC libraries.

ICCFCL - JCL to prelink and link a CFC user program

CICSTS31.CICS.SDFHLOAD contains the members
DFHCURDS - program definitions required for CICS system definition.
DFHCURDI - program definitions required for CICS system definition.

Chapter 3. Hello World

When you start programming in an unaccustomed environment the hardest task is usually getting something—anything—to work and to be seen to be working. The initial difficulty is not in the internals of the program, but in bringing everything together—the CICS server, the programming environment, program inputs and program outputs.

The example shown in this chapter shows how to get started in CICS OO programming. It is intended as an appetizer; Chapter 5, Overview of the foundation classes," on page 17 is a more formal introduction and you should read it before you attempt serious OO programming.

This example could not be much simpler but when it works it is a visible demonstration that you have got everything together and can go on to greater things. The program writes a simple message to the CICS terminal.

There follows a series of program fragments interspersed with commentary. The source for this program can be found in sample ICC$HEL (see Sample source code″ on page 6 for the location).

```
#include icceh.hpp
#include iccmain.hpp
```

The first line includes the header file, ICCEH, which includes the header files for all the CICS Foundation Class definitions. Note that it is coded as `icceh.hpp` to preserve cross-platform, C++ language conventions.

The second line includes the supplied program stub. This stub contains the **main** function, which is the point of entry for any program that uses the supplied classes and is responsible for initializing them correctly. (See Chapter 64, main function," on page 275 for more details). You are strongly advised to use the stub provided but you may in certain cases tailor this stub to your own requirements. The stub initializes the class environment, creates the program control object, then invokes the **run** method, which is where the application program should live.

```
void IccUserControl::run()
{
```

The code that controls the program flow resides not in the **main** function but in the **run** method of a class derived from **IccControl** (see Chapter 20, IccControl class," on page 111). The user can define their own subclass of **IccControl** or, as here, use the default one – **IccUserControl**, which is defined in ICCMAIN – and just provide a definition for the **run** method.

```
    IccTerminal* pTerm = terminal();
```

The **terminal** method of **IccControl** class is used to obtain a pointer to the terminal object for the application to use.

```
    pTerm->erase();
```

The **erase** method clears the current contents of the terminal.

```
    pTerm->send(10, 35, Hello World);
```

The **send** method is called on the terminal object. This causes Hello World to be written to the terminal screen, starting at row 10, column 35.

```
    pTerm->waitForAID();
```

This waits until the terminal user hits an AID (Action Identifier) key.

```
    return;
}
```

Returning from the **run** method causes program control to return to CICS.

Compile and link Hello World

The Hello World sample is provided as sample ICC$HEL (see Sample source code˝ on page 6). Find this sample and copy it to your own work area.

To compile and link any CICS C++ Foundation program you need access to:
1. The source of the program, here ICC$HEL.
2. The Foundation Classes header files (see Header files˝ on page 5).
3. The Foundation Classes dynamic link library (see Dynamic link library˝ on page 6).

See Chapter 8, Compiling, executing, and debugging,” on page 49 for the JCL required to compile the sample program.

Running Hello World on your CICS server

To run the program you have just compiled on your CICS server, you need to make the executable program available to CICS (that is, make sure it is in a suitable directory or load library). Then, depending on your server, you may need to create a CICS program definition for your executable. Finally, you may logon to a CICS terminal and run the program.

To do this,
1. Logon to a CICS terminal and enter either:
   `IHEL`
   or
   `CECI LINK PROGRAM(ICC$HEL)`
2. If you are not using program autoinstall on your CICS region, define the program ICC$HEL to CICS using the supplied transaction CEDA.
3. Log on to a CICS terminal.
4. On CICS terminal run:
   `CECI LINK PROGRAM(ICC$HEL)`

Expected Output from Hello World

This is what you should see on the CICS terminal if program ICC$HEL has been successfuly built and executed.

```
                                   Hello World
```

Hit an Action Identifier, such as the ENTER key, to return.

Part 2. Using the CICS foundation classes

This part of the book describes the CICS foundation classes and how to use them. There is a formal listing of the user interface in Part 3, Foundation Classes—reference," on page 65.

Chapter 4. C++ Objects

This chapter describes how to create, use, and delete objects. In our context an object is an instance of a class. An object cannot be an instance of a base or abstract base class. It is possible to create objects of all the concrete (non-base) classes described in the reference part of this book.

Creating an object

If a class has a constructor it is executed when an object of that class is created. This constructor typically initializes the state of the object. Foundation Classes' constructors often have mandatory positional parameters that the programmer must provide at object creation time.

C++ objects can be created in one of two ways:

1. Automatically, where the object is created on the C++ stack. For example:

```
{
   ClassX     objX
   ClassY     objY(parameter1);
}     //objects deleted here
```

   Here, objX and objY are automatically created on the stack. Their lifetime is limited by the context in which they were created; when they go out of scope they are automatically deleted (that is, their destructors run and their storage is released).

2. Dynamically, where the object is created on the C++ heap. For example:

```
{
   ClassX*   pObjX = new ClassX;
   ClassY*   pObjY = new ClassY(parameter1);
}     //objects NOT deleted here
```

   Here we deal with pointers to objects instead of the objects themselves. The lifetime of the object outlives the scope in which it was created. In the above sample the pointers (pObjX and pObjY) are lost as they go out of scope but the objects they pointed to still exist! The objects exist until they are explicitly deleted as shown here:

```
{
   ClassX*     pObjX = new ClassX;
   ClassY*     pObjY = new ClassY(parameter1);

⋮
   pObjX->method1();
   pObjY->method2();

⋮
   delete pObjX;
   delete pObjY;
}
```

Most of the samples in this book use automatic storage. You are ***advised*** to use automatic storage, because you do not have remember to explicitly delete objects, but you are free to use either style for CICS C++ Foundation Class programs. For more information on Foundation Classes and storage management see Storage management” on page 61.

Using an object

Any of the class public methods can be called on an object of that class. The following example creates object *obj* and then calls method **doSomething** on it:

```
ClassY   obj(TEMP1234);
obj.doSomething();
```

Alternatively, you can do this using dynamic object creation:

```
ClassY*   pObj = new ClassY(parameter1);
pObj->doSomething();
```

Deleting an object

When an object is destroyed its destructor function, which has the same name as the class preceded with ~(tilde), is automically called. (You cannot call the destructor explicitly).

If the object was created automatically it is automatically destroyed when it goes out of scope.

If the object was created dynamically it exists until an explicit **delete** operator is used.

Chapter 5. Overview of the foundation classes

This chapter is a formal introduction to what the Foundation Classes can do for you. See Chapter 3, Hello World," on page 9 for a simple example to get you started. The chapter takes a brief look at the CICS C++ Foundation Class library by considering the following categories in turn:

- Base classes˝
- Resource identification classes˝ on page 18
- Resource classes˝ on page 19
- Support Classes" on page 20.

See Part 3, Foundation Classes—reference," on page 65 for more detailed information on the Foundation Classes.

Every class that belongs to the CICS Foundation Classes is prefixed by **Icc**.

Base classes

IccBase
- **IccRecordIndex**
- **IccResource**
  - **IccControl**
  - **IccTime**
- **IccResourceId**

Figure 1. Base classes

All classes inherit, directly or indirectly, from **IccBase**.

All resource identification classes, such as **IccTermId**, and **IccTransId**, inherit from **IccResourceId** class. These are typically CICS table entries.

All CICS resources—in fact any class that needs access to CICS services—inherit from **IccResource** class.

Base classes enable common interfaces to be defined for categories of class. They are used to create the foundation classes, as provided by IBM, and they can be used by application programmers to create their own derived classes.

IccBase
: The base for every other foundation class. It enables memory management and allows objects to be interrogated to discover which type they are.

IccControl
: The abstract base class that the application program has to subclass and provide with an implementation of the **run** method.

IccResource
: The base class for all classes that access CICS resources or services. See Resource classes" on page 19.

IccResourceId
: The base class for all table entry (resource name) classes, such as **IccFileId** and **IccTempStoreId**.

IccTime
The base class for the classes that store time information: **IccAbsTime**, **IccTimeInterval** and **IccTimeOfDay**.

Resource identification classes

IccBase
- **IccResourceId**
  - **IccConvId**
  - **IccDataQueueId**
  - **IccFileId**
  - **IccGroupId**
  - **IccJournalId**
  - **IccJournalTypeId**
  - **IccLockId**
  - **IccPartnerId**
  - **IccProgramId**
  - **IccRequestId**
    - **IccAlarmRequestId**
  - **IccSysId**
  - **IccTempStoreId**
  - **IccTermId**
  - **IccTPNameId**
  - **IccTransId**
  - **IccUserId**

Figure 2. Resource identification classes

CICS resource identification classes define CICS resource identifiers – typically entries in one of the CICS tables. For example an **IccFileId** object represents a CICS file name – an FCT (file control table) entry. All concrete resource identification classes have the following properties:

- The name of the class ends in **Id**.
- The class is a subclass of the **IccResourceId** class.
- The constructors check that any supplied table entry meets CICS standards. For example, an **IccFileId** object must contain a 1 to 8 byte character field; providing a 9-byte field is not tolerated.

The resource identification classes improve type checking; methods that expect an **IccFileId** object as a parameter do not accept an **IccProgramId** object instead. If character strings representing the resource names are used instead, the compiler cannot check for validity – it cannot check whether the string is a file name or a program name.

Many of the resource classes, described in Resource classes” on page 19, contain resource identification classes. For example, an **IccFile** object contains an **IccFileId** object. You must use the resource object, not the resource identification object to operate on a CICS resource. For example, you must use **IccFile**, rather than **IccFileId** to read a record from a file.

Class	CICS resource	CICS table
IccAlarmRequestId	alarm request	
IccConvId	conversation	

Class	CICS resource	CICS table
IccDataQueueId	data queue	
IccFileId	file	FCT
IccGroupId	group	
IccJournalId	journal	
IccJournalTypeId	journal type	
IccLockId	(Not applicable)	
IccPartnerId	APPC partner definition files	
IccProgramId	program	PPT
IccRequestId	request	
IccSysId	remote system	
IccTempStoreId	temporary storage	TST
IccTermId	terminal	TCT
IccTPNameId	remote APPC TP name	
IccTransId	transaction	PCT
IccUserId	user	

Resource classes

IccBase
- **IccResource**
  - **IccAbendData**
  - **IccClock**
  - **IccConsole**
  - **IccControl**
  - **IccDataQueue**
  - **IccFile**
  - **IccFileIterator**
  - **IccJournal**
  - **IccProgram**
  - **IccSemaphore**
  - **IccSession**
  - **IccStartRequestQ**
  - **IccSystem**
  - **IccTask**
  - **IccTempStore**
  - **IccTerminal**
  - **IccTerminalData**
  - **IccUser**

Figure 3. Resource classes

These classes model the behaviour of the major CICS resources, for example:
- Terminals are modelled by **IccTerminal**.
- Programs are modelled by **IccProgram**.
- Temporary Storage queues are modelled by **IccTempStore**.
- Transient Data queues are modelled by **IccDataQueue**.

All CICS resource classes inherit from the **IccResource** base class. For example, any operation on a CICS resource may raise a CICS condition; the **condition** method of **IccResource** (see page 178) can interrogate it.

Resource classes

(Any class that accesses CICS services ***must*** be derived from **IccResource**).

Class	CICS resource
IccAbendData	task abend data
IccClock	CICS time and date services
IccConsole	CICS console
IccControl	control of executing program
IccDataQueue	transient data queue
IccFile	file
IccFileIterator	file iterator (browsing files)
IccJournal	user or system journal
IccProgram	program (outside executing program)
IccSemaphore	semaphore (locking services)
IccSession	session
IccStartRequestQ	start request queue; asynchronous transaction starts
IccSystem	CICS system
IccTask	current task
IccTempStore	temporary storage queue
IccTerminal	terminal belonging to current task
IccTerminalData	attributes of **IccTerminal**
IccTime	time specification
IccUser	user (security attributes)

Support Classes

IccBase
- **IccBuf**
- **IccEvent**
- **IccException**
- **IccMessage**
- **IccRecordIndex**
  - **IccKey**
  - **IccRBA**
  - **IccRRN**
- **IccResource**
  - **IccTime**
    - **IccAbsTime**
    - **IccTimeInterval**
    - **IccTimeOfDay**

Figure 4. Support classes

These classes are tools that complement the resource classes: they make life easier for the application programmer and thus add value to the object model.

Resource class	Description
IccAbsTime	Absolute time (milliseconds since January 1 1900)
IccBuf	Data buffer (makes manipulating data areas easier)
IccEvent	Event (the outcome of a CICS command)
IccException	Foundation Class exception (supports the C++ exception handling model)
IccTimeInterval	Time interval (for example, five minutes)

Resource class	Description
IccTimeOfDay	Time of day (for example, five minutes past six)

IccAbsTime, IccTimeInterval and **IccTimeOfDay** classes make it simpler for the application programmer to specify time measurements as objects within an application program. **IccTime** is a base class: **IccAbsTime**, **IccTimeInterval**, and **IccTimeOfDay** are derived from **IccTime**.

Consider method **delay** in class **IccTask**, whose signature is as follows:

```
void delay(const IccTime& time, const IccRequestId* reqId = 0);
```

To request a delay of 1 minute and 7 seconds (that is, a time interval) the application programmer can do this:

```
IccTimeInterval time(0, 1, 7);
task()->delay(time);
```

Note: The task method is provided in class **IccControl** and returns a pointer to the application's task object.

Alternatively, to request a delay until 10 minutes past twelve (lunchtime?) the application programmer can do this:

```
IccTimeOfDay lunchtime(12, 10);
task()->delay(lunchtime);
```

The **IccBuf** class allows easy manipulation of buffers, such as file record buffers, transient data record buffers, and COMMAREAs (for more information on **IccBuf** class see Chapter 6, Buffer objects," on page 25).

IccMessage class is used primarily by **IccException** class to encapsulate a description of why an exception was thrown. The application programmer can also use **IccMessage** to create their own message objects.

IccException objects are thrown from many of the methods in the Foundation Classes when an error is encountered.

The **IccEvent** class allows a programmer to gain access to information relating to a particular CICS event (command).

Using CICS resources

To use a CICS resource, such as a file or program, you must first create an appropriate object and then call methods on the object.

Creating a resource object

When you create a resource object you create a representation of the actual CICS resource (such as a file or program). You do not create the CICS resource; the object is simply the application's view of the resource. The same is true of destroying objects.

You are recommended to use an accompanying resource identification object when creating a resource object. For example:
This allows the C++ compiler to protect you against doing something wrong such

```
IccFileId  id(XYZ123);
IccFile    file(id);
```

as:

```
IccDataQueueId  id(WXYZ);
IccFile         file(id);       //gives error at compile time
```

The alternative of using the text name of the resource when creating the object is also permitted:

```
IccFile  file(XYZ123);
```

Singleton classes

Many resource classes, such as **IccFile**, can be used to create multiple resource objects within a single program:

```
IccFileId   id1(File1);
IccFileId   id2(File2);
IccFile     file1(id1);
IccFile     file2(id2);
```

However, some resource classes are designed to allow the programmer to create only ***one*** instance of the class; these are called singleton classes. The following Foundation Classes are singleton:

- **IccAbendData** provides information about task abends.
- **IccConsole**, or a derived class, represents the system console for operator messages.
- **IccControl**, or a derived class, such as **IccUserControl**, controls the executing program.
- **IccStartRequestQ**, or a derived class, allows the application program to start CICS transactions (tasks) asynchronously.
- **IccSystem**, or a derived class, is the application view of the CICS system in which it is running.
- **IccTask**, or a derived class, represents the CICS task under which the executing program is running.
- **IccTerminal**, or a derived class, represents your tasks terminal, provided that your principal facility is a 3270 terminal.

Any attempt to create more than one object of a singleton class results in an error – a C++ exception is thrown.

A class method, **instance**, is provided for each of these singleton classes, which returns a pointer to the requested object and creates one if it does not already exist. For example:

```
IccControl* pControl = IccControl::instance();
```

Calling methods on a resource object

Any of the public methods can be called on an object of that class. For example:

```
IccTempStoreId id(TEMP1234);
IccTempStore   temp(id);
temp.writeItem(Hello TEMP1234);
```

Method **writeItem** writes the contents of the string it is passed (Hello TEMP1234) to the CICS Temporary Storage queue TEMP1234.

Chapter 6. Buffer objects

The Foundation Classes make extensive use of **IccBuf** objects – buffer objects that simplify the task of handling pieces of data or records. Understanding the use of these objects is a necessary precondition for much of the rest of this book.

Each of the CICS Resource classes that involve passing data to CICS (for example by writing data records) and getting data from CICS (for example by reading data records) make use of the **IccBuf** class. Examples of such classes are **IccConsole**, **IccDataQueue**, **IccFile**, **IccFileIterator**, **IccJournal**, **IccProgram**, **IccSession**, **IccStartRequestQ**, **IccTempStore**, and **IccTerminal**.

IccBuf class

IccBuf, which is described in detail in the reference part of this book, provides generalized manipulation of data areas. Because it can be used in a number of ways, there are several **IccBuf** constructors that affect the behavior of the object. Two important attributes of an **IccBuf** object are now described.

Data area ownership

IccBuf has an attribute indicating whether the data area has been allocated inside or outside of the object. The possible values of this attribute are internal and external. It can be interrogated by using the **dataAreaOwner** method.

Internal/External ownership of buffers

When **DataAreaOwner** = external, it is the application programmer's responsibility to ensure the validity of the storage on which the **IccBuf** object is based. If the storage is invalid or inappropriate for a particular method applied to the object, unpredictable results will occur.

Data area extensibility

This attribute defines whether the length of the data area within the **IccBuf** object, once created, can be increased. The possible values of this attribute are fixed and extensible. It can be interrogated by using the **dataAreaType** method.

As an object that is fixed cannot have its data area size increased, the length of the data (for example, a file record) assigned to the **IccBuf** object must not exceed the data area length, otherwise a C++ exception is thrown.

Note: By definition, an extensible buffer ***must*** also be internal.

IccBuf constructors

There are several forms of the **IccBuf** constructor, used when creating **IccBuf** objects. Some examples are shown here.

```
IccBuf buffer;
```

This creates an internal and extensible data area that has an initial length of zero. When data is assigned to the object the data area length is automatically extended to accommodate the data being assigned.

```
IccBuf buffer(50);
```

This creates an internal and extensible data area that has an initial length of 50 bytes. The data length is zero until data is assigned to the object. If 50 bytes of data are assigned to the object, both the data length and the data area length return a value of 50. When more than 50 bytes of data are assigned into the object, the data area length is automatically (that is, without further intervention) extended to accommodate the data.

```
IccBuf buffer(50, IccBuf::fixed);
```

This creates an internal and fixed data area that has a length of 50 bytes. If an attempt is made to assign more than 50 bytes of data into the object, the data is truncated and an exception is thrown to notify the application of the error situation.

```
struct MyRecordStruct
{
    short id;
    short code;
    char  data(30);
    char  rating;
};
MyRecordStruct myRecord;
IccBuf buffer(sizeof(MyRecordStruct), &myRecord);
```

This creates an **IccBuf** object that uses an external data area called myRecord. By definition, an external data area is also fixed. Data can be assigned using the methods on the **IccBuf** object or using the `myRecord` structure directly.

```
IccBuf buffer(Hello World);
```

This creates an internal and extensible data area that has a length equal to the length of the string Hello World. The string is copied into the objects data area. This initial data assignment can then be changed using one of the manipulation methods (such as **insert**, **cut**, or **replace**) provided.

```
IccBuf buffer(Hello World);
buffer <<  out there;
IccBuf buffer2(buffer);
```

Here the copy constructor creates the second buffer with almost the same attributes as the first; the exception is the data area ownership attribute – the second object always contains an internal data area that is a copy of the data area in the first. In the above example buffer2 contains Hello World out there and has both data area length and data length of 21.

IccBuf methods

An **IccBuf** object can be manipulated using a number of supplied methods; for example you can append data to the buffer, change the data in the buffer, cut data out of the buffer, or insert data into the middle of the buffer. The operators **const char***, **=**, **+=**, **==**, **!=**, and **<<** have been overloaded in class **IccBuf**. There are also methods that allow the **IccBuf** attributes to be queried. For more details see the reference section.

Working with IccResource subclasses

To illustrate this, consider writing a queue item to CICS temporary storage using **IccTempstore** class.

```
IccTempStore store(TEMP1234);
IccBuf       buffer(50);
```

The **IccTempStore** object created is the applications view of the CICS temporary storage queue named TEMP1234. The **IccBuf** object created holds a 50-byte data area (it also happens to be extensible).

```
buffer = Hello Temporary Storage Queue;
store.writeItem(buffer);
```

The character string Hello Temporary Storage Queue is copied into the buffer. This is possible because the **operator=** method has been overloaded in the **IccBuf** class.

The **IccTempStore** object calls its **writeItem** method, passing a reference to the **IccBuf** object as the first parameter. The contents of the **IccBuf** object are written out to the CICS temporary storage queue.

Now consider the inverse operation, reading a record from the CICS resource into the application programs **IccBuf** object:

```
buffer = store.readItem(5);
```

The **readItem** method reads the contents of the fifth item in the CICS Temporary Storage queue and returns the data as an **IccBuf** reference.

The C++ compiler actually resolves the above line of code into two method calls, **readItem** defined in class **IccTempStore** and **operator=** which has been overloaded in class **IccBuf**. This second method takes the contents of the returned **IccBuf** reference and copies its data into the buffer.

The above style of reading and writing records using the foundation classes is typical. The final example shows how to write code – using a similar style to the above example – but this time accessing a CICS transient data queue.

```
IccDataQueue queue(DATQ);
IccBuf       buffer(50);
buffer = queue.readItem();
buffer << Some extra data;
queue.writeItem(buffer);
```

The **readItem** method of the **IccDataQueue** object is called, returning a reference to an **IccBuf** which it then assigns (via **operator=** method, overloaded in class **IccBuf**) to the `buffer` object. The character string – Some extra data – is appended to the buffer (via **operator<<** method, overloaded in class **IccBuf**). The **writeItem** method then writes back this modified buffer to the CICS transient data queue.

You can find further examples of this syntax in the samples presented in the following chapters, which describe how to use the foundation classes to access CICS services.

Please refer to the reference section for further information on the **IccBuf** class. You might also find the supplied sample – ICC$BUF – helpful.

Chapter 7. Using CICS Services

This chapter describes how to use CICS services. The following services are considered in turn:

- File control˝
- Program control˝ on page 34
- Starting transactions asynchronously˝ on page 36
- Transient Data˝ on page 39
- Temporary storage˝ on page 41
- Terminal control˝ on page 43
- Time and date services˝ on page 45

File control

The file control classes – **IccFile**, **IccFileId**, **IccKey**, **IccRBA**, and**IccRRN** – allow you to read, write, update and delete records in files.In addition, **IccFileIterator** class allows you to browse through all the records in a file.

An **IccFile** object is used to represent a file. It is convenient, but not necessary, to use an **IccFileId** object to identify a file by name.

An application program reads and writes its data in the form of individual records. Each read or write request is made by a method call. To access a record, the program must identify both the file and the particular record.

VSAM (or VSAM-like) files are of the following types:

KSDS
: Key-sequenced: each record is identified by a key – a field in a predefined position in the record. Each key must be unique in the file.

  The logical order of records within a file is determined by the key. The physical location is held in an index which is maintained by VSAM.

  When browsing, records are found in their logical order.

ESDS
: Entry-sequenced: each record is identified by its relative byte address (RBA).

  Records are held in an ESDS in the order in which they were first loaded into the file. New records are always added at the end and records may not be deleted or have their lengths altered.

  When browsing, records are found in the order in which they were originally written.

RRDS file
: Relative record: records are written in fixed-length slots. A record is identified by the relative record number (RRN) of the slot which holds it.

Reading records

A read operation uses two classes – **IccFile** to perform the operation and one of **IccKey**, **IccRBA**, and **IccRRN** to identify the particular record, depending on whether the file access type is KSDS, ESDS, or RRDS.

The **readRecord** method of **IccFile** class actually reads the record.

Reading KSDS records

Before reading a record you must use the **registerRecordIndex** method of **IccFile** to associate an object of class **IccKey** with the file.

You must use a key, held in the **IccKey** object, to access records. A complete key is a character string of the same length as the physical file's key. Every record can be separately identified by its complete key.

A key can also be generic. A generic key is shorter than a complete key and is used for searching for a set of records.The **IccKey** class has methods that allow you to set and change the key.

IccFile class has methods **isReadable**, **keyLength**, **keyPosition**, **recordIndex**, and **recordLength**, which help you when reading KSDS records.

Reading ESDS records

You must use a relative byte address (RBA) held in an **IccRBA** object to access the beginning of a record.

Before reading a record you must use the **registerRecordIndex** method of **IccFile** to associate an object of class **IccRBA** with the file.

IccFile class has methods **isReadable**, **recordFormat**, **recordIndex**, and **recordLength** that help you when reading ESDS records.

Reading RRDS records

You must use a relative record number (RRN) held in an **IccRRN** object to access a record.

Before reading a record you must use **registerRecordIndex** method of **IccFile** to associate an object of class **IccRRN** with the file.

IccFile class has methods **isReadable**, **recordFormat**, **recordIndex**, and **recordLength** which help you when reading RRDS records.

Writing records

Writing records is also known as adding records. This section describes writing records that have not previously been written. Writing records that already exist is not permitted unless they have been previously been put into update mode. See Updating records˝ on page 31 for more information.

Before writing a record you must use **registerRecordIndex** method of **IccFile** to associate an object of class **IccKey**, **IccRBA**, or **IccRRN** with the file. The **writeRecord** method of **IccFile** class actually writes the record.

A write operation uses two classes – **IccFile** to perform the operation and one of **IccKey**, **IccRBA**, and **IccRRN** to identify the particular record, depending on whether the file access type is KSDS, ESDS, or RRDS.

If you have more than one record to write, you can improve the speed of writing by using mass insertion of data. You begin and end this mass insertion by calling the **beginInsert** and **endInsert** methods of **IccFile**.

Writing KSDS records

You must use a key, held in an **IccKey** object to access records. A complete key is a character string that uniquely identifies a record. Every record can be separately identified by its complete key.

The **writeRecord** method of **IccFile** class actually writes the record.

IccFile class has methods **isAddable**, **keyLength**, **keyPosition**, **recordIndex**, **recordLength**, and **registerRecordIndex** which help you when writing KSDS records.

Writing ESDS records

You must use a relative byte address (RBA) held in an **IccRBA** object to access the beginning of a record.

IccFile class has methods **isAddable**, **recordFormat**, **recordIndex**, **recordLength**, and **registerRecordIndex** that help you when writing ESDS records.

Writing RRDS records

Use the **writeRecord** method to add a new ESDS record. After writing the record you can use the **number** method on the **IccRBA** object to discover the assigned relative byte address for the record you have just written.

IccFile class has methods **isAddable**, **recordFormat**, **recordIndex**, **recordLength**, and **registerRecordIndex** that help you when writing RRDS records.

Updating records

Updating a record is also known as rewriting a record. Before updating a record you must first read it, using **readRecord** method in update mode. This locks the record so that nobody else can change it.

Use **rewriteRecord** method to actually update the record. Note that the **IccFile** object remembers which record is being processed and this information is not passed in again.

For an example, see code fragment: Read record for update” on page 34.

The base key in a KSDS file must not be altered when the record is modified. If the file definition allows variable-length records, the length of the record can be changed.

The length of records in an ESDS, RRDS, or fixed-length KSDS file must not be changed on update.

For a file defined to CICS as containing fixed-length records, the length of record being updated must be the same as the original length. The length of an updated record must not be greater than the maximum defined to VSAM.

Deleting records

Records can never be deleted from an ESDS file.

Deleting normal records

The **deleteRecord** method of **IccFile** class deletes one or more records, provided they are not locked by virtue of being in update mode. The records to be deleted are defined by the **IccKey** or **IccRRN** object.

Deleting locked records

The **deleteLockedRecord** method of **IccFile** class deletes a record which has been previously locked by virtue of being put in update mode by the **readRecord** method.

Browsing records

Browsing, or sequential reading of files uses another class – **IccFileIterator**. An object of this class must be associated with an **IccFile** object and an **IccKey**, **IccRBA**, or **IccRRN** object. After this association has been made the **IccFileIterator** object can be used without further reference to the other objects.

Browsing can be done either forwards, using **readNextRecord** method or backwards, using **readPreviousRecord** method. The **reset** method resets the **IccFileIterator** object to point to the record specified by the **IccKey** or **IccRBA** object.

Examples of browsing files are shown in page
Code fragment List all records in assending order of key˝ on page 33.

Example of file control

This sample program demonstrates how to use the **IccFile** and **IccFileIterator** classes. The source for this sample can be found in the samples directory (see Sample source code˝ on page 6) in file ICC$FIL. Here the code is presented without any of the terminal input and output that can be found in the source file.

```
#include icceh.hpp
#include iccmain.hpp
```

The first two lines include the header files for the Foundation Classes and the standard **main** function which sets up the operating environment for the application program.

```
const char* fileRecords[] =
{
   //NAME          KEY  PHONE     USERID
   BACH, J S      003  00-1234   BACH      ,
   BEETHOVEN, L   007  00-2244   BEET      ,
   CHOPIN, F      004  00-3355   CHOPIN    ,
   HANDEL, G F    005  00-4466   HANDEL    ,
   MOZART, W A    008  00-5577   WOLFGANG
};
```

This defines several lines of data that are used by the sample program.

```
void IccUserControl::run()
{
```

The **run** method of **IccUserControl** class contains the user code for this example. As a terminal is to be used, the example starts by creating a terminal object and clearing the associated screen.

```
short        recordsDeleted = 0;
IccFileId    id(ICCKFILE);
IccKey       key(3,IccKey::generic);
IccFile      file( id );
file.registerRecordIndex( &key );
key = 00;
recordsDeleted = file.deleteRecord();
```

The *key* and *file* objects are first created and then used to delete all the records whose key starts with 00 in the KSDS file ICCKFILE. *key* is defined as a generic key having 3 bytes, only the first two of which are used in this instance.

```
IccBuf       buffer(40);
key.setKind( IccKey::complete );
for (short j = 0; j < 5; j++)
{
    buffer = fileRecords[j];
    key.assign(3, fileRecords[j]+15);
    file.writeRecord( buffer );
}
```

This next fragment writes all the data provided into records in the file. The data is passed by means of an **IccBuf** object that is created for this purpose. **setKind** method is used to change *key* from generic to complete.

The **for** loop between these calls loops round all the data, passing the data into the buffer, using the **operator=** method of **IccBuf**, and thence into a record in the file, by means of **writeRecord**. On the way the key for each record is set, using **assign**, to be a character string that occurs in the data (3 characters, starting 15 characters in).

```
IccFileIterator fIterator( &file, &key );
key = 000;
buffer = fIterator.readNextRecord();
while (fIterator.condition() == IccCondition::NORMAL)
{
    term->sendLine(- record read: [%s],(const char*) buffer);
    buffer = fIterator.readNextRecord();
}
```

The loop shown here lists to the terminal, using **sendLine**, all the records in ascending order of key. It uses an **IccFileIterator** object to browse the records. It starts by setting the minimum value for the key which, as it happens, does not actually exist in this example, and relying on CICS to find the first record in key sequence.

The loop continues until any condition other than NORMAL is returned.

```
key = \xFF\xFF\xFF;
fIterator.reset( &key );
buffer = fIterator.readPreviousRecord();
while (fIterator.condition() == IccCondition::NORMAL)
{
    buffer = fIterator.readPreviousRecord();
}
```

The next loop is nearly identical to the last, but lists the records in reverse order of key.

```
    key = 008;
    buffer = file.readRecord( IccFile::update );
    buffer.replace( 4, 5678, 23);
    file.rewriteRecord( buffer );
```

This fragment reads a record for update, locking it so that others cannot change it. It then modifies the record in the buffer and writes the updated record back to the file.

```
    buffer = file.readRecord();
```

The same record is read again and sent to the terminal, to show that it has indeed been updated.

```
    return;
}
```

The end of **run**, which returns control to CICS.

See Appendix C, Output from sample programs," on page 291 for the expected output from this sample.

Program control

This section describes how to access and use a program other than the one that is currently executing. Program control uses **IccProgram** class, one of the resource classes.

Programs may be loaded, unloaded and linked to, using an **IccProgram** object. An **IccProgram** object can be interrogated to obtain information about the program. See Chapter 37, IccProgram class," on page 165 for more details.

The example shown here shows one program calling another two programs in turn, with data passing between them via a COMMAREA. One program is assumed to be local, the second is on a remote CICS system. The programs are in two files, ICC$PRG1 and ICC$PRG2, in the samples directory (see Sample source code˝ on page 6).

Most of the terminal IO in these samples has been omitted from the code that follows.

```
#include icceh.hpp
#include iccmain.hpp
void IccUserControl::run()
{
```

The code for both programs starts by including the header files for the Foundation Classes and the stub for **main** method. The user code is located in the **run** method of the **IccUserControl** class for each program.

```
IccSysId       sysId( ICC2 );
IccProgram     icc$prg2( ICC$PRG2 );
IccProgram     remoteProg( ICC$PRG3 );
IccBuf         commArea( 100, IccBuf::fixed );
```

The first program (ICC$PRG1) creates an **IccSysId** object representing the remote region, and two **IccProgram** objects representing the local and remote programs that will be called from this program. A 100 byte, fixed length buffer object is also created to be used as a communication area between programs.

```
icc$prg2.load();
if (icc$prg2.condition() == IccCondition::NORMAL)
{
  term->sendLine( Loaded program: %s <%s> Length=%ld Address=%x,
                  icc$prg2.name(),
                  icc$prg2.conditionText(),
                  icc$prg2.length(),
                  icc$prg2.address() );
  icc$prg2.unload();
}
```

The program then attempts to load and interrogate the properties of program ICC$PRG2.

```
 commArea = DATA SET BY ICC$PRG1;
 icc$prg2.link( &commArea );
```

The communication area buffer is set to contain some data to be passed to the first program that ICC$PRG1 links to (ICC$PRG2). ICC$PRG1 is suspended while ICC$PRG2 is run.

The called program, ICC$PRG2, is a simple program, the gist of which is as follows:

```
IccBuf& commArea = IccControl::commArea();
commArea = DATA RETURNED BY ICC$PRG2;
return;
```

ICC$PRG2 gains access to the communication area that was passed to it. It then modifies the data in this communication area and passes control back to the program that called it.

The first program (ICC$PRG1) now calls another program, this time on another system, as follows:

```
remoteProg.setRouteOption( sysId );
commArea = DATA SET BY ICC$PRG1;
remoteProg.link( &commArea );
```

The **setRouteOption** requests that calls on this object are routed to the remote system. The communication area is set again (because it will have been changed by ICC$PRG2) and it then links to the remote program (ICC$PRG3 on system ICC2).

The called program uses CICS temporary storage but the three lines we consider are:

```
IccBuf& commArea = IccControl::commArea();
commArea = DATA RETURNED BY ICC$PRG3;
return;
```

Again, the remote program (ICC$PRG3) gains access to the communication area that was passed to it. It modifies the data in this communication area and passes control back to the program that called it.

```
  return;
};
```

Finally, the calling program itself ends and returns control to CICS.

See Appendix C, Output from sample programs,” on page 291 for the expected output from these sample programs.

Starting transactions asynchronously

The **IccStartRequestQ** class enables a program to start another CICS transaction instance asynchronously (and optionally pass data to the started transaction). The same class is used by a started transaction to gain access to the data that the task that issued the start request passed to it. Finally start requests (for some time in the future) can be cancelled.

Starting transactions

You can use any of the following methods to establish what data will be sent to the started transaction:

- **registerData** or **setData**
- **setQueueName**
- **setReturnTermId**
- **setReturnTransId**

The actual start is requested using the **start** method.

Accessing start data

A started transaction can access its start data by invoking the **retrieveData** method. This method stores all the start data attributes in the **IccStartRequestQ** object such that the individual attributes can be accessed using the following methods:

- **data**
- **queueName**
- **returnTermId**
- **returnTransId**

Cancelling unexpired start requests

Unexpired start requests (that is, start requests for some future time that has not yet been reached) can be cancelled using the **cancel** method.

Example of starting transactions

CICS system	ICC1	ICC2
Transaction	ISR1/ITMP	ISR2
Program	ICC$SRQ1/ICC$TMP	ICC$SRQ2
Terminal	PEO1	PEO2

The scenario is as follows. We start transaction ISR1 on terminal PEO1 on system ICC1. This issues two start requests; the first is cancelled before it has expired. The second starts transaction ISR2 on terminal PEO2 on system ICC2. This transaction accesses its start data and finishes by starting transaction ITMP on the original terminal (PEO1 on system ICC1).

The programs can be found in the samples directory (see Sample source code˝ on page 6) as files ICC$SRQ1 and ICC$SRQ2. Here the code is presented without the terminal IO requests.

Transaction ISR1 runs program ICC$SRQ1 on system ICC1. Let us consider this program first:

```
#include icceh.hpp
#include iccmain.hpp
void IccUserControl::run()
{
```

These lines include the header files for the Foundation Classes, and the **main** function needed to set up the class library for the application program. The **run** method of **IccUserControl** class contains the user code for this example.

```
   IccRequestId          req1;
   IccRequestId          req2(REQUEST1);
   IccTimeInterval       ti(0,0,5);
   IccTermId             remoteTermId(PE02);
   IccTransId            ISR2(ISR2);
   IccTransId            ITMP(ITMP);
   IccBuf                buffer;
   IccStartRequestQ* startQ = startRequestQ();
```

Here we are creating a number of objects:

req1 An empty **IccRequestId** object ready to identify a particular start request.

req2 An **IccRequestId** object containing the user-supplied identifier REQUEST1.

ti An **IccTimeInterval** object representing 0 hours, 0 minutes, and 5 seconds.

remoteTermId
An **IccTermId** object; the terminal on the remote system where we start a transaction.

ISR2 An **IccTransId** object; the transaction we start on the remote system.

ITMP An **IccTransId** object; the transaction that the started transaction starts on this program's terminal.

buffer
An **IccBuf** object that holds start data.

Finally, the **startRequestQ** method of **IccControl** class returns a pointer to the single instance (singleton) class **IccStartRequestQ**.

```
  startQ->setRouteOption( ICC2 );
  startQ->registerData( &buffer );
  startQ->setReturnTermId( terminal()->name() );
  startQ->setReturnTransId( ITMP );
  startQ->setQueueName( startqnm );
```

This code fragment prepares the start data that is passed when we issue a start request. The **setRouteOption** says we will issue the start request on the remote system, ICC2. The **registerData** method associates an **IccBuf** object that will contain the start data (the contents of the **IccBuf** object are not extracted until we actually issue the start request). The **setReturnTermId** and **setReturnTransId** methods allow the start requester to pass a transaction and terminal name to the started transaction. These fields are typically used to allow the started transaction to start ***another*** transaction (as specified) on another terminal, in this case ours.

The **setQueueName** is another piece of information that can be passed to the started transaction.

```
  buffer = This is a greeting from program 'icc$srq1'!!;
  req1 = startQ->start( ISR2, &remoteTermId, &ti );
  startQ->cancel( req1 );
```

Here we set the data that we pass on the start requests. We start transaction ISR2 after an interval *ti* (5 seconds). The request identifier is stored in *req1*. Before the five seconds has expired (that is, immediately) we cancel the start request.

```
  req1 = startQ->start( ISR2, &remoteTermID, &ti, &req2 );
  return;
}
```

Again we start transaction ISR2 after an interval *ti* (5 seconds). This time the request is allowed to expire so transaction ISR2 is started on the remote system. Meanwhile, we end by returning control to CICS.

Let us now consider the started program, ICC$SRQ2.

```
  IccBuf                buffer;
  IccRequestId          req(REQUESTX);
  IccTimeInterval       ti(0,0,5);
  IccStartRequestQ* startQ = startRequestQ();
```

Here, as in ICC$SRQ1, we create a number of objects:

buffer
: An **IccBuf** object to hold the start data we were passed by our caller (ICC$SRQ1).

req
: An **IccRequestId** object to identify the start we will issue on our caller's terminal.

ti
: An **IccTimeInterval** object representing 0 hours, 0 minutes, and 5 seconds.

The **startRequestQ** method of **IccControl** class returns a pointer to the singleton class **IccStartRequestQ**.

```
  if ( task()->startType() != IccTask::startRequest )
  {
    term->sendLine(
          This program should only be started via the StartRequestQ);
    task()->abend( OOPS );
  }
```

Here we use the **startType** method of **IccTask** class to check that ICC$SRQ2 was started by the **start** method, and not in any other way (such as typing the transaction name on a terminal). If it was not started as intended, we abend with an OOPS abend code.

```
  startQ->retrieveData();
```

We retrieve the start data that we were passed by ICC$SRQ1 and store within the **IccStartRequestQ** object for subsequent access.

```
buffer = startQ->data();
term->sendLine( Start buffer contents = [%s], buffer.dataArea() );
term->sendLine( Start queue= [%s], startQ->queueName() );
term->sendLine( Start rtrn = [%s], startQ->returnTransId().name());
term->sendLine( Start rtrm = [%s], startQ->returnTermId().name() );
```

The start data buffer is copied into our **IccBuf** object. The other start data items (queue, returnTransId, and returnTermId) are displayed on the terminal.

```
task()->delay( ti );
```

We delay for five seconds (that is, we sleep and do nothing).

```
startQ->setRouteOption( ICC1 );
```

The **setRouteOption** signals that we will start on our caller's system (ICC1).

```
startQ->start( startQ->returnTransId(),startQ->returnTermId());
return;
```

We start a transaction called ITMP (the name of which was passed by ICC$SRQ1 in the returnTransId start information) on the originating terminal (where ICC$SRQ1 completed as it started this transaction). Having issued the start request, ICC$SRQ1 ends, by returning control to CICS.

Finally, transaction ITMP runs on the first terminal. This is the end of this demonstration of starting transactions asynchronously.

See Appendix C, Output from sample programs,” on page 291 for the expected output from these sample programs.

Transient Data

The transient data classes, **IccDataQueue** and **IccDataQueueId**, allow you to store data in transient data queues for subsequent processing.

You can:
- Read data from a transient data queue (**readItem** method)
- Write data to a transient data queue (**writeItem** method)
- Delete a transient data queue (**empty** method)

An **IccDataQueue** object is used to represent a temporary storage queue. An **IccDataQueueId** object is used to identify a queue by name. Once the **IccDataQueueId** object is initialized it can be used to identify the queue as an alternative to using its name, with the advantage of additional error detection by the C++ compiler.

The methods available in **IccDataQueue** class are similar to those in the **IccTempStore** class. For more information on these see Temporary storage” on page 41.

Reading data

The **readItem** method is used to read items from the queue. It returns a reference to the **IccBuf** object that contains the information.

Writing data

The **writeItem** method of **IccDataQueue** adds a new item of data to the queue, taking the data from the buffer specified.

Deleting queues

The **empty** method deletes all items on the queue.

Example of managing transient data

This sample program demonstrates how to use the **IccDataQueue** and **IccDataQueueId** classes. It can be found in the samples directory (see Sample source code˝ on page 6) as file ICC$DAT. Here the code is presented without the terminal IO requests.

```
#include icceh.hpp
#include iccmain.hpp
```

The first two lines include the header files for the foundation classes and the standard **main** function that sets up the operating environment for the application program.

```
const char* queueItems[] =
{
  Hello World - item 1,
  Hello World - item 2,
  Hello World - item 3
};
```

This defines some buffer for the sample program.

```
void IccUserControl::run()
{
```

The **run** method of **IccUserControl** class contains the user code for this example.

```
  short itemNum =1;
  IccBuf              buffer( 50 );
  IccDataQueueId      id( ICCQ );
  IccDataQueue        queue( id );
  queue.empty();
```

This fragment first creates an identification object, of type IccDataQueueId containing ICCQ. It then creates an **IccDataQueue** object representing the transient data queue ICCQ, which it empties of data.

```
  for (short i=0 ; i<3 ; i++)
  {
    buffer = queueItems[i];
    queue.writeItem( buffer );
  }
```

This loop writes the three data items to the transient data object. The data is passed by means of an **IccBuf** object that was created for this purpose.

```
  buffer = queue.readItem();
  while ( queue.condition() == IccCondition::NORMAL )
  {
    buffer = queue.readItem();
  }
```

Having written out three records we now read them back in to show they were successfully written.

```
  return;
}
```

The end of **run**, which returns control to CICS.

See Appendix C, Output from sample programs,” on page 291 for the expected output from this sample program.

Temporary storage

The temporary storage classes, **IccTempStore** and **IccTempStoreId**, allow you to store data in temporary storage queues.

You can:
- Read an item from the temporary storage queue (**readItem** method)
- Write a new item to the end of the temporary storage queue (**writeItem** method)
- Update an item in the temporary storage queue (**rewriteItem** method)
- Read the next item in the temporary storage queue (**readNextItem** method)
- Delete all the temporary data (**empty** method)

An **IccTempStore** object is used to represent a temporary storage queue. An **IccTempStoreId** object is used to identify a queue by name. Once the **IccTempStoreId** object is initialized it can be used to identify the queue as an alternative to using its name, with the advantage of additional error detection by the C++ compiler.

The methods available in **IccTempStore** class are similar to those in the **IccDataQueue** class. For more information on these see Transient Data” on page 39.

Reading items

The **readItem** method of **IccTempStore** reads the specified item from the temporary storage queue. It returns a reference to the **IccBuf** object that contains the information.

Writing items

Writing items is also known as adding items. This section describes writing items that have not previously been written. Writing items that already exist can be done using the **rewriteItem** method. See Updating items˝ on page 42 for more information.

The **writeItem** method of **IccTempStore** adds a new item at the end of the queue, taking the data from the buffer specified. If this is done successfully, the item number of the record added is returned.

Updating items

Updating an item is also known as rewriting an item. The **rewriteItem** method of **IccTempStore** class is used to update the specified item in the temporary storage queue.

Deleting items

You cannot delete individual items in a temporary storage queue. To delete *all* the temporary data associated with an **IccTempStore** object use the **empty** method of **IccTempStore** class.

Example of Temporary Storage

This sample program demonstrates how to use the **IccTempStore** and **IccTempStoreId** classes. This program can be found in the samples directory (see Sample source code˝ on page 6) as file ICC$TMP. The sample is presented here without the terminal IO requests.

```
#include icceh.hpp
#include iccmain.hpp
#include <stdlib.h>
```

The first three lines include the header files for the foundation classes, the standard **main** function that sets up the operating environment for the application program, and the standard library.

```
const char* bufferItems[] =
{
  Hello World - item 1,
  Hello World - item 2,
  Hello World - item 3
};
```

This defines some buffer for the sample program.

```
void IccUserControl::run()
{
```

The **run** method of **IccUserControl** class contains the user code for this example.

```
  short itemNum = 1;
  IccTempStoreId id(ICCSTORE);
  IccTempStore   store( id );
  IccBuf         buffer( 50 );
  store.empty();
```

This fragment first creates an identification object, **IccTempStoreId** containing the field ICCSTORE. It then creates an **IccTempStore** object representing the temporary storage queue ICCSTORE, which it empties of records.

```
  for (short j=1 ; j <= 3 ; j++)
  {
    buffer = bufferItems[j-1];
    store.writeItem( buffer );
  }
```

This loop writes the three data items to the Temporary Storage object. The data is passed by means of an **IccBuf** object that was created for this purpose.

```
buffer = store.readItem( itemNum );
while ( store.condition() == IccCondition::NORMAL )
{
  buffer.insert( 9, Modified  );
  store.rewriteItem( itemNum, buffer );
  itemNum++;
  buffer = store.readItem( itemNum );
}
```

This next fragment reads the items back in, modifies the item, and rewrites it to the temporary storage queue. First, the **readItem** method is used to read the buffer from the temporary storage object. The data in the buffer object is changed using the **insert** method of **IccBuf** class and then the **rewriteItem** method overwrites the buffer. The loop continues with the next buffer item being read.

```
itemNum = 1;
buffer = store.readItem( itemNum );
while ( store.condition() == IccCondition::NORMAL )
{
  term->sendLine(  - record #%d = [%s], itemNum,
                  (const char*)buffer );
  buffer = store.readNextItem();
}
```

This loop reads the temporary storage queue items again to show they have been updated.

```
  return;
}
```

The end of **run**, which returns control to CICS.

See Appendix C, Output from sample programs,” on page 291 for the expected output from this sample program.

Terminal control

The terminal control classes, **IccTerminal**, **IccTermId**, and **IccTerminalData**, allow you to send data to, receive data from, and find out information about the terminal belonging to the CICS task.

An **IccTerminal** object is used to represent the terminal that belongs to the CICS task. It can only be created if the transaction has a 3270 terminal as its principal facility. The **IccTermId** class is used to identify the terminal. **IccTerminalData**, which is owned by **IccTerminal**, contains information about the terminal characteristics.

Sending data to a terminal

The **send** and **sendLine** methods of **IccTerminal** class are used to write data to the screen. Alternatively, you can use the << operators to send data to the terminal.

Before sending data to a terminal, you may want to set, for example, the position of the cursor on the screen or the color of the text. The **set...** methods allow you to do this. You may also want to erase the data currently displayed at the terminal, using the **erase** method, and free the keyboard so that it is ready to receive input, using the **freeKeyboard** method.

Receiving data from a terminal

The **receive** and **receive3270data** methods of **IccTerminal** class are used to receive data from the terminal.

Finding out information about a terminal

You can find out information about both the characteristics of the terminal and its current state.

The **data** object points to the **IccTerminalData** object that contains information about the characteristics of the terminal. The methods described in **IccTerminalData** on page 247 allow you to discover, for example, the height of the screen or whether the terminal supports Erase Write Alternative. Some of the methods in **IccTerminal** also give you information about characteristics, such as how many lines a screen holds.

Other methods give you information about the current state of the terminal. These include **line**, which returns the current line number, and **cursor**, which returns the current cursor position.

Example of terminal control

This sample program demonstrates how to use the **IccTerminal**, **IccTermId**, and **IccTerminalData** classes. This program can be found in the samples directory (see Sample source code˝ on page 6) as file ICC$TRM.

```
#include icceh.hpp
#include iccmain.hpp
```

The first two lines include the header files for the Foundation Classes and the standard **main** function that sets up the operating environment for the application program.

```
void IccUserControl::run()
{
  IccTerminal& term = *terminal();
  term.erase();
```

The **run** method of **IccUserControl** class contains the user code for this example. As a terminal is to be used, the example starts by creating a terminal object and clearing the associated screen.

```
  term.sendLine( First part of the line... );
  term.send( ... a continuation of the line. );
  term.sendLine( Start this on the next line );
  term.sendLine( 40, Send this to column 40 of current line );
  term.send( 5, 10, Send this to row 5, column 10 );
  term.send( 6, 40, Send this to row 6, column 40 );
```

This fragment shows how the **send** and **sendLine** methods are used to send data to the terminal. All of these methods can take **IccBuf** references (const IccBuf&) instead of string literals (const char*).

```
  term.setNewLine();
```

This sends a blank line to the screen.

```
  term.setColor( IccTerminal::red );
  term.sendLine( A Red line of text.);
  term.setColor( IccTerminal::blue );
  term.setHighlight( IccTerminal::reverse );
  term.sendLine( A Blue, Reverse video line of text.);
```

The **setColor** method is used to set the colour of the text on the screen and the **setHighlight** method to set the highlighting.

```
  term << A cout sytle interface...  << endl;
  term << you can  << chain input together;
       << use different types, eg numbers:  << (short)123 <<
       << (long)4567890 <<   << (double)123456.7891234 << endl;
  term << ... and everything is buffered till you issue a flush.
       << flush;
```

This fragment shows how to use the iostream–like interface **endl** to start data on the next line. To improve performance, you can buffer data in the terminal until **flush** is issued, which sends the data to the screen.

```
  term.send( 24,1, Program 'icc$trm' complete: Hit PF12 to End );
  term.waitForAID( IccTerminal::PF12 );
  term.erase();
```

The **waitForAID** method causes the terminal to wait until the specified key is hit, before calling the **erase** method to clear the display.

```
  return;
}
```

The end of **run**, which returns control to CICS.

See Appendix C, Output from sample programs,” on page 291 for the expected output from this sample program.

Time and date services

The **IccClock** class controls access to the CICS time and date services. **IccAbsTime** holds information about absolute time (the time in milliseconds that have elapsed since the beginning of 1900), and this can be converted to other forms of date and time. The methods available on **IccClock** objects and on **IccAbsTime** objects are very similar.

Example of time and date services

This sample program demonstrates how to use **IccClock** class. The source for this program can be found in the samples directory (see Sample source code˝ on page 6) as file ICC$CLK. The sample is presented here without the terminal IO requests.

```
#include icceh.hpp
#include iccmain.hpp
void IccUserControl::run()
{
```

The first two lines include the header files for the Foundation Classes and the standard **main** function that sets up the operating environment for the application program.

The **run** method of **IccUserControl** class contains the user code for this example.

```
IccClock clock;
```

This creates a clock object.

```
term->sendLine( date() = [%s],
                clock.date() );
term->sendLine( date(DDMMYY) = [%s],
                clock.date(IccClock::DDMMYY) );
term->sendLine( date(DDMMYY,':') = [%s],
                clock.date(IccClock::DDMMYY,':'));
term->sendLine( date(MMDDYY) = [%s],
                clock.date(IccClock::MMDDYY));
term->sendLine( date(YYDDD) = [%s],
                clock.date(IccClock::YYDDD));
```

Here the **date** method is used to return the date in the format specified by the *format* enumeration. In order the formats are system, DDMMYY, DD:MM:YY, MMDDYY and YYDDD. The character used to separate the fields is specified by the *dateSeparator* character (that defaults to nothing if not specified).

```
term->sendLine( daysSince1900() = %ld,
                clock.daysSince1900());
term->sendLine( dayOfWeek() = %d,
                clock.dayOfWeek());
if ( clock.dayOfWeek() == IccClock::Friday )
  term->sendLine( 40, Today IS Friday );
else
  term->sendLine( 40, Today is NOT Friday );
```

This fragment demonstrates the use of the **daysSince1900** and **dayOfWeek** methods. **dayOfWeek** returns an enumeration that indicates the day of the week. If it is Friday, a message is sent to the screen, Today IS Friday; otherwise the message Today is NOT Friday is sent.

```
term->sendLine( dayOfMonth() = %d,
                clock.dayOfMonth());
term->sendLine( monthOfYear() = %d,
                clock.monthOfYear());
```

This demonstrates the **dayOfMonth** and **monthOfYear** methods of **IccClock** class.

```
term->sendLine( time() = [%s],
                clock.time() );
term->sendLine( time('-') = [%s],
                clock.time('-') );
term->sendLine( year() = [%ld],
                clock.year());
```

The current time is sent to the terminal, first without a separator (that is HHMMSS format), then with - separating the digits (that is, HH-MM-SS format). The year is sent, for example 1996.

```
  return;
};
```

The end of **run**, which returns control to CICS.

See Appendix C, Output from sample programs," on page 291 for the expected output from this sample program.

Chapter 8. Compiling, executing, and debugging

This chapter describes how to compile, execute, and debug a CICS Foundation Class program. The following are considered in turn:

- Compiling Programs˝
- Executing Programs˝
- Debugging Programs˝

Compiling Programs

To compile and link a CICS Foundation Class program you need access to the following:

- The source of the program you are compiling

  Your C++ program source code needs `#include` statements for the Foundation Class headers and the Foundation Class `main()` program stub:

  ```
  #include icceh.hpp
  #include iccmain.hpp
  ```
- The IBM C++ compiler
- The Foundation Classes header files (see Header files˝ on page 5)
- The Foundation Classes dynamic link library (DLL) (see Dynamic link library˝ on page 6)

Note that, when using the Foundation Classes, you do not need to translate the EXEC CICS API so the translator program should not be used.

The following sample job statements show how to compile, prelink and link a program called ICC$HEL:

```
//ICC$HEL   JOB 1,user_name,MSGCLASS=A,CLASS=A,NOTIFY=userid
//PROCLIB  JCLLIB ORDER=(CICSTS31.CICS.SDFHPROC)
//ICC$HEL   EXEC ICCFCCL,INFILE=indatasetname(ICC$HEL),OUTFILE=outdatasetname(ICC$HEL)
//
```

Executing Programs

To run a compiled and linked (that is, executable) Foundation Classes program you need to do the following:

1. Make the executable program available to CICS. This involves making sure the program is in a suitable directory or load library. Depending on your server, you may also need to create a CICS program definition (using CICS resource definition facilities) before you can execute the program.
2. Logon to a CICS terminal.
3. Run the program.

Debugging Programs

Having successfully compiled, linked and attempted to execute your Foundation Classes program you may need to debug it.

There are three options available to help debug a CICS Foundation Classes program:

1. Use a symbolic debugger

2. Run the Foundation Class Program with tracing active
3. Run the Foundation Class Program with the CICS Execution Diagnostic Facility

Symbolic Debuggers

A symbolic debugger allows you to step through the source of your CICS Foundation Classes program. **Debug Tool**, a component of CODE/370, is shipped as a feature with IBM C/C++ for OS/390®.

To debug a CICS Foundation Classes program with a symbolic debugger, you need to compile the program with a flag that adds debugging information to your executable. For CICS Transaction Server for z/OS, this is TEST(ALL).

For more information see *Debug Tool User's Guide and Reference*, SC09-2137.

Tracing a Foundation Class Program

The CICS Foundation Classes can be configured to write a trace file for debugging/service purposes.

Activating the trace output

In CICS Transaction Server for z/OS, exception trace is always active.

The CETR transaction controls the auxilliary and internal traces for all CICS programs including those developed using the C++ classes.

Execution Diagnostic Facility

For the EXEC CICS API, there is a CICS facility called the Execution Diagnostic Facility (EDF) that allows you to step through your CICS program stopping at each EXEC CICS call. This does not make much sense from the CICS Foundation Classes because the display screen shows the procedural EXEC CICS call interface rather than the CICS Foundation Class type interface. However, this may be of use to programmers familiar with the EXEC CICS interface.

Enabling EDF

To enable EDF, use the pre-processor macro `ICC_EDF` – this can be done in your source code ***before*** including the file ICCMAIN as follows:

```
#define ICC_EDF       //switch EDF on
#include iccmain.hpp
```

Alternatively use the appropriate flag on your compiler CPARM to declare ICC_EDF.

For more information about using EDF see "Execution diagnostic facility (EDF)" in *CICS Application Programming Guide*.

Chapter 9. Conditions, errors, and exceptions

This chapter describes how the Foundation Classes have been designed to respond to various error situations they might encounter. These will be discussed under the following headings:

- Foundation Class Abend codes˝
- C++ Exceptions and the Foundation Classes˝
- CICS conditions˝ on page 53
- Platform differences˝ on page 56

Foundation Class Abend codes

For serious errors (such as insufficient storage to create an object) the Foundation Classes immediately terminate the CICS task.

All CICS Foundation Class abend codes are of the form ACLx. If your application is terminated with an abend code starting ACL then please refer to *CICS Messages and Codes*, GC34-6442.

C++ Exceptions and the Foundation Classes

C++ exceptions are managed using the reserved words **try**, **throw**, and **catch**. Please refer to your compiler's documentation or one of the C++ books in the bibliography for more information.

Here is sample ICC$EXC1 (see Sample source code˝ on page 6):

```
#include icceh.hpp
#include iccmain.hpp
class Test {
public:
  void  tryNumber( short num ) {
    IccTerminal* term = IccTerminal::instance();
    *term << Number passed =  << num << endl << flush;
    if ( num > 10 ) {
      *term << >>Out of Range - throwing exception << endl << flush;
      throw !!Number is out of range!!;
    }
  }
};
```

The first two lines include the header files for the Foundation Classes and the standard **main** function that sets up the operating environment for the application program.

We then declare class **Test**, which has one public method, **tryNumber**. This method is implemented inline so that if an integer greater than ten is passed an exception is thrown. We also write out some information to the CICS terminal.


```
void IccUserControl::run()
{
  IccTerminal* term = IccTerminal::instance();
  term->erase();
  *term << This is program 'icc$exc1' ... << endl;
  try  {
    Test test;
    test.tryNumber( 1 );
    test.tryNumber( 7 );
    test.tryNumber( 11 );
    test.tryNumber( 6 );
  }
  catch( const char* exception )   {
    term->setLine( 22 );
    *term << Exception caught:  << exception << endl << flush;
  }
  term->send( 24,1,Program 'icc$exc1' complete: Hit PF12 to End );
  term->waitForAID( IccTerminal::PF12 );
  term->erase();
  return;
}
```

The **run** method of **IccUserControl** class contains the user code for this example.

After erasing the terminal display and writing some text, we begin our **try** block. A **try** block can scope any number of lines of C++ code.

Here we create a **Test** object and invoke our only method, **tryNumber**, with various parameters. The first two invocations (1, 7) succeed, but the third (11) causes **tryNumber** to throw an exception. The fourth **tryNumber** invocation (6) is not executed because an exception causes the program execution flow to leave the current **try** block.

We then leave the **try** block and look for a suitable **catch** block. A suitable **catch** block is one with arguments that are compatible with the type of exception being thrown (here a **char***). The **catch** block writes a message to the CICS terminal and then execution resumes at the line after the **catch** block.

The output from this CICS program is as follows:

```
This is program 'icc$exc1' ...
Number passed = 1
Number passed = 7
Number passed = 11
>>Out of Range - throwing exception
Exception caught: !!Number is out of range!!
Program 'icc$exc1' complete: Hit PF12 to End
```

The CICS C++ Foundation Classes do not throw **char*** exceptions as in the above sample but they do throw **IccException** objects instead.

There are several types of **IccException**. The **type** method returns an enumeration that indicates the type. Here is a description of each type in turn.

objectCreationError
: An attempt to create an object was invalid. This happens, for example, if an attempt is made to create a second instance of a singleton class, such as **IccTask**.

invalidArgument

A method was called with an invalid argument. This happens, for example, if an **IccBuf** object with too much data is passed to the **writeItem** method of the **IccTempStore** class by the application program.

It also happens when attempting to create a subclass of **IccResourceId**, such as **IccTermId**, with a string that is too long.

The following sample can be found in the samples directory (see Sample source code˝ on page 6) as file ICC$EXC2. The sample is presented here without many of the terminal IO requests.

```
#include icceh.hpp
#include iccmain.hpp
void IccUserControl::run()
{
  try
  {
    IccTermId id1( 1234 );
    IccTermId id2( 12345);
  }
  catch( IccException& exception )
  {
    terminal()->send( 21, 1, exception.summary() );
  }
  return;
}
```

In the above example the first **IccTermId** object is successfully created, but the second caused an **IccException** to be thrown, because the string 12345 is 5 bytes where only 4 are allowed. See Appendix C, Output from sample programs," on page 291 for the expected output from this sample program.

invalidMethodCall

A method cannot be called. A typical reason is that the object cannot honor the call in its current state. For example, a **readRecord** call on an **IccFile** object is only honored if an **IccRecordIndex** object, to specify ***which*** record is to be read, has already been associated with the file.

CICSCondition

A CICS condition, listed in the **IccCondition** structure, has occurred in the object and the object was configured to throw an exception.

familyConformanceError

Family subset enforcement is on for this program and an operation that is not valid on all supported platforms has been attempted.

internalError

The CICS foundation classes have detected an internal error. Please call service.

CICS conditions

The CICS foundation classes provide a powerful framework for handling conditions that happen when executing an application. Accessing a CICS resource can raise a number of CICS conditions as documented in Part 3, Foundation Classes—reference," on page 65.

A condition might represent an error or simply information being returned to the calling application; the deciding factor is often the context in which the condition is raised.

The application program can handle the CICS conditions in a number of ways. Each CICS resource object, such as a program, file, or data queue, can handle CICS conditions differently, if required.

A resource object can be configured to take one of the following actions for each condition it can encounter:

noAction
: Manual condition handling

callHandleEvent
: Automatic condition handling

throwException
: Exception handling

abendTask
: Severe error handling.

Manual condition handling (noAction)

This is the default action for all CICS conditions (for any resource object). It can be explicitly activated as follows:

```
IccTempStore   temp(TEMP1234);
temp.setActionOnCondition(IccResource::noAction,
                          IccCondition::QIDERR);
```

This setting means that when CICS raises the QIDERR condition as a result of action on the temp object, no action is taken. This means that the condition must be handled manually, using the **condition** method. For example:

```
IccTempStore   temp(TEMP1234);
IccBuf         buf(40);
temp.setActionOnCondition(IccResource::noAction,
                          IccCondition::QIDERR);
buf = temp.readNextItem();
switch (temp.condition())
{
case IccCondition::QIDERR:
    //do whatever here
   ⋮
default:
    //do something else here
}
```

Automatic condition handling (callHandleEvent)

Activate this for any CICS condition, such as QIDERR, as follows:

```
IccTempStore   temp(TEMP1234);
temp.setActionOnCondition(IccResource::callHandleEvent,
                          IccCondition::QIDERR);
```

When a call to any method on object temp causes CICS to raise the QIDERR condition, **handleEvent** method is automatically called. As the **handleEvent** method is only a virtual method, this call is only useful if the object belongs to a subclass of **IccTempStore** and the **handleEvent** method has been overridden.

Make a subclass of **IccTempStore**, declare a constructor, and override the **handleEvent** method.

```
class MyTempStore : public IccTempStore
{
public:
    MyTempStore(const char* storeName) : IccTempStore(storeName) {}
    HandleEventReturnOpt handleEvent(IccEvent& event);
};
```

Now implement the **handleEvent** method.

```
IccResource::HandleEventReturnOpt MyTempStore::handleEvent(IccEvent& event)
{
    switch (event.condition())
    {
    case ...

   ⋮
    case IccCondition::QIDERR:
        //Handle QIDERR condition here.

   ⋮
        //
    default:
        return rAbendTask;
    }
}
```

This code is called for any **MyTempStore** object which is configured to callHandleEvent for a particular CICS condition.

Exception handling (throwException)

Activate this for any CICS condition, such as QIDERR, as follows:

```
IccTempStore   temp(TEMP1234);
temp.setActionOnCondition(IccResource::throwException,
                          IccCondition::QIDERR);
```

Exception handling is by means of the C++ exception handling model using **try**, **throw**, and **catch**. For example:

```
try
{
    buf = temp.readNextItem();

   ⋮
}
catch (IccException& exception)
{
    //Exception handling code

   ⋮
}
```

An exception is thrown if any of the methods inside the try block raise the QIDERR condition for object temp. When an exception is thrown, C++ unwinds the stack and resumes execution at an appropriate **catch** block – it is not possible to resume within the **try** block. For a fuller example of the above, see sample ICC$EXC3.

Note: Exceptions can be thrown from the Foundation Classes for many reasons other than this example – see C++ Exceptions and the Foundation Classes˝ on page 51 for more details.

Severe error handling (abendTask)

This option allows CICS to terminate the task when certain conditions are raised. Activate this for any CICS condition, such as QIDERR, as follows:

```
IccTempStore   temp(TEMP1234);
temp.setActionOnCondition(IccResource::abendTask,
                          IccCondition::QIDERR);
```

If CICS raises the QIDERR condition for object temp the CICS task terminates with an ACL3 abend.

Platform differences

Note: References in this section to other CICS platforms—CICS OS/2 and CICS for AIX®—are included for completeness. There have been Technology Releases of the CICS Foundation Classes on those platforms.

The CICS Foundation Classes, as described here, are designed to be independent of the particular CICS platform on which they are running. There are however some differences between platforms; these, and ways of coping with them, are described here.

Applications can be run in one of two modes:

fsAllowPlatformVariance
: Applications written using the CICS Foundation Classes are able to access all the functions available on the target CICS server.

fsEnforce
: Applications are restricted to the CICS functions that are available across ***all*** CICS Servers (MVS, UNIX, and OS/2).

The default is to allow platform variance and the alternative is to force the application to only use features which are common to all CICS platforms.

The class headers are the same for all platforms and they support (that is, define) all the CICS functions that are available through the Foundation Classes on any of the CICS platforms. The restrictions on each platform are documented in Part 3, Foundation Classes—reference,” on page 65. Platform variations exist at:

- object level
- method level
- parameter level

Object level

Some objects are not supported on certain platforms. For example **IccJournal** objects cannot be created on CICS OS/2 as CICS OS/2 does not support journalling services. **IccConsole** objects cannot be created on CICS for AIX as CICS for AIX does not support console services.

Any attempt to create **IccJournal** on CICS OS/2, or an **IccConsole** object on CICS for AIX causes an **IccException** object of type platformError to be thrown, but would be acceptable on the other platforms

For example:

```
IccJournal journal7(7);  //No good on CICS OS/2
```

or

```
IccConsole* cons = console();  //No good on CICS for AIX
```

If you initialize your application with fsEnforce selected (see initializeEnvironment˝ on page 67) the previous examples both cause an **IccException** object, of type familyConformanceError to be thrown on all platforms.

Unlike objects of the **IccConsole** and **IccJournal** classes, most objects can be created on any CICS server platform. However the use of the methods can be restricted. Part 3, Foundation Classes—reference,” on page 65 fully documents all platform restrictions.

Method level

Consider, for example method **programId** in the **IccControl** class:

```
void IccUserControl::run()
{
    if (strcmp(programId.name(), PROG1234) == 0)
        //do something
}
```

Here method **programId** executes correctly on CICS OS/2 and CICS/ESA but throws an **IccException** object of type platformError on CICS for AIX.

Alternatively, if you initialize your application with family subset enforcement on (see **initializeEnvironment** function of **Icc** structure) then method **programId** throws an **IccException** object of type familyConformanceError on *any* CICS server platform.

Parameter level

At this level a method is supported on all platforms, but a particular positional parameter has some platform restrictions. Consider method **abend** in **IccTask** class.

Platform differences

```
task()->abend();                                        1

task()->abend(WXYZ);                                    2

task()->abend(WXYZ, IccTask::respectAbendHandler);      3

task()->abend(WXYZ, IccTask::ignoreAbendHandler);       4

task()->abend(WXYZ, IccTask::ignoreAbendHandler,        5

              IccTask::suppressDump);
```

Abends 1 to 4 run successfully on all CICS server platforms.

If family subset enforcement is off, abend 5 throws an **IccException** object of type platformError on a CICS for AIX platform, but not on a CICS OS/2 or CICS/ESA platform.

If family subset enforcement is on, abend 5 throws an **IccException** object of type familyConformanceError, irrespective of the target CICS platform.

Chapter 10. Miscellaneous

This chapter describes the following:

Polymorphic Behavior

Polymorphism (*poly* = many, *morphe* = form) is the ability to treat many different forms of an object as if they were the same.

Polymorphism is achieved in C++ by using inheritance and virtual functions. Consider the scenario where we have three forms (ExpenseForm, LoanForm, PurchaseForm) that are specializations of a general Form:

```
                     Form
          ______________________________

   ExpenseForm        LoanForm      PurchaseForm
```

Each form needs printing at some time. In procedural programming, we would either code a print function to handle the three different forms or we would write three different functions (printExpenseForm, printLoanForm, printPurchaseForm).

In C++ this can be achieved far more elegantly as follows:

```
class   Form   {
public:
   virtual void print();
};
class ExpenseForm : public Form {
public:
   virtual void print();
};
class LoanForm : public Form {
public:
   virtual void print();
};
class PurchaseForm : public Form {
public:
   virtual void print();
};
```

Each of these overridden functions is implemented so that each form prints correctly. Now an application using form objects can do this:

```
Form* pForm[10]
//create Expense/Loan/Purchase Forms...
for (short i=0 ; i < 9 ; i++)
   pForm->print();
```

Here we create ten objects that might be any combination of Expense, Loan, and Purchase Forms. However, because we are dealing with pointers to the base class, **Form**, we do not need to know which sort of form object we have; the correct **print** method is called automatically.

Limited polymorphic behavior is available in the Foundation Classes. Three virtual functions are defined in the base class **IccResource**:

```
virtual void clear();
virtual const IccBuf& get();
virtual void put(const IccBuf& buffer);
```

These methods have been implemented in the subclasses of **IccResource** wherever possible:

Class	clear	get	put
IccConsole			✓
IccDataQueue	✓	✓	✓
IccJournal			✓
IccSession		✓	✓
IccTempStore	✓	✓	✓
IccTerminal	✓	✓	✓

These virtual methods are ***not*** supported by any subclasses of **IccResource** except those in the table above.

Note: The default implementations of **clear**, **get**, and **put** in the base class **IccResource** throw an exception to prevent the user from calling an unsupported method.

Example of polymorphic behavior

The following sample can be found in the samples directory (see Sample source code˝ on page 6) as file ICC$RES2. It is presented here without the terminal IO requests.

```
#include icceh.hpp
#include iccmain.hpp
char* dataItems[] =
{
  Hello World - item 1,
  Hello World - item 2,
  Hello World - item 3
};
void IccUserControl::run()
{
```

Here we include Foundation Class headers and the **main** function. **dataItems** contains some sample data items. We write our application code in the **run** method of **IccUserControl** class.

```
  IccBuf buffer( 50 );
  IccResource* pObj[2];
```

We create an **IccBuf** object (50 bytes initially) to hold our data items. An array of two pointers to **IccResource** objects is declared.

```
pObj[0] = new IccDataQueue(ICCQ);
pObj[1] = new IccTempStore(ICCTEMPS);
```

We create two objects whose classes are derived from **IccResource** – **IccDataQueue** and **IccTempStore**.

```
for ( short index=0; index <= 1 ; index++ )
{
  pObj[index]->clear();
}
```

For both objects we invoke the **clear** method. This is handled differently by each object in a way that is transparent to the application program; this is polymorphic behavior.

```
for ( index=0; index <= 1 ; index++ )
{
  for (short j=1 ; j <= 3 ; j++)
  {
    buffer = dataItems[j-1];
    pObj[index]->put( buffer );
  }
}
```

Now we **put** three data items in each of our resource objects. Again the **put** method responds to the request in a way that is appropriate to the object type.

```
  for ( index=0; index <= 1 ; index++ )
  {
    buffer = pObj[index]->get();
    while (pObj[index]->condition() == IccCondition::NORMAL)
    {
      buffer = pObj[index]->get();
    }
    delete pObj[index];
  }
  return;
}
```

The data items are read back in from each of our resource objects using the **get** method. We delete the resource objects and return control to CICS.

Storage management

C++ objects are usually stored on the stack or heap– see Creating an object” on page 15. Objects on the stack are automatically destroyed when they go out of scope, but objects on the heap are not.

Many of the objects that the CICS Foundation Classes create internally are created on the heap rather than the stack. This can cause a problem in some CICS server environments.

On CICS Transaction Server for OS/390, CICS and Language Environment® manage ***all*** task storage so that it is released at task termination (normal or abnormal).

In a CICS for OS/2 or CICS for AIX environment, as in the earlier Technology Releases for those platforms, storage allocated on the heap is ***not*** automatically released at task termination. This can lead to memory leaks if the application programmer forgets to explicitly delete an object on the heap, or, more seriously, if the task abends.

This problem has been overcome in the CICS Foundation Classes by providing operators **new** and **delete** in the base Foundation Class, **IccBase**. These can be configured to map dynamic storage allocation requests to CICS task storage, so that ***all*** storage is automatically released at task termination. The disadvantage of this approach is a performance hit as the Foundation Classes typically issue a large number of small storage allocation requests rather than a single, larger allocation request.

This facility is affected by the **Icc::initializeEnvironment** call that must be issued before using the Foundation Classes. (This function is called from the default **main** function—see Chapter 64, main function," on page 275.)

The first parameter passed to the **initializeEnvironment** function is an enumeration that takes one of these three values:

cmmDefault
: The default action is platform dependent:

  MVS/ESA
  : same as cmmNonCICS - see below.

  UNIX same as cmmCICS - see below.

  OS/2 same as cmmCICS - see below.

cmmNonCICS
: The **new** and **delete** operators in class **IccBase** ***do not*** map dynamic storage allocation requests to CICS task storage; instead the C++ default **new** and **delete** operators are invoked.

cmmCICS
: The **new** and **delete** operators in class **IccBase** map dynamic storage allocation requests to CICS task storage (which is automatically released at normal or abnormal task termination).

The default **main** function supplied with the Foundation Classes calls **initializeEnvironment** with an enum of cmmDefault. You can change this in your program without changing the supplied header file ICCMAIN as follows:

```
#define ICC_CLASS_MEMORY_MGMT Icc::cmmNonCICS
#include iccmain.hpp
```

Alternatively, set the option **DEV(ICC_CLASS_MEMORY_MGMT)** when compiling.

Parameter passing conventions

The convention used for passing objects on Foundation Classes method calls is as follows:

> If the object is mandatory, pass by reference; if it is optional pass by pointer.

For example, consider method **start** of class **IccStartRequestQ**, which has the following signature:

```
const IccRequestId& start( const IccTransId& transId,
                           const IccTime* time=0,
                           const IccRequestId* reqId=0 );
```

Using the above convention, we see that an **IccTransId** object is mandatory, while an **IccTime** and an **IccRequestId** object are both optional. This enables an application to use this method in any of the following ways:

```
IccTransId      trn(ABCD);
IccTimeInterval int(0,0,5);
IccRequestId    req(MYREQ);
IccStartRequestQ* startQ = startRequestQ();
startQ->start( trn );
startQ->start( trn, &int );
startQ->start( trn, &int, &req );
startQ->start( trn, 0, &req );
```

Scope of data in IccBuf reference returned from read methods

Many of the subclasses of **IccResource** have read methods that return **const IccBuf** references; for example, **IccFile::readRecord**, **IccTempStore::readItem** and **IccTerminal::receive**.

Care should be taken if you choose to maintain a reference to the **IccBuf** object, rather than copy the data from the **IccBuf** reference into your own **IccBuf** object. For example, consider the following

```
IccBuf       buf(50);
IccTempStore store(TEMPSTOR);
buf = store.readNextItem();
```

Here, the data in the **IccBuf** reference returned from **IccTempStore::readNextItem** is ***immediately*** copied into the application's own **IccBuf** object, so it does not matter if the data is later invalidated. However, the application might look like this

```
IccTempStore store(TEMPSTOR);
const IccBuf& buf = store.readNextItem();
```

Here, the **IccBuf** reference returned from **IccTempStore::readNextItem** is ***not*** copied into the application's own storage and care must therefore be taken.

Note: You are recommended not to use this style of programming to avoid using a reference to an **IccBuf** object that does not contain valid data.

The returned **IccBuf** reference typically contains valid data until one of the following conditions is met:

- Another read method is invoked on the **IccResource** object (for example, another **readNextItem** or **readItem** method in the above example).
- The resource updates are committed (see method **IccTask::commitUOW**).
- The task ends (normally or abnormally).

Part 3. Foundation Classes—reference

This part contains the reference information on the Foundation Classes and structures that are provided as part of CICS. The classes and structures are arranged in alphabetic order. All the functionality you require to create object-oriented CICS programs is included within these classes and structures.

All of the classes and structures begin with the unique prefix **Icc**. You are advised not to create your own classes with this prefix.

Icc structure contains some functions and enumerations that are widely applicable. **IccValue** structure consists of a large enumeration of all the CVDA values used in traditional CICS programs.

The description of each class starts with a simple diagram that shows how it is derived from **IccBase** class, the basis of all the other classes. This is followed by a short description and an indication of the name of the header file that includes it and, where appropriate, a sample source file that uses it.

Within each class or structure description are, where appropriate, the following sections:

1. Inheritance diagram
2. Brief description of class
3. Header file where class is defined. For the location of the C++ header files on your system see Header files" on page 5.
4. Sample program demonstrating class. For the location of the supplied C++ sample programs on your system see Sample source code" on page 6.
5. Icc... constructors
6. Public methods (in alphabetic order)
7. Protected methods (in alphabetic order)
8. Inherited public methods (in tabular form)
9. Inherited protected methods (in tabular form)
10. Enumerations

Methods, including constructors, start with a formal function prototype that shows what a call returns and what the parameters are. There follows a description, in order, of the parameters. To avoid duplication, inherited methods just have an indication of the class from which they are derived (and where they are described).

The convention for names is:

1. Variable names are shown as *variable*.
2. Names of classes, structures, enumerations and methods are shown as **method**
3. Members of enumerations are shown as enumMember.
4. The names of all the supplied classes and structures begin with **Icc**.
5. Compound names have no separators, but have capital letters to demark the beginning of second and subsequent words, as in **IccJournalTypeId**.
6. Class and structure names and enumeration types begin with capital letters. Other names begin with lower case letters.

For further information on how to use these classes, see Part 2, Using the CICS foundation classes," on page 13.

Chapter 11. Icc structure

This structure holds global enumerations and functions for the CICS Foundation Classes. These globals are defined within this structure to avoid name conflicts.

Header file: ICCGLBEH

Functions

boolText

static const char* boolText (Bool *test***,**
BoolSet *set* **=** trueFalse**)**

test
A boolean value, defined in this structure, that has one of two values, chosen from a set of values given by *set*.

set
An enumeration, defined in this structure, that indicates from which pair of values *test* is selected. The default is to use true and false.

Returns the text that represents the boolean value described by the parameters, such as yes or on.

catchException

static void catchException(IccException& *exception***)**

exception
A reference to an **IccException** object that holds information about a particular type of exception.

This is the function of last resort, used to intercept **IccException** objects that the application fails to catch. It can be called from the **main** function in the stub program, listed in ICCMAIN header file, and described in Chapter 64, main function," on page 275. All OO CICS programs should use this stub or a close equivalent.

conditionText

static const char* conditionText(IccCondition::Codes *condition***)**

condition
An enumeration, defined in the **IccCondition** structure, that indicates the condition returned by a call to CICS.

Returns the symbolic name associated with a condition value. For example, if **conditionText** is called with *condition* of IccCondition::NORMAL, it returns NORMAL, if it is called with *condition* of IccCondition::IOERR, it returns IOERR, and so on.

initializeEnvironment

static void initializeEnvironment (ClassMemoryMgmt *mem* **=** cmmDefault**,**
FamilySubset *fam* **=** fsDefault**,**
Icc::Bool *EDF***)**

mem
An enumeration, defined in this structure, that indicates the memory management policy for the foundation classes.

fam
An enumeration, defined in this structure, that indicates whether the use of CICS features that are not available on all platforms is permitted.

EDF
A boolean that indicates whether EDF tracing is initially on.

Initializes the CICS Foundation Classes. The rest of the class library can only be called after this function has been called. It is called from the **main** function in the stub program, listed in ICCMAIN header file, and described in Chapter 64, main function," on page 275. All OO CICS programs should use this stub or a close equivalent.

isClassMemoryMgmtOn

static Bool isClassMemoryMgmtOn()

Returns a boolean value, defined in this structure, that indicates whether class memory management is on.

isEDFOn

static Bool isEDFOn()

Returns a Boolean value, defined in this structure, that indicates whether EDF tracing is on at the global level. (See **setEDF** in this structure, **isEDFOn** and **setEDF** in **IccResource** class on page 177 and **Execution Diagnostic Facility˝ on page 50**).

isFamilySubsetEnforcementOn

static Bool isFamilySubsetEnforcementOn()

Returns a boolean value, defined in this structure, that indicates whether it is permitted to use CICS features that are not available on all platforms.

returnToCICS

static void returnToCICS()

This call returns the program flow to CICS. It is called by the **main** function in the stub program, listed in ICCMAIN header file, and described in Chapter 64, main function," on page 275. All OO CICS programs should use this stub or a close equivalent.

setEDF

static void setEDF(Icc::Bool *onOff* **=** off**)**

onOff
: A boolean, defined in this structure, that indicates whether EDF tracing is enabled. As EDF is more suitable for tracing programs that use EXEC CICS calls than object oriented programs, the default is off.

Sets EDF tracing on or off at the global level.

unknownException

static void unknownException()

This function is called by the **main** function in ICCMAIN header file (see Chapter 64, main function," on page 275) and is used to intercept unknown exceptions. (See also **catchException** in this structure).

Enumerations

Note: References in this section to other CICS platforms—CICS OS/2 and CICS for AIX—are included for completeness. There have been Technology Releases of the CICS Foundation Classes on those platforms.

Bool

Three equivalent pairs of boolean values:
- true, yes, on
- false, no, off

true, yes, and on evaluate to 1, while false, no, and off evaluate to zero. Thus you can code test functions as follows:

```
if (task()->isStartDataAvailable())
{
    //do something
}
```

Note: 'true' and 'false' are compiler keywords in the z/OS 1.2 C/C++ compiler and will not be generated by ICCGLBEH when using this compiler, or any later version.

BoolSet

- trueFalse
- yesNo
- onOff

ClassMemoryMgmt

cmmDefault
: The defaults for the different platforms are:
  MVS/ESA
  : cmmNonCICS
  OS/2 cmmCICS
  UNIX cmmCICS

cmmNonCICS
The C++ environment performs the memory management required by the program.

In MVS/ESA Language Environment ensures that the storage for CICS tasks is released at the end of the task, or if the task terminates abnormally.

On CICS for AIX or CICS for OS/2 dynamic storage release does not occur at normal or abnormal task termination. This means that programs are susceptible to memory leaks.

cmmCICS
The **new** and **delete** operators defined in **IccBase** class map storage allocations to CICS; storage is automatically released at task termination.

FamilySubset

fsDefault
The defaults for the different platforms are all the same: fsAllowPlatformVariance

fsEnforce
Enforces Family Subset conformance; that is, it disallows use of any CICS features that are not available on ***all*** CICS servers (OS/2, AIX, and MVS/ESA).

fsAllowPlatformVariance
Allows each platform to access all the CICS features available on that platform.

GetOpt

This enumeration is used on a number of methods throughout the classes.

It indicates whether the value held internally by the object is to be returned to the caller, or whether it has to be refreshed from CICS first.

object
If the value has been previously retrieved from CICS and stored within the object, return this stored value. Otherwise, get a copy of the value from CICS and store within the object.

CICS
Force the object to retrieve a fresh value from CICS (and store it within the object) even if there is already a value stored within the object from a previous invocation.

Platforms

Indicates on which operating system the program is being run. Possible values are:

OS2
UNIX
MVS

Chapter 12. IccAbendData class

IccBase
    IccResource
        IccAbendData

This is a singleton class used to retrieve diagnostic information from CICS about a program abend.

Header file: ICCABDEH

IccAbendData constructor (protected)

Constructor

IccAbendData()

Public methods

> **The opt parameter**
> Many methods have the same parameter, *opt*, which is described under the **abendCode** method.

abendCode

const char* abendCode(Icc::GetOpt *opt* **=** Icc::object**)**

opt
    An enumeration, defined in the **Icc** structure, that indicates whether a value should be refreshed from CICS or whether the existing value should be retained. The possible values are described under the **GetOpt** enumeration in the **Icc** structure on pageGetOpt" on page 70.

Returns the current 4-character abend code.

Conditions

INVREQ

ASRAInterrupt

const char* ASRAInterrupt(Icc::GetOpt *opt* **=** Icc::object**)**

Returns 8 characters of status word (PSW) interrupt information at the point when the latest abend with a code of ASRA, ASRB, ASRD, or AICA occurred.

The field contains binary zeroes if no ASRA or ASRB abend occurred during the execution of the issuing transaction, or if the abend originally occurred in a remote DPL server program.

Conditions

INVREQ

ASRAKeyType

IccValue::CVDA ASRAKeyType(Icc::GetOpt *opt* **=** Icc::object**)**

Returns an enumeration, defined in **IccValue**, that indicates the execution key at the time of the last ASRA, ASRB, AICA, or AEYD abend, if any. The possible values are:

CICSEXECKEY
The task was executing in CICS-key at the time of the last ASRA, ASRB, AICA, or AEYD abend. Note that all programs execute in CICS key if CICS subsystem storage protection is not active.

USEREXECKEY
The task was executing in user-key at the time of the last ASRA, ASRB, AICA, or AEYD abend. Note that all programs execute in CICS key if CICS subsystem storage protection is not active.

NONCICS
The execution key at the time of the last abend was not one of the CICS keys; that is, not key 8 or key 9.

NOTAPPLIC
There has not been an ASRA, ASRB, AICA, or AEYD abend.

Conditions

INVREQ

ASRAPSW

const char* ASRAPSW(Icc::GetOpt *opt* **=** Icc::object**)**

Returns an 8-character status word (PSW) at the point when the latest abend with a code of ASRA, ASRB, ASRD, or AICA occurred.

The field contains nulls if no ASRA, ASRB, ASRD, or AICA abend occurred during the execution of the issuing transaction, or if the abend originally occurred in a remote DPL server.

Conditions

INVREQ

ASRARegisters

const char* ASRARegisters(Icc::GetOpt *opt* **=** Icc::object**)**

Returns the contents of general registers 0–15, as a 64-byte data area, at the point when the latest ASRA, ASRB, ASRD, or AICA abend occurred. The contents of the registers are returned in the order 0, 1, ..., 15.

Note that nulls are returned if no ASRA, ASRB, ASRD, or AICA abend occurred during the execution of the issuing transaction, or if the abend originally occurred in a remote DPL server program.

Conditions

INVREQ

ASRASpaceType

IccValue::CVDA ASRASpaceType(Icc::GetOpt *opt* **=** Icc::object**)**

Returns an enumeration, defined in **IccValue** structure, that indicates what type of space, if any, was in control at the time of the last ASRA, ASRB, AICA, or AEYD abend. Possible values are:

SUBSPACE
The task was executing in either its own subspace or the common subspace at the time of the last ASRA, ASRB, AICA, or AEYD abend.

BASESPACE
The task was executing in the base space at the time of the last ASRA, ASRB, AICA, or AEYD abend. Note that all tasks execute in the base space if transaction isolation is not active.

NOTAPPLIC
There has not been an ASRA, ASRB, AICA, or AEYD abend.

Conditions

INVREQ

ASRAStorageType

IccValue::CVDA ASRAStorageType(Icc::GetOpt *opt* **=** Icc::object**)**

Returns an enumeration, defined in **IccValue** structure, that indicates what type of storage, if any, was being addressed at the time of the last ASRA, ASRB, AICA, or AEYD abend. Possible values are:

CICS CICS-key storage is being addressed. This can be in one of the CICS dynamic storage areas (CDSA or ECDSA), or in one of the read-only dynamic storage areas (RDSA or ERDSA) if either of the following apply:

- CICS is running with the NOPROTECT option on the RENTPGM system initialization parameter
- storage protection is not active

USER
User-key storage in one of the user dynamic storage areas (RDSA or ERDSA) is being addressed.

READONLY
Read-only storage in one of the read-only dynamic storage areas (RDSA or ERDSA) when CICS is running with the PROTECT option on the RENTPGM system initialization parameter.

NOTAPPLIC
One of:

- No ASRA or AEYD abend has been found for this task.
- The storage affected by an abend is not managed by CICS.
- The ASRA abend is not caused by a 0C4 abend.
- An ASRB or AICA abend has occurred since the last ASRA or AEYD abend.

Conditions

INVREQ

instance

static IccAbendData* instance()

Returns a pointer to the single **IccAbendData** object. If the object does not already exist, it is created by this method.

isDumpAvailable

Icc::Bool isDumpAvailable(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean, defined in **Icc** structure, that indicates whether a dump has been produced. If it has, use **programName** method to find the name of the failing program of the latest abend.

Conditions

INVREQ

originalAbendCode

const char* originalAbendCode(Icc::GetOpt *opt* **=** Icc::object**)**

Returns the original abend code for this task in case of repeated abends.

Conditions

INVREQ

programName

const char* programName(Icc::GetOpt *opt* **=** Icc::oldValue**)**

Returns the name of the program that caused the abend.

Conditions

INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource

Method	Class
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 13. IccAbsTime class

IccBase
  IccResource
    IccTime
      IccAbsTime

This class holds information about absolute time, the time in milliseconds that has elapsed since the beginning of the year 1900.

Header file: ICCTIMEH

IccAbsTime constructor

Constructor (1)

IccAbsTime(const char* *absTime***)**

absTime
The 8-byte value of time, in packed decimal format.

Constructor (2)

IccAbsTime(const IccAbsTime& *time***)**

The copy constructor.

Public methods

date

const char* date (IccClock::DateFormat *format* **=** IccClock::defaultFormat**,**
char *dateSeparator* **= '\0')**

format
An enumeration, defined in **IccClock** class, that indicates the format of the date. The default is to use the installation default, the value set when the CICS region is initialized.

dateSeparator
The character that separates the different fields of the date The default is no separation character.

Returns the date, as a character string.

Conditions

INVREQ

dayOfMonth

unsigned long dayOfMonth()

Returns the day of the month in the range 1 to 31.

Conditions
INVREQ

dayOfWeek

IccClock::DayOfWeek dayOfWeek()

Returns an enumeration, defined in **IccClock** class, that indicates the day of the week.

Conditions
INVREQ

daysSince1900

unsigned long daysSince1900()

Returns the number of days that have elapsed since the first day of 1900.

Conditions
INVREQ

hours

virtual unsigned long hours() const

Returns the hours component of the time.

milliSeconds

long double milliSeconds()

Returns the number of milliseconds that have elapsed since the first day of 1900.

minutes

virtual unsigned long minutes() const

Returns the minutes component of the time.

monthOfYear

IccClock::MonthOfYear monthOfYear()

Returns an enumeration, defined in **IccClock** class, that indicates the month of the year.

Conditions
INVREQ

operator=

IccAbsTime& operator=(const IccAbsTime& *absTime***)**

Assigns one **IccAbsTime** object to another.

packedDecimal

const char* packedDecimal() const

Returns the time as an 8-byte packed decimal string that expresses the number of milliseconds that have elapsed since the beginning of the year 1900.

seconds

virtual unsigned long seconds() const

Returns the seconds component of the time.

time

const char* time(char *timeSeparator* **= '\0')**

timeSeparator
: The character that delimits the time fields. The default is no time separation character.

Returns the time as a text string.

Conditions

INVREQ

timeInHours

unsigned long timeInHours()

Returns the number of hours that have elapsed since the day began.

timeInMinutes

unsigned long timeInMinutes()

Returns the number of minutes that have elapsed since the day began.

timeInSeconds

unsigned long timeInSeconds()

Returns the number of seconds that have elapsed since the day began.

year

unsigned long year()

Returns the year as a 4-digit integer, e.g. 1996.

Conditions

INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
hours	IccTime
isEDFOn	IccResource
minutes	IccTime
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource
timeInHours	IccTime
timeInMinutes	IccTime
timeInSeconds	IccTime
type	IccTime

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 14. IccAlarmRequestId class

IccBase
  IccResourceId
    IccRequestId
      IccAlarmRequestId

An **IccAlarmRequestId** object represents a unique alarm request. It contains the 8-character name of the request identifier and a pointer to a 4-byte timer event control area. **IccAlarmRequestId** is used by the **setAlarm** method of **IccClock** class when setting an alarm, and the **waitOnAlarm** method of **IccTask** when waiting for an alarm.

Header file: ICCRIDEH

IccAlarmRequestId constructors

Constructor (1)

IccAlarmRequestId()

Creates a new object with no information present.

Constructor (2)

IccAlarmRequestId (const char* *nam*,
  const void* *timerECA***)**

name
  The 8-character name of the request.

timerECA
  A pointer to a 4-byte timer event control area.

Creates an object with information already set.

Constructor (3)

IccAlarmRequestId(const IccAlarmRequestId& *id***)**

id
  A reference to an **IccAlarmRequestId** object.

The copy constructor.

Public methods

isExpired

Icc::Bool isExpired()

Returns a boolean, defined in **Icc** structure, that indicates whether the alarm has expired.

operator= (1)

IccAlarmRequestId& operator=(const IccRequestId& *id***)**

id
A reference to an **IccRequestId** object.

operator= (2)

IccAlarmRequestId& operator=(const IccAlarmRequestId& *id***)**

id
A reference to an **IccAlarmRequestId** object.

operator= (3)

IccAlarmRequestId& operator=(const char* *requestName***)**

requestName
The 8-character name of the alarm request.

These methods are used to copy information into an **IccAlarmRequestId** object.

setTimerECA

void setTimerECA(const void* *timerECA***)**

timerECA
A pointer to a 4-byte timer event control area.

timerECA

const void* timerECA() const

Returns a pointer to the 4-byte timer event control area.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase

Method	Class
setCustomClassNum	IccBase

Chapter 15. IccBase class

IccBase

IccBase class is the base class from which ***all*** CICS Foundation Classes are derived. (The methods associated with **IccBase** are described here although, in practice, they can only be called on objects of the derived classes).

Header file: ICCBASEH

IccBase constructor (protected)

Constructor

IccBase(ClassType *type***)**

type
: An enumeration that indicates what the subclass type is. For example, for an **IccTempStore** object, the class type is cTempStore.

Public methods

The opt parameter

Many methods have the same parameter, *opt*, which is described under the **abendCode** method inabendCode” on page 71.

classType

ClassType classType() const

Returns an enumeration that indicates what the subclass type is. For example, for an **IccTempStore** object, the class type is cTempStore. The possible values are listed under **ClassType** on page 87.

className

const char* className(NameOpt *opt***=customName)**

opt
: An enumerator, defined in this class, that indicates whether to return the base name of the class or the name as customized by a derived class.

Returns the name of the class. For example, an **IccTempStore** object returns IccTempStore.

Suppose a class **MyDataQueue** inherits from **IccDataQueue**. If **MyDataQueue** calls **setClassName("MyDataQueue")**, **MyDataQueue::className(IccBase::customName)** returns "MyDataQueue" and **MyDataQueue::className(IccBase::baseName)** returns "IccDataQueue". An **IccDataQueue** object returns "IccDataQueue" for both *opt* values.

customClassNum

unsigned short customClassNum() const

Returns the number that an application designer has associated with a subclass that he or she has designed.

operator delete

void operator delete(void* *object***)**

object
A pointer to an object that is to be destroyed.

Destroys an object in an orderly manner.

operator new

void* operator new(size_t *size***)**

size
The size of the object that is to be created, in bytes.

Creates a new object of given size. This operator enables the Foundation Classes to use CICS storage allocation (see initializeEnvironment˝ on page 67).

Protected methods

setClassName

void setClassName(const char* *className***)**

className
The name of the class. For example, if you create a class **MyTempStore** that is a specialization of **IccTempStore**, you might call **setClassName(MyTempStore)**.

Sets the name of the class. It is useful for diagnostic purposes to be able to get a string representation of the name of the class to which an object belongs.

setCustomClassNum

void setCustomClassNum(unsigned short *number***)**

number
The number that an application designer associates with a subclass for identification purposes.

Assigns an identification number to a subclass that is not an original part of the classes, as supplied.

Enumerations

ClassType

The names are derived by deleting the first two characters from the name of the class. The possible values are:

cAbendData	cGroupId	cSystem
cAlarmRequestId	cJournal	cTask
cBuf	cJournalId	cTempStore
cClock	cJournalTypeId	cTempStoreId
cConsole	cLockId	cTermId
cControl	cMessage	cTerminal
cConvId	cPartnerId	cTerminalData
cCUSTOM	cProgram	cTime
cDataQueue	cProgramId	cTPNameId
cDataQueueId	cRecordIndex	cTransId
cEvent	cRequestId	cUser
cException	cSemaphore	cUserId
cFile	cSession	
cFileId	cStartRequestQ	
cFileIterator	cSysId	

Note: cCUSTOM allows the class library to be extended by non-IBM developers.

NameOpt

SeeclassName” on page 85.

baseName
Returns the default name assigned to the class as provided by IBM.

customName
Returns the name assigned using **setClassName** method from a subclass *or*, if **setClassName** has not been invoked, the same as *baseName*.

Chapter 16. IccBuf class

IccBase
  IccBuf

IccBuf class is supplied for the general manipulation of buffers. This class is used by other classes that make calls to CICS, but does not itself call CICS services. See Chapter 6, Buffer objects," on page 25.

Header file: ICCBUFEH

Sample: ICC$BUF

IccBuf constructors

Constructor (1)

IccBuf (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 that indicates whether the data area can be dynamically extended. Possible values are extensible or fixed. The default is extensible.

Creates an **IccBuf** object, allocating its own data area with the given length and with all the bytes within it set to NULL.

Constructor (2)

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

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

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

Creates an **IccBuf** object that cannot be extended, adopting the given data area as its own.

See warning about Internal/External ownership of buffers˝ on page 25.

Constructor (3)

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

text
: A null-terminated string to be copied into the new **IccBuf** object.

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

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

Constructor (4)

IccBuf(const IccBuf& *buffer***)**

buffer
: A reference to an **IccBuf** object that is to be copied into the new object.

The copy constructor—creates a new **IccBuf** object that is a copy of the given object. The created **IccBuf** object ***always*** has an internal data area.

Public methods

append (1)

IccBuf& append (unsigned long *length***,**
const void* *dataArea***)**

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

dataArea
: The address of the source data area.

Appends data from the given data area to the data area in the object.

append (2)

IccBuf& append (const char* *format***,**
...)

format
: The null-terminated format string

...
: The optional parameters.

Append data, in the form of format string and variable argument, to the data area in the object. This is the same as the form used by **printf** in the standard C library. Note that it is the responsibility of the application programmer to ensure that the optional parameters are consistent with the format string.

assign (1)

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

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

dataArea
: The address of the source data area.

Assigns data from the given data area to the data area in the object.

assign (2)

IccBuf& assign (const char* *format***,**
...)

format
The format string

...
The optional parameters.

Assigns data, in the form of format string and variable argument, to the data area in the object. This is the same as the form used by **printf** in the standard C library.

cut

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

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

offset
The offset into the data area. The default is no offset.

Makes the specified cut to the data in the data area and returns a reference to the **IccBuf** object.

dataArea

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

offset
The offset into the data area. The default is no offset.

Returns the address of data at the given offset into the data area.

dataAreaLength

unsigned long dataAreaLength() const

Returns the length of the data area in bytes.

dataAreaOwner

DataAreaOwner dataAreaOwner() const

Returns an enumeration that indicates whether the data area has been allocated by the **IccBuf** constructor or has been supplied from elsewhere. The possible values are listed under DataAreaOwner” on page 97.

dataAreaType

DataAreaType dataAreaType() const

Returns an enumeration that indicates whether the data area can be extended. The possible values are listed under DataAreaType” on page 97.

dataLength

unsigned long dataLength() const

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

insert

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

length
The length of the data, in bytes, to be inserted into the **IccBuf** object

dataArea
The start of the source data to be inserted into the **IccBuf** object

offset
The offset in the data area where the data is to be inserted. The default is no offset.

Inserts the given data into the data area at the given offset and returns a reference to the **IccBuf** object.

isFMHContained

Icc::Bool isFMHContained() const

Returns a boolean, defined in **Icc** structure, that indicates whether the data area contains FMHs (function management headers).

operator const char*

operator const char*() const

Casts an **IccBuf** object to a null terminated string.

```
IccBuf data(Hello World);
cout << (const char*) data;
```

operator= (1)

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

buffer
A reference to an **IccBuf** object.

Assigns data from another buffer object and returns a reference to the **IccBuf** object.

operator= (2)

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

text
The null-terminated string to be assigned to the **IccBuf** object.

Assigns data from a null-terminated string and returns a reference to the **IccBuf** object.

See also the **assign** method.

operator+= (1)

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

buffer
A reference to an **IccBuf** object.

Appends data from another buffer object and returns a reference to the **IccBuf** object.

operator+= (2)

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

text
The null-terminated string to be appended to the **IccBuf** object.

Appends data from a null-terminated string and returns a reference to the **IccBuf** object.

See also the **append** method.

operator==

Icc::Bool operator==(const IccBuf& *buffer***) const**

buffer
A reference to an **IccBuf** object.

Returns a boolean, defined in **Icc** structure, that indicates whether the data contained in the buffers of the two **IccBuf** objects is the same. It is true if the current lengths of the two data areas are the same and the contents are the same.

operator!=

Icc::Bool operator!=(const IccBuf& *buffer***) const**

buffer
A reference to an **IccBuf** object.

Returns a boolean, defined in **Icc** structure, that indicates whether the data contained in the buffers of the two **IccBuf** objects is different. It is true if the current lengths of the two data areas are different or if the contents are different.

operator<< (1)

operator<<(const IccBuf& *buffer***)**

Appends another buffer.

operator<< (2)

operator<<(const char* *text*)

Appends a string.

operator<< (3)

operator<<(char *ch*)

Appends a character.

operator<< (4)

operator<<(signed char *ch*)

Appends a character.

operator<< (5)

operator<<(unsigned char *ch*)

Appends a character.

operator<< (6)

operator<<(const signed char* *text*)

Appends a string.

operator<< (7)

operator<<(const unsigned char* *text*)

Appends a string.

operator<< (8)

operator<<(short *num*)

Appends a short.

operator<< (9)

operator<<(unsigned short *num*)

Appends an unsigned short.

operator<< (10)

operator<<(long *num*)

Appends a long.

operator<< (11)

operator<<(unsigned long *num***)**

Appends an unsigned long.

operator<< (12)

operator<<(int *num***)**

Appends an integer.

operator<< (13)

operator<<(float *num***)**

Appends a float.

operator<< (14)

operator<<(double *num***)**

Appends a double.

operator<< (15)

operator<<(long double *num***)**

Appends a long double.

Appends data of various types to the **IccBuf** object. The types are converted to a readable format, for example from a long to a string representation.

overlay

IccBuf& overlay (unsigned long *length***,**
void* *dataArea***)**

length
: The length of the existing data area.

dataArea
: The address of the existing data area.

Makes the data area external and fixed. Any existing internal data area is destroyed.

See warning about Internal/External ownership of buffers˝ on page 25.

replace

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

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

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

offset
The position where the new data is to be written, relative to the start of the **IccBuf** data area. The default is no offset.

Replaces the current contents of the data area at the given offset with the data provided and returns a reference to the **IccBuf** object.

setDataLength

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

length
The new length of the data area, in bytes

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

setFMHContained

void setFMHContained(Icc::Bool *yesNo* **=** Icc::yes**)**

yesNo
A boolean, defined in **Icc** structure, that indicates whether the data area contains FMHs. The default value is yes.

Allows an application program to indicate that a data area contains function management headers.

Inherited public methods

Method	**Class**
className	IccBase
classType	IccBase
customClassNum	IccBase
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	**Class**
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

DataAreaOwner

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

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

external
: The data area has been allocated externally.

DataAreaType

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

extensible
: The data area can be automatically extended to accommodate more data.

fixed
: The data area cannot grow in size. If you attempt to assign too much data, the data is truncated, and an exception is thrown.

Chapter 17. IccClock class

IccBase
  IccResource
    IccClock

The **IccClock** class controls access to the CICS time and date services.

Header file: ICCCLKEH

Sample: ICC$CLK

IccClock constructor

Constructor

IccClock(UpdateMode *update* **=** manual**)**

update
: An enumeration, defined in this class, that indicates whether the clock is to update its time automatically whenever a time or date service is used, or whether it is to wait until an explicit **update** method call is made. If the time is updated manually, the initial clock time is the time when the **IccClock object** object is created.

Public methods

absTime

IccAbsTime& absTime()

Returns a reference to an **IccAbsTime** object that contains the absolute time as provided by CICS.

cancelAlarm

void cancelAlarm(const IccRequestId* *reqId* **= 0)**

reqId
: An optional pointer to the **IccRequestId** object that holds information on an alarm request.

Cancels a previous **setAlarm** request if the alarm time has not yet been reached, that is, the request has not expired.

Conditions

ISCINVREQ, NOTAUTH, NOTFND, SYSIDERR

date

const char* date (DateFormat *format* **=** defaultFormat**,**
  char *dateSeparator* **= '\0')**

format
An enumeration, defined in this class, that indicates in which format you want the date to be returned.

dateSeparator
The character that is used to separate different fields in the date. The default is no separation character.

Returns the date as a string.

Conditions

INVREQ

dayOfMonth

unsigned long dayOfMonth()

Returns the day component of the date, in the range 1 to 31.

Conditions

INVREQ

dayOfWeek

DayOfWeek dayOfWeek()

Returns an enumeration, defined in this class, that indicates the day of the week.

Conditions

INVREQ

daysSince1900

unsigned long daysSince1900()

Returns the number of days that have elapsed since 1st January, 1900.

Conditions

INVREQ

milliSeconds

long double milliSeconds()

Returns the number of milliseconds, rounded to the nearest hundredth of a second, that have elapsed since 00:00 on 1st January, 1900.

monthOfYear

MonthOfYear monthOfYear()

Returns an enumeration, defined in this class, that indicates the month of the year.

Conditions

INVREQ

setAlarm

const IccAlarmRequestId& setAlarm (const IccTime& *time*,
const IccRequestId* *reqId* **= 0)**

time
A reference to an **IccTime** object that contains time information. As **IccTime** is an abstract class *time* is, in practise, an object of class **IccAbsTime**, **IccTimeOfDay**, or **IccTimeInterval**.

reqId
An optional pointer to an **IccRequestId** object that is used to identify this particular alarm request.

Sets an alarm at the time specified in *time*. It returns a reference to an **IccAlarmRequestId** object that can be used to cancel the alarm—see **cancelAlarm** method. See also the **waitOnAlarm** method on page 221 of class **IccTask**.

Conditions

EXPIRED, INVREQ

time

const char* time(char *timeSeparator* **= '\0')**

timeSeparator
The character that delimits the time fields. The default is no separation character.

Returns the time as a text string.

Conditions

INVREQ

update

void update()

Updates the clock time and date from CICS. See the **IccClock** constructor.

year

unsigned long year()

Returns the 4-figure year number, such as 1996.

Conditions

INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase

Method	Class
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

DateFormat

defaultFormat
DDMMYY
MMDDYY
YYDDD
YYDDMM
YYMMDD
DDMMYYYY
MMDDYYYY
YYYYDDD
YYYYDDMM
YYYYMMDD

DayOfWeek

Indicates the day of the week.

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

MonthOfYear

Indicates the month of the year.

January

February
March
April
May
June
July
August
September
October
November
December

UpdateMode

Indicates whether the clock is automatically updated.

manual
: The clock initially holds the time at which it was created. It is subsequently updated only when an **update** method call is made.

automatic
: The clock is updated to the current CICS time and date whenever any time or date method is called (for example, **daysSince1900**).

Chapter 18. IccCondition structure

This structure contains an enumeration of all the CICS condition codes.

Header file: ICCCNDEH

Enumerations

Codes

The possible values are:

	Value		**Value**		**Value**
0	NORMAL	35	TSIOERR	70	NOTAUTH
1	ERROR	36	MAPFAIL	__	
2	RDATT	37	INVERRTERM	72	SUPPRESSED
3	WRBRK	38	INVMPSZ	__	
4	ICCEOF	39	IGREQID	__	
5	EODS	40	OVERFLOW	75	RESIDERR
6	EOC	41	INVLDC	__	
7	INBFMH	42	NOSTG	__	
8	ENDINPT	43	JIDERR	__	
9	NONVAL	44	QIDERR	__	
10	NOSTART	45	NOJBUFSP	80	NOSPOOL
11	TERMIDERR	46	DSSTAT	81	TERMERR
12	FILENOTFOUND	47	SELNERR	82	ROLLEDBACK
13	NOTFND	48	FUNCERR	83	END
14	DUPREC	49	UNEXPIN	84	DISABLED
15	DUPKEY	50	NOPASSBKRD	85	ALLOCERR
16	INVREQ	51	NOPASSBKWR	86	STRELERR
17	IOERR	__		87	OPENERR
18	NOSPACE	53	SYSIDERR	88	SPOLBUSY
19	NOTOPEN	54	ISCINVREQ	89	SPOLERR
20	ENDFILE	55	ENQBUSY	90	NODEIDERR
21	ILLOGIC	56	ENVDEFERR	91	TASKIDERR
22	LENGERR	57	IGREQCD	92	TCIDERR
23	QZERO	58	SESSIONERR	93	DSNNOTFOUND
24	SIGNAL	59	SYSBUSY	94	LOADING
25	QBUSY	60	SESSBUSY	95	MODELIDERR
26	ITEMERR	61	NOTALLOC	96	OUTDESCERR
27	PGMIDERR	62	CBIDERR	97	PARTNERIDERR
28	TRANSIDERR	63	INVEXITREQ	98	PROFILEIDERR
29	ENDDATA	64	INVPARTNSET	99	NETNAMEIDERR
30	INVTSREQ	65	INVPARTN	100	LOCKED
31	EXPIRED	66	PARTNFAIL	101	RECORDBUSY
32	RETPAGE	__		102	UOWNOTFOUND
33	RTEFAIL	__		103	UOWLNOTFOUND
34	RTESOME	69	USERIDERR		

Range

maxValue
The highest CICS condition, currently 103.

Chapter 19. IccConsole class

IccBase
  IccResource
    IccConsole

This is a singleton class that represents the CICS console.

Header file: ICCCONEH

Sample: ICC$CON

IccConsole constructor (protected)

Constructor

IccConsole()

No more than one of these objects is permitted in a task. An attempt to create more objects causes an exception to be thrown.

Public methods

The opt parameter

Many methods have the same parameter, *opt*, which is described under the **abendCode** method inabendCode” on page 71.

instance

static IccConsole* instance()

Returns a pointer to the single **IccConsole** object that represents the CICS console. If the object does not already exist, it is created by this method.

put

virtual void put(const IccBuf& *send*)

send
: A reference to an **IccBuf** object that contains the data that is to be written to the console.

Writes the data in *send* to the CICS console. **put** is a synonym for **write**. See Polymorphic Behavior” on page 59.

replyTimeout

unsigned long replyTimeout() const

Returns the length of the reply timeout in milliseconds.

resetRouteCodes

void resetRouteCodes()

Removes all route codes held in the **IccConsole** object.

setAllRouteCodes

void setAllRouteCodes()

Sets all possible route codes in the **IccConsole** object, that is, 1 through 28.

setReplyTimeout (1)

void setReplyTimeout(IccTimeInterval& *interval***)**

interval
A reference to a **IccTimeInterval** object that describes the length of the time interval required.

setReplyTimeout (2)

void setReplyTimeout(unsigned long *seconds***)**

seconds
The length of the time interval required, in seconds.

The two different forms of this method are used to set the length of the reply timeout.

setRouteCodes

void setRouteCodes (unsigned short *numRoutes***,**
...)

numRoutes
The number of route codes provided in this call—the number of arguments that follow this one.

...
One or more arguments, the number of which is given by *numRoutes*. Each argument is a route code, of type **unsigned short**, in the range 1 to 28.

Saves route codes in the object for use on subsequent **write** and **writeAndGetReply** calls. Up to 28 codes can be held in this way.

write

void write (const IccBuf& *send***,**
SeverityOpt *opt* **=** none**)**

send
A reference to an **IccBuf** object that contains the data that is to be written to the console.

opt
An enumeration, defined below, that indicates the severity of the console message.

Writes the data in *send* to the CICS console.

Conditions

INVREQ, LENGERR, EXPIRED

writeAndGetReply

const IccBuf& writeAndGetReply (const IccBuf& *send*,
SeverityOpt *opt*= none**)**

send
A reference to an **IccBuf** object that contains the data that is to be written to the console.

opt
An enumeration, defined below, that indicates the severity of the console message.

Writes the data in *send* to the CICS console and returns a reference to an **IccBuf** object that contains the reply from the CICS operator.

Conditions

INVREQ, LENGERR, EXPIRED

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

SeverityOpt

Possible values are:
- none
- warning
- error
- severe

Chapter 20. IccControl class

IccBase
    IccResource
        IccControl

IccControl class controls an application program that uses the supplied Foundation Classes. This class is a singleton class in the application program; each program running under a CICS task has a single **IccControl** object.

IccControl has a pure virtual **run** method, where application code is written, and is therefore an abstract base class. The application programmer must subclass **IccControl**, and implement the **run** method.

Header file: ICCCTLEH

IccControl constructor (protected)

Constructor

IccControl()

Public methods

callingProgramId

const IccProgramId& callingProgramId()

Returns a reference to an **IccProgramId** object that represents the program that called this program. The returned **IccProgramId** reference contains a null name if the executing program was not called by another program.

Conditions

INVREQ

cancelAbendHandler

void cancelAbendHandler()

Cancels a previously established exit at this logical program level.

Conditions

NOTAUTH, PGMIDERR

commArea

IccBuf& commArea()

Returns a reference to an **IccBuf** object that encapsulates the COMMAREA—the communications area of CICS memory that is used for passing data between CICS programs and transactions.

Conditions

INVREQ

console

IccConsole* console()

Returns a pointer to the single **IccConsole** object. If this object has not yet been created, this method creates the object before returning a pointer to it.

initData

const IccBuf& initData()

Returns a reference to an **IccBuf** object that contains the initialization parameters specified for the program in the INITPARM system initialization parameter.

Conditions

INVREQ

instance

static IccControl* instance()

Returns a pointer to the single **IccControl** object. The object is created if it does not already exist.

isCreated

static Icc::Bool isCreated()

Returns a boolean value that indicates whether the **IccControl** object already exists. Possible values are true or false.

programId

const IccProgramId& programId()

Returns a reference to an **IccProgramId** object that refers to this executing program.

Conditions

INVREQ

resetAbendHandler

void resetAbendHandler()

Reactivates a previously cancelled abend handler for this logical program level. (See **cancelAbendHandler on page** 111).

Conditions

NOTAUTH, PGMIDERR

returnProgramId

const IccProgramId& returnProgramId()

Returns a reference to an **IccProgramId** object that refers to the program that resumes control when this logical program level issues a return.

run

virtual void run() = 0

This method should be implemented in a subclass of **IccControl** by the application programmer.

session

IccSession* session()

Returns a pointer to the **IccSession** object that represents the principal facility for this program. An exception is thrown if this program does not have a session as its principal facility.

setAbendHandler (1)

void setAbendHandler(const IccProgramId& *programId***)**

programId
: A reference to the **IccProgramId** object that indicates which program is affected.

setAbendHandler (2)

void setAbendHandler(const char* *programName***)**

programName
: The name of the program affected.

These methods set the abend handler to the named program for this logical program level.

Conditions

NOTAUTH, PGMIDERR

startRequestQ

IccStartRequestQ* startRequestQ()

Returns a pointer to the **IccStartRequestQ** object. If this object has not yet been created, this method creates the object before returning a pointer to it.

system

IccSystem* system()

Returns a pointer to the **IccSystem** object. If this object has not yet been created, this method creates the object before returning a pointer to it.

task

IccTask* task()

Returns a pointer to the **IccTask** object. If this object has not yet been created, this method creates the object before returning a pointer to it.

terminal

IccTerminal* terminal()

Returns a pointer to the **IccTerminal** object. If this object has not yet been created, this method creates the object before returning a pointer to it.

This method has a condition, that the transaction must have a terminal as its principal facility. That is, there must be a physical terminal involved.

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 21. IccConvId class

IccBase
  IccResourceId
    IccConvId

IccConvId class is used to identify an APPC conversation.

Header file: ICCRIDEH

IccConvId constructors

Constructor (1)

IccConvId(const char* *convName***)**

convName
  The 4-character name of the conversation.

Constructor (2)

IccConvId(const IccConvId& *convId***)**

convId
  A reference to an **IccConvId** object.

The copy constructor.

Public methods

operator= (1)

IccConvId& operator=(const char* *convName***)**

operator= (2)

IccConvId& operator=(const IccConvId *id***)**

Assigns new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 22. IccDataQueue class

IccBase
  IccResource
    IccDataQueue

This class represents a CICS transient data queue.

Header file: ICCDATEH

Sample: ICC$DAT

IccDataQueue constructors

Constructor (1)

IccDataQueue(const IccDataQueueId& *id***)**

id
: A reference to an **IccDataQueueId** object that contains the name of the CICS transient data queue.

Constructor (2)

IccDataQueue(const char* *queueName***)**

queueName
: The 4-byte name of the queue that is to be created. An exception is thrown if *queueName* is not valid.

Public methods

clear

virtual void clear()

A synonym for **empty**. See Polymorphic Behavior" on page 59.

empty

void empty()

Empties the queue, that is, deletes all items on the queue.

Conditions

ISCINVREQ, NOTAUTH, QIDERR, SYSIDERR, DISABLED, INVREQ

get

virtual const IccBuf& get()

A synonym for **readItem**. See Polymorphic Behavior" on page 59.

put

virtual void put(const IccBuf& *buffer***)**

buffer
A reference to an **IccBuf** object that contains data to be put into the queue.

A synonym for **writeItem**. See Polymorphic Behavior" on page 59.

readItem

const IccBuf& readItem()

Returns a reference to an **IccBuf** object that contains one item read from the data queue.

Conditions

IOERR, ISCINVREQ, LENGERR, NOTAUTH, NOTOPEN, QBUSY, QIDERR, QZERO, SYSIDERR, DISABLED, INVREQ

writeItem (1)

void writeItem(const IccBuf& *item***)**

item
A reference to an **IccBuf** object that contains data to be written to the queue.

writeItem (2)

void writeItem(const char* *text***)**

text
Text that is to be written to the queue.

Writes an item of data to the queue.

Conditions

IOERR, ISCINVREQ, LENGERR, NOSPACE, NOTAUTH, NOTOPEN, QIDERR, SYSIDERR, DISABLED, INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
className	IccBase
classType	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource

Method	Class
isEDFOn	IccResource
isRouteOptionOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
routeOption	IccResource
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource
setRouteOption	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 23. IccDataQueueId class

IccBase
    IccResourceId
        IccDataQueueId

IccDataQueueId is used to identify a CICS Transient Data Queue name.

Header file: ICCRIDEH

IccDataQueueId constructors

Constructor (1)

IccDataQueueId(const char* *queueName***)**

queueName
The 4-character name of the queue

Constructor (2)

IccDataQueueId(const IccDataQueueId& *id***)**

id A reference to an **IccDataQueueId** object.

Public methods

operator= (1)

IccDataQueueId& operator=(const char* *queueName***)**

queueName
The 4-character name of the queue

operator= (2)

IccDataQueueId& operator=(const IccDataQueueId& *id***)**

id A reference to an **IccDataQueueId** object.
Assigns new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 24. IccEvent class

IccBase
  IccEvent

The **IccEvent** class contains information on a particular CICS call, which we call a CICS event.

Header file: ICCEVTEH

Sample: ICC$RES1

IccEvent constructor

Constructor

IccEvent (const IccResource* *object***,**
const char* *methodName***)**

object
  A pointer to the **IccResource** object that is responsible for this event.

methodName
  The name of the method that caused the event to be created.

Public methods

className

const char* className() const

Returns the name of the class responsible for this event.

classType

IccBase::ClassType classType() const

Returns an enumeration, described under **classType** on page 85 in **IccBase** class, that indicates the type of class that is responsible for this event.

condition

IccCondition::Codes condition(IccResource::ConditionType *type* **=**
IccResource::majorCode**) const**

type
  An enumeration that indicates whether a major code or minor code is being requested. Possible values are majorCode or minorCode. majorCode is the default value.

Returns an enumerated type that indicates the condition returned from this CICS event. The possible values are described under the **Codes** type in the **IccCondition** structure.

conditionText

const char* conditionText() const

Returns the text of the CICS condition code, such as NORMAL or LENGERR.

methodName

const char* methodName() const

Returns the name of the method responsible for this event.

summary

const char* summary()

Returns a summary of the CICS event in the form:

```
CICS event summary: IccDataQueue::readItem condition=23 (QZERO) minor=0
```

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 25. IccException class

IccBase
  IccException

IccException class contains information about CICS Foundation Class exceptions. It is used to create objects that are 'thrown' to application programs. They are generally used for error conditions such as invalid method calls, but the application programmer can also request an exception is thrown when CICS raises a particular condition.

Header file: ICCEXCEH

Samples: ICC$EXC1, ICC$EXC2, ICC$EXC3

IccException constructor

Constructor

IccException (Type *exceptionType*,
  IccBase::ClassType *classType*,
  const char* *className*,
  const char* *methodName*,
  IccMessage* *message*,
  IccBase* *object* **= 0,**
  unsigned short *exceptionNum* **= 0)**

exceptionType
: An enumeration, defined in this class, that indicates the type of the exception

classType
: An enumeration, defined in this class, that indicates from which type of class the exception was thrown

className
: The name of the class from which the exception was thrown

methodName
: The name of the method from which the exception was thrown

message
: A pointer to the **IccMessage** object that contains information about why the exception was created.

object
: A pointer to the object that threw the exception

exceptionNum
: The unique exception number.

Note: When the **IccException** object is created it takes ownership of the **IccMessage** given on the constructor. When the **IccException** is deleted, the **IccMessage** object is deleted automatically by the **IccException** destructor. Therefore, do not delete the **IccMessage** object before deleting the **IccException** object.

Public methods

className

const char* className() const

Returns the name of the class responsible for throwing this exception.

classType

IccBase::ClassType classType() const

Returns an enumeration, described under **ClassType** in **IccBase** class, that indicates the type of class which threw this exception.

message

IccMessage* message() const

Returns a pointer to an **IccMessage** object that contains information on any message associated with this exception.

methodName

const char* methodName() const

Returns the name of the method responsible for throwing this exception.

number

unsigned short number() const

Returns the unique exception number.

This is a useful diagnostic for IBM service. The number uniquely identifies from where in the source code the exception was thrown.

summary

const char* summary()

Returns a string containing a summary of the exception. This combines the **className**, **methodName**, **number**, **Type**, and **IccMessage::text** methods into the following form:

```
CICS exception summary: 094 IccTempStore::readNextItem type=CICSCondition
```

type

Type type() const

Returns an enumeration, defined in this class, that indicates the type of exception.

typeText

const char* typeText() const

Returns a string representation of the exception type, for example, objectCreationError, invalidArgument.

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

Type

objectCreationError
An attempt to create an object was invalid. This happens, for example, if an attempt is made to create a second instance of a singleton class, such as **IccTask**.

invalidArgument
A method was called with an invalid argument. This happens, for example, if an **IccBuf** object with too much data is passed to the **writeItem** method of the **IccTempStore** class by the application program. An attempt to create an **IccFileId** object with a 9-character filename also generates an exception of this type.

invalidMethodCall
A method call cannot proceed. A typical reason is that the object cannot honor the call in its current state. For example, a **readRecord** call on an **IccFile** object is only honored if an **IccRecordIndex** object, to specify ***which*** record is to be read, has already been associated with the file.

CICSCondition
A CICS condition, listed in the **IccCondition** structure, has occurred in the object and the object was configured to throw an exception.

platformError

An operation is invalid because of limitations of this particular platform. For example, an attempt to create an **IccJournal** object would fail under CICS for OS/2 because there are no CICS journal services on this server.

A platformError exception can occur at 3 levels:
1. An object is not supported on this platform.
2. An object is supported on this platform, but a particular method is not.
3. A method is supported on this platform, but a particular positional parameter is not.

See Platform differences˝ on page 56 for more details.

familyConformanceError

Family subset enforcement is on for this program and an operation that is not valid on all supported platforms has been attempted.

internalError

The CICS Foundation Classes have detected an internal error. Please call your support organization.

Chapter 26. IccFile class

IccBase
  IccResource
    IccFile

IccFile class enables the application program to access CICS files.

Header file: ICCFILEH

Sample: ICC$FIL

IccFile constructors

Constructor (1)

IccFile (const IccFileId& *id*,
  IccRecordIndex* *index* **= 0)**

id
: A reference to the **IccFileId** object that identifies which file is being operated on

index
: An optional pointer to the **IccRecordIndex** object that identifies which record in the file is being operated on.

Constructor (2)

IccFile (const char* *fileName*,
  IccRecordIndex* *index* **= 0)**

fileName
: The 8-character name of the file

index
: An optional pointer to the **IccRecordIndex** object that identifies which record in the file is being operated on.

To access files using an **IccFile** object, it must have an **IccRecordIndex** object associated with it. If this association is not made when the object is created, use the **registerRecordIndex** method.

Public methods

The opt parameter
Many methods have the same parameter, *opt*, which is described under the **abendCode** method inabendCode” on page 71.

access

unsigned long access(Icc::GetOpt *opt* **=Icc::object)**

opt
An enumeration, defined in **Icc** structure, that indicates whether you can use a value previously retrieved from CICS (object), or whether the object should retrieve a fresh value from CICS.

Returns a composite number indicating the access properties of the file. See also **isReadable**, **isBrowsable**, **isAddable**, **isDeletable**, and **isUpdatable** methods.

accessMethod

IccValue::CVDA accessMethod(Icc::GetOpt *opt* **=** Icc::object**)**

opt
See **access** method.

Returns an enumeration, defined in **IccValue**, that represents the access method for this file. Possible values are:
VSAM
BDAM
SFS

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

beginInsert(VSAM only)

void beginInsert()

Signals the start of a mass insertion of data into the file.

deleteLockedRecord

void deleteLockedRecord(unsigned long *updateToken* **= 0)**

updateToken
A token that indicates which previously read record is to be deleted. This is the token that is returned from **readRecord** method when in update mode.

Deletes a record that has been previously locked by **readRecord** method in update mode. (See also **readRecord** method.)

Conditions

DISABLED, DUPKEY, FILENOTFOUND, ILLOGIC, INVREQ, IOERR, ISCINVREQ, NOTAUTH, NOTFIND, NOTOPEN, SYSIDERR, LOADING

deleteRecord

unsigned short deleteRecord()

Deletes one or more records, as specified by the associated **IccRecordIndex** object, and returns the number of deleted records.

Conditions

DISABLED, DUPKEY, FILENOTFOUND, ILLOGIC, INVREQ, IOERR, ISCINVREQ, NOTAUTH, NOTFIND, NOTOPEN, SYSIDERR, LOADING

enableStatus

IccValue::CVDA enableStatus(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Returns an enumeration, defined in **IccValue**, that indicates whether the file is enabled to be used by programs. Possible values are:
- DISABLED
- DISABLING
- ENABLED
- UNENABLED

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

endInsert(VSAM only)

void endInsert()

Marks the end of a mass insertion operation. See **beginInsert**.

isAddable

Icc::Bool isAddable(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Indicates whether more records can be added to the file.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

isBrowsable

Icc::Bool isBrowsable(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Indicates whether the file can be browsed.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

isDeletable

Icc::Bool isDeletable(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Indicates whether the records in the file can be deleted.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

isEmptyOnOpen

Icc::Bool isEmptyOnOpen(Icc::GetOpt *opt* **=** Icc::object**)**

opt
See **access** method.

Returns a Boolean that indicates whether the EMPTYREQ option is specified. EMPTYREQ causes the object associated with this file to be set to empty when opened, if it is a VSAM data set defined as reusable.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

isReadable

Icc::Bool isReadable(Icc::GetOpt *opt* **=** Icc::object**)**

opt
See **access** method.

Indicates whether the file records can be read.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

isRecoverable

Icc::Bool isRecoverable(Icc::GetOpt *opt* **=** Icc::object**)**

opt
See **access** method.

Conditions: END, FILENOTFOUND, ILLOGIC, NOTAUTH

isUpdatable

Icc::Bool isUpdatable(Icc::GetOpt *opt* **=** Icc::object**)**

opt
See **access** method.

Indicates whether the file can be updated.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

keyLength

unsigned long keyLength(Icc::GetOpt *opt* **=** Icc::object**)**

opt
See **access** method.

Returns the length of the search key.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

keyPosition

long keyPosition(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Returns the position of the key field in each record relative to the beginning of the record. If there is no key, zero is returned.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

openStatus

IccValue::CVDA openStatus(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Returns a CVDA that indicates the open status of the file. Possible values are:

CLOSED
: The file is closed.

CLOSING
: The file is in the process of being closed. Closing a file may require dynamic deallocation of data sets and deletion of shared resources, so the process may last a significant length of time.

CLOSEREQUEST
: The file is open and one or more application tasks are using it. A request has been received to close it.

OPEN
: The file is open.

OPENING
: The file is in the process of being opened.

Conditions: END, FILENOTFOUND, ILLOGIC, NOTAUTH

readRecord

const IccBuf& readRecord (ReadMode *mode* **=** normal**,**
unsigned long* *updateToken* **= 0)**

mode
: An enumeration, defined in this class, that indicates in which mode the record is to be read.

updateToken
: A pointer to an **unsigned long** token that will be updated by the method when *mode* is update and you wish to make multiple read updates. The token uniquely identifies the update request and is passed to the **deleteLockedRecord**, **rewriteRecord**, or **unlockRecord** methods

Reads a record and returns a reference to an **IccBuf** object that contains the data from the record.

Conditions

DISABLED, DUPKEY, FILENOTFOUND, ILLOGIC, INVREQ, IOERR, ISCINVREQ, LENGERR, NOTAUTH, NOTFND, NOTOPEN, SYSIDERR, LOADING

recordFormat

IccValue::CVDA recordFormat(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Returns a CVDA that indicates the format of the data. Possible values are:

FIXED
: The records are of fixed length.

UNDEFINED (BDAM data sets only)
: The format of records on the file is undefined.

VARIABLE
: The records are of variable length. If the file is associated with a data table, the record format is always variable length, even if the source data set contains fixed-length records.

Conditions: END, FILENOTFOUND, ILLOGIC, NOTAUTH

recordIndex

IccRecordIndex* recordIndex() const

Returns a pointer to an **IccRecordIndex** object that indicates which records are to be accessed when using methods such as **readRecord**, **writeRecord**, and **deleteRecord**.

recordLength

unsigned long recordLength(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Returns the length of the current record.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

registerRecordIndex

void registerRecordIndex(IccRecordIndex* *index***)**

index
: A pointer to an **IccKey**, **IccRBA**, or **IccRRN** object that will be used by methods such as **readRecord**, **writeRecord**, etc..

rewriteRecord

void rewriteRecord (const IccBuf& *buffer***,**
unsigned long *updateToken* **= 0)**

buffer
A reference to the **IccBuf** object that holds the new record data to be written to the file.

updateToken
The token that identifies which previously read record is to be rewritten. See **readRecord**.

Updates a record with the contents of *buffer*.

Conditions

DISABLED, FILENOTFOUND, ILLOGIC, INVREQ, IOERR, ISCINVREQ, NOTAUTH, NOTFND, NOTOPEN, SYSIDERR, LOADING

setAccess

void setAccess(unsigned long *access***)**

access
A positive integer value created by ORing (or adding) one or more of the values of the Access enumeration, defined in this class.

Sets the permitted access to the file. For example:

```
file.setAccess(IccFile::readable + IccFile::notUpdatable);
```

Conditions

FILENOTFOUND, INVREQ, IOERR, NOTAUTH

setEmptyOnOpen

void setEmptyOnOpen(Icc::Bool *trueFalse***)**

Specifies whether or not to make the file empty when it is next opened.

Conditions

FILENOTFOUND, INVREQ, IOERR, NOTAUTH

setStatus

void setStatus(Status *status***)**

status
An enumeration, defined in this class, that indicates the required status of the file after this method is called.

Sets the status of the file.

Conditions

FILENOTFOUND, INVREQ, IOERR, NOTAUTH

type

IccValue::CVDA type(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: See **access** method.

Returns a CVDA that identifies the type of data set that corresponds to this file. Possible values are:

ESDS	The data set is an entry-sequenced data set.
KEYED	The data set is addressed by physical keys.
KSDS	The data set is a key-sequenced data-set.
NOTKEYED	The data set is not addressed by physical keys.
RRDS	The data set is a relative record data set.
VRRDS	The data set is a variable relative record data set.

Conditions: END, FILENOTFOUND, ILLOGIC, NOTAUTH

unlockRecord

void unlockRecord(unsigned long *updateToken* **= 0)**

updateToken
: A token that indicates which previous **readRecord** update request is to be unlocked.

Unlock a record, previously locked by reading it in update mode. See **readRecord**.

Conditions

DISABLED, FILENOTFOUND, ILLOGIC, IOERR, ISCINVREQ, NOTAUTH, NOTOPEN, SYSIDERR, INVREQ

writeRecord

void writeRecord(const IccBuf& *buffer***)**

buffer
: A reference to the **IccBuf** object that holds the data that is to be written into the record.

Write either a single record or a sequence of records, if used with the **beginInsert** and **endInsert** methods.

Conditions

DISABLED, DUPREC, FILENOTFOUND, ILLOGIC, INVREEQ, IOERR, ISCINVREQ, LENGERR, NOSPACE, NOTAUTH, NOTOPEN, SYSIDERR, LOADING, SUPPRESSED

Inherited public methods

Method	**Class**
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
className	IccBase
classType	IccBase

Method	Class
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
isRouteOptionOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
routeOption	IccResource
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource
setRouteOption	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

Access

readable
File records can be read by CICS tasks.

notReadable
File records cannot be read by CICS tasks.

browsable
File records can be browsed by CICS tasks.

notBrowsable
File records cannot be browsed by CICS tasks.

addable
Records can be added to the file by CICS tasks.

notAddable
Records cannot be added to the file by CICS tasks.

updatable
Records in the file can be updated by CICS tasks.

notUpdatable
Records in the file cannot be updated by CICS tasks.

deletable
Records in the file can be deleted by CICS tasks.

notDeletable
Records in the file cannot be deleted by CICS tasks.

fullAccess
Equivalent to readable AND browsable AND addable AND updatable AND deletable.

noAccess
Equivalent to notReadable AND notBrowsable AND notAddable AND notUpdatable AND notDeletable.

ReadMode

The mode in which a file is read.

normal
No update is to be performed (that is, read-only mode)

update
The record is to be updated. The record is locked by CICS until:

- it is rewritten using the **rewriteRecord** method *or*

- it is deleted using the **deleteLockedRecord** method *or*
- it is unlocked using the **unlockRecord** method *or*
- the task commits or rolls back its resource updates *or*
- the task is abended.

SearchCriterion

equalToKey
The search only finds an exact match.

gteqToKey
The search finds either an exact match or the next record in search order.

Status

open
File is open, ready for read/write requests by CICS tasks.

closed
File is closed, and is therefore not currently being used by CICS tasks.

enabled
File is enabled for access by CICS tasks.

disabled
File is disabled from access by CICS tasks.

Chapter 27. IccFileId class

IccBase
　　IccResourceId
　　　　IccFileId

IccFileId is used to identify a file name in the CICS system. On MVS/ESA this is an entry in the FCT (file control table).

Header file: ICCRIDEH

IccFileId constructors

Constructor (1)

IccFileId(const char* *fileName***)**

fileName
　The name of the file.

Constructor (2)

IccFileId(const IccFileId& *id***)**

id
　A reference to an **IccFileId** object.

Public methods

operator= (1)

IccFileId& operator=(const char* *fileName***)**

fileName
　The 8-byte name of the file.

operator= (2)

IccFileId& operator=(const IccFileId& *id***)**

id
　A reference to an **IccFileId** object.

Assigns new value.

Inherited public methods

Method	**Class**
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId

Method	Class
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 28. IccFileIterator class

IccBase
  IccResource
    IccFileIterator

This class is used to create **IccFileIterator** objects that can be used to browse through the records of a CICS file, represented by an **IccFile** object.

Header file: ICCFLIEH

Sample: ICC$FIL

IccFileIterator constructor

Constructor

IccFileIterator (IccFile* *file***,**
IccRecordIndex* *index***,**
IccFile::SearchCriterion *search* **=** IccFile::gteqToKey**)**

file
A pointer to the **IccFile** object that is to be browsed

index
A pointer to the **IccRecordIndex** object that is being used to select a record in the file

search
An enumeration, defined in **IccFile**, that indicates the criterion being used to find a search match. The default is gteqToKey.

The **IccFile** and **IccRecordIndex** object must exist before the **IccFileIterator** is created.

Conditions

DISABLED, FILENOTFOUND, ILLOGIC, INVREQ, IOERR, ISCINVREQ, NOTAUTH, NOTFND, NOTOPEN, SYSIDERR, LOADING

Public methods

readNextRecord

const IccBuf& readNextRecord (IccFile::ReadMode *mode* **=** IccFile::normal**,**
unsigned long* *updateToken* **=** 0**)**

mode
An enumeration, defined in **IccFile** class, that indicates the type of read request

updateToken
A returned token that is used to identify this unique update request on a subsequent **rewriteRecord**, **deleteLockedRecord**, or **unlockRecord** method on the file object.

Read the record that follows the current record.

Conditions

DUPKEY, ENDFILE, FILENOTFOUND, ILLOGIC, INVREQ, IOERR, ISCINVREQ, LENGERR, NOTAUTH, NOTFIND, SYSIDERR

readPreviousRecord

const IccBuf& readPreviousRecord (IccFile::ReadMode *mode* **=** IccFile::normal**,**
unsigned long* *updateToken* **=** 0**)**

mode
An enumeration, defined in **IccFile** class, that indicates the type of read request.

updateToken
See **readNextRecord**.

Read the record that precedes the current record.

Conditions

DUPKEY, ENDFILE, FILENOTFOUND, ILLOGIC, INVREQ, IOERR, ISCINVREQ, LENGERR, NOTAUTH, NOTFIND, SYSIDERR

reset

void reset (IccRecordIndex* *index***,**
IccFile::SearchCriterion *search* **=** IccFile::gteqToKey**)**

index
A pointer to the **IccRecordIndex** object that is being used to select a record in the file.

search
An enumeration, defined in **IccFile**, that indicates the criterion being used to find a search match. The default is gteqToKey.

Resets the **IccFileIterator** object to point to the record identified by the **IccRecordIndex** object and the specified search criterion.

Conditions

FILENOTFOUND, ILLOGIC, INVREQ, IOERR, ISCINVREQ, NOTAUTH, NOTFND, SYSIDERR

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
className	IccBase
classType	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
isRouteOptionOn	IccResource
name	IccResource

Method	Class
operator delete	IccBase
operator new	IccBase
routeOption	IccResource
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource
setRouteOption	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 29. IccGroupId class

IccBase
    IccResourceId
        IccGroupId

IccGroupId class is used to identify a CICS group.

Header file: ICCRIDEH

IccGroupId constructors

Constructor (1)

IccGroupId(const char* *groupName***)**

groupName
    The 8-character name of the group.

Constructor (2)

IccGroupId(const IccGroupId& *id***)**

id A reference to an **IccGroupId** object.
The copy constructor.

Public methods

operator= (1)

IccGroupId& operator=(const char* *groupName***)**

groupName
    The 8-character name of the group.

operator= (2)

IccGroupId& operator=(const IccGroupId& *id***)**

id A reference to an **IccGroupId** object.
Assigns new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase

Method	Class
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 30. IccJournal class

IccBase
  IccResource
    IccJournal

IccJournal class represents a user or system CICS journal.

Header file: ICCJRNEH

Sample: ICC$JRN

IccJournal constructors

Constructor (1)

IccJournal (const IccJournalId& *id***,**
    unsigned long *options* **= 0)**

id
: A reference to an **IccJournalId** object that identifies which journal is being used.

options
: An integer, constructed from the **Options** enumeration defined in this class, that affects the behavior of **writeRecord** calls on the **IccJournal** object. The values may be combined by addition or bitwise ORing, for example:

```
IccJournal::startIO | IccJournal::synchronous
```

The default is to use the system default.

Constructor (2)

IccJournal (unsigned short *journalNum***,**
    unsigned long *options* **= 0)**

journalNum
: The journal number (in the range 1-99)

options
: See above.

Public methods

clearPrefix

void clearPrefix()

Clears the current prefix as set by **registerPrefix** or **setPrefix**.

If the current prefix was set using **registerPrefix**, then the **IccJournal** class only removes its own reference to the prefix. The buffer itself is left unchanged.

If the current prefix was set by **setPrefix**, then the **IccJournal**'s copy of the buffer is deleted.

journalTypeId

const IccJournalTypeId& journalTypeId() const

Returns a reference to an **IccJournalTypeId** object that contains a 2-byte field used to identify the origin of journal records.

put

virtual void put(const IccBuf& *buffer***)**

buffer
A reference to an **IccBuf** object that holds data to be put into the journal.

A synonym for **writeRecord**—puts data into the journal. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

registerPrefix

void registerPrefix(const IccBuf* *prefix***)**

Stores pointer to prefix object for use when the **writeRecord** method is called on this **IccJournal** object.

setJournalTypeId (1)

void setJournalTypeId(const IccJournalTypeId& *id***)**

setJournalTypeId (2)

void setJournalTypeId(const char* *jtypeid***)**

Sets the journal type—a 2 byte identifier—included in the journal record created when using the **writeRecord** method.

setPrefix (1)

void setPrefix(const IccBuf& *prefix***)**

setPrefix (2)

void setPrefix(const char* *prefix***)**

Stores the *current* contents of *prefix* for inclusion in the journal record created when the **writeRecord** method is called.

wait

void wait (unsigned long *requestNum***=0,**
unsigned long *option* **= 0)**

requestNum
The write request. Zero indicates the last write on this journal.

option
An integer that affects the behaviour of **writeRecord** calls on the **IccJournal** object. Values other than 0 should be made from the **Options** enumeration, defined in this class. The values may be combined by addition or bitwise ORing, for example `IccJournal::startIO + IccJournal::synchronous`. The default is to use the system default.

Waits until a previous journal write has completed.

Condition: IOERR, JIDERR, NOTOPEN

writeRecord (1)

unsigned long writeRecord (const IccBuf& *record***,**
unsigned long *option* **= 0)**

record
A reference to an **IccBuf** object that holds the record

option
See above.

writeRecord (2)

unsigned long writeRecord (const char* *record***,**
unsigned long *option* **= 0)**

record
The name of the record

option
See above.

Writes the data in the record to the journal.

The returned number represents the particular write request and can be passed to the **wait** method in this class.

Conditions

IOERR, JIDERR, LENGERR, NOJBUFSP, NOTAUTH, NOTOPEN

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource

Method	Class
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

Options

The behaviour of **writeRecord** calls on the **IccJournal** object. The values can be combined in an integer by addition or bitwise ORing.

startIO
: Specifies that the output of the journal record is to be initiated immediately. If synchronous is specified for a journal that is not frequently used, you should also specify startIO to prevent the requesting task waiting for the journal buffer to be filled. If the journal is used frequently, startIO is unnecessary.

noSuspend
: Specifies that the NOJBUFSP condition does not suspend an application program.

synchronous
: Specifies that synchronous journal output is required. The requesting task waits until the record has been written.

Chapter 31. IccJournalId class

IccBase
  IccResourceId
    IccJournalId

IccJournalId is used to identify a journal number in the CICS sytem.

Header file: ICCRIDEH

IccJournalId constructors

Constructor (1)

IccJournalId(unsigned short *journalNum***)**

journalNum
  The number of the journal, in the range 1 to 99

Constructor (2)

IccJournalId(const IccJournalId& *id***)**

id
  A reference to an **IccJournalId** object.

The copy constructor.

Public methods

number

unsigned short number() const

Returns the journal number, in the range 1 to 99.

operator= (1)

IccJournalId& operator=(unsigned short *journalNum***)**

journalNum
  The number of the journal, in the range 1 to 99

operator= (2)

IccJournalId& operator=(const IccJournalId& *id***)**

id
  A reference to an **IccJournalId** object.

Assigns new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 32. IccJournalTypeId class

IccBase
  IccResourceId
    IccJournalTypeId

An **IccJournalTypeId** class object is used to help identify the origin of a journal record—it contains a 2-byte field that is included in the journal record.

Header file: ICCRIDEH

IccJournalTypeId constructors

Constructor (1)

IccJournalTypeId(const char* *journalTypeName***)**

journalTypeName
A 2-byte identifier used in journal records.

Constructor (2)

IccJournalTypeId(const IccJournalId& *id***)**

id A reference to an **IccJournalTypeId** object.

Public methods

operator= (1)

void operator=(const IccJournalTypeId& *id***)**

id A reference to an **IccJournalTypeId** object.

operator= (2)

void operator=(const char* *journalTypeName***)**

journalTypeName
A 2-byte identifier used in journal records.

Sets the 2-byte field that is included in the journal record.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase

Method	Class
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 33. IccKey class

IccBase
  IccRecordIndex
    IccKey

IccKey class is used to hold a search key for an indexed (KSDS) file.

Header file: ICCRECEH

Sample: ICC$FIL

IccKey constructors

Constructor (1)

IccKey (const char* *initValue***,**
Kind *kind* **=** complete**)**

Constructor (2)

IccKey (unsigned short *completeLength***,**
Kind *kind***=** complete**)**

Constructor (3)

IccKey(const IccKey& *key***)**

Public methods

assign

void assign (unsigned short *length***,**
const void* *dataArea***)**

length
: The length of the data area

dataArea
: A pointer to the start of the data area that holds the search key.

Copies the search key into the **IccKey** object.

completeLength

unsigned short completeLength() const

Returns the length of the key when it is complete.

kind

Kind kind() const

Returns an enumeration, defined in this class, that indicates whether the key is generic or complete.

operator= (1)

IccKey& operator=(const IccKey& *key***)**

operator= (2)

IccKey& operator=(const IccBuf& buffer)

operator= (3)

IccKey& operator=(const char* *value***)**

Assigns new value to key.

operator== (1)

Icc::Bool operator==(const IccKey& *key***) const**

operator== (2)

Icc::Bool operator==(const IccBuf& *text***) const**

operator== (3)

Icc::Bool operator==(const char* *text***) const**

Tests equality.

operator!= (1)

Icc::Bool operator !=(const IccKey& *key***) const**

operator!= (2)

Icc::Bool operator!=(const IccBuf& *text***) const**

operator!= (3)

Icc::Bool operator!=(const char* *text***) const**

Tests inequality.

setKind

void setKind(Kind *kind***)**

kind
An enumeration, defined in this class, that indicates whether the key is generic or complete.

Changes the type of key from generic to complete or vice versa.

value

const char* value()

Returns the start of the data area containing the search key.

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
length	IccRecordIndex
operator delete	IccBase
operator new	IccBase
type	IccRecordIndex
value	IccRecordIndex

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

Kind

complete
Specifies that the supplied key is not generic.

generic
Specifies that the search key is generic. A search is satisfied when a record is found with a key whose prefix matches the supplied key.

Chapter 34. IccLockId class

IccBase
  IccResourceId
    IccLockId

IccLockId class is used to identify a lock request.

Header file: ICCRIDEH

IccLockId constructors

Constructor (1)

IccLockId(const char* *name***)**

name
: The 8-character name of the lock request.

Constructor (2)

IccLockId(const IccLockId& *id***)**

id A reference to an **IccLockId** object.

The copy constructor.

Public methods

operator= (1)

IccLockId& operator=(const char* *name***)**

name
: The 8-character name of the lock request.

operator= (2)

IccLockId& operator=(const IccLockId& *id***)**

id A reference to an **IccLockId** object.

Assigns new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase

Method	Class
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 35. IccMessage class

IccBase
    IccMessage

IccMessage can be used to hold a message description. It is used primarily by the **IccException** class to describe why the **IccException** object was created.

Header file: ICCMSGEH

IccMessage constructor

Constructor

IccMessage (unsigned short *number*,
    const char* *text*,
    const char* *className* **= 0,**
    const char* *methodName* **= 0)**

number
: The number associated with the message

text
: The text associated with the message

className
: The optional name of the class associated with the message

methodName
: The optional name of the method associated with the message.

Public methods

className

const char* className() const

Returns the name of the class with which the message is associated, if any. If there is no name to return, a null pointer is returned.

methodName

const char* methodName() const

Returns the name of the method with which the message is associated, if any. If there is no name to return, a null pointer is returned.

number

unsigned short number() const

Returns the number of the message.

summary

const char* summary()

Returns the text of the message.

text

const char* text() const

Returns the text of the message in the same way as summary.

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 36. IccPartnerId class

IccBase
  IccResourceId
    IccPartnerId

IccPartnerId class represents CICS remote (APPC) partner transaction definitions.

Header file: ICCRIDEH

IccPartnerId constructors

Constructor (1)

IccPartnerId(const char* *partnerName***)**

partnerName
The 8-character name of an APPC partner.

Constructor (2)

IccPartnerId(const IccPartnerId& *id***)**

id A reference to an **IccPartnerId** object.

The copy constructor.

Public methods

operator= (1)

IccPartnerId& operator=(const char* *partnerName***)**

partnerName
The 8-character name of an APPC partner.

operator= (2)

IccPartnerId& operator=(const IccPartnerId& *id***)**

id A reference to an **IccPartnerId** object.

Assigns new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase

Method	Class
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 37. IccProgram class

IccBase
  IccResource
    IccProgram

The **IccProgram** class represents any CICS program outside of your currently executing one, which the **IccControl** object represents.

Header file: ICCPRGEH

Sample: ICC$PRG1, ICC$PRG2, ICC$PRG3

IccProgram constructors

Constructor (1)

IccProgram(const IccProgramId& *id***)**

id
A reference to an **IccProgramId** object.

Constructor (2)

IccProgram(const char* *progName***)**

progName
The 8-character name of the program.

Public methods

The opt parameter
Many methods have the same parameter, *opt*, which is described under the **abendCode** method inabendCode" on page 71.

address

const void* address() const

Returns the address of a program module in memory. This is only valid after a successful **load** call.

clearInputMessage

void clearInputMessage()

Clears the current input message which was set by **setInputMessage** or **registerInputMessage**.

If the current input message was set using **registerInputMessage** then only the pointer is deleted: the buffer is left unchanged.

If the current input message was set using **setInputMessage** then **clearInputMessage** releases the memory used by that buffer.

entryPoint

const void* entryPoint() const

Returns a pointer to the entry point of a loaded program module. This is only valid after a successful **load** call.

length

unsigned long length() const

Returns the length of a program module. This is only valid after a successful **load** call.

link

void link (const IccBuf* *commArea* **= 0,**
const IccTransId* *transId* **= 0,**
CommitOpt *opt* **=** noCommitOnReturn**)**

commArea
: An optional pointer to the **IccBuf** object that contains the COMMAREA—the buffer used to pass information between the calling program and the program that is being called

transId
: An optional pointer to the **IccTransId** object that indicates the name of the mirror transaction under which the program is to run if it is a remote (DPL) program link

opt
: An enumeration, defined in this class, that affects the behavior of the link when the program is remote (DPL). The default (noCommitOnReturn) is not to commit resource changes on the remote CICS region until the current task commits its resources. The alternative (commitOnReturn) means that the resources of the remote program are committed whether or not this task subsequently abends or encounters a problem.

Conditions: INVREQ, NOTAUTH, PGMIDERR, SYSIDERR, LENGERR, ROLLEDBACK, TERMERR

Restrictions

Links may be nested, that is, a linked program may **link** to another program. However, due to implementation restrictions, you may only nest such programs 15 times. If this is exceeded, an exception is thrown.

load

void load(LoadOpt *opt* **=** releaseAtTaskEnd**)**

opt
An enumeration, defined in this class, that indicates whether CICS should automatically allow the program to be unloaded at task termination (releaseAtTaskEnd), or not (hold).

Conditions: NOTAUTH, PGMIDERR, INVREQ, LENGERR

registerInputMessage

void registerInputMessage(const IccBuf& *msg***)**

Store pointer to InputMessage for when the **link** method is called.

setInputMessage

void setInputMessage(const IccBuf& *msg***)**

Specifies data to be made available, by the **IccSession::receive()** method, to the called program, when using the **link** method in this class.

unload

void unload()

Allow a program to be unloaded. It can be reloaded by a call to **load**.

Conditions

NOTAUTH, PGMIDERR, INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
className	IccBase
classType	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
isRouteOptionOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
routeOption	IccResource
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource
setRouteOption	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

CommitOpt

noCommitOnReturn
Changes to resources on the remote CICS region are not committed until the current task commits its resources. This is the default setting.

commitOnReturn
Changes to resources on the remote CICS region are committed whether or not the current task subsequently abends or encounters a problem.

LoadOpt

releaseAtTaskEnd
Indicates that CICS should automatically allow the program to be unloaded at task termination.

hold
Indicates that CICS should not automatically allow the program to be unloaded at task termination. (In this case, this or another task must explicitly use the **unload** method).

Chapter 38. IccProgramId class

IccBase
  IccResourceId
    IccProgramId

IccProgramId objects represent program names in the CICS system. On MVS/ESA this is an entry in the PPT (program processing table).

Header file: ICCRIDEH

IccProgramId constructors

Constructor (1)

IccProgramId(const char* *progName***)**

progName
  The 8-character name of the program.

Constructor (2)

The copy constructor.

IccProgramId(const IccProgramId& *id***)**

id
  A reference to an **IccProgramId** object.

Public methods

operator= (1)

IccProgramId& operator=(const char* *progName***)**

progName
  The 8-character name of the program.

operator= (2)

IccProgramId& operator=(const IccProgramId& *id***)**

id
  A reference to an **IccProgramId** object.

Assigns new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase

Method	Class
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 39. IccRBA class

IccBase
    IccRecordIndex
        IccRBA

An **IccRBA** object holds a relative byte address which is used for accessing VSAM ESDS files.

Header file: ICCRECEH

IccRBA constructor

Constructor

IccRBA(unsigned long *initRBA* **= 0)**

initRBA
    An initial value for the relative byte address.

Public methods

operator= (1)

IccRBA& operator=(const IccRBA& *rba***)**

operator= (2)

IccRBA& operator=(unsigned long *num***)**

num
    A valid relative byte address.

Assigns a new value for the relative byte address.

operator== (1)

Icc::Bool operator== (const IccRBA& *rba***) const**

operator== (2)

Icc::Bool operator== (unsigned long *num***) const**

Tests equality

operator!= (1)

Icc!:Bool operator== (const IccRBA& *rba***) const**

operator!= (2)

Icc::Bool operator!=(unsigned long *num*) const

Tests inequality

number

unsigned long number() const

Returns the relative byte address.

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
length	IccRecordIndex
operator delete	IccBase
operator new	IccBase
type	IccRecordIndex
value	IccRecordIndex

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 40. IccRecordIndex class

IccBase
  IccRecordIndex
    IccKey
    IccRBA
    IccRRN

CICS File Control Record Identifier. Used to tell CICS which particular record the program wants to retrieve, delete, or update. **IccRecordIndex** is a base class from which **IccKey**, **IccRBA**, and **IccRRN** are derived.

Header file: ICCRECEH

IccRecordIndex constructor (protected)

Constructor

IccRecordIndex(Type *type***)**

type
An enumeration, defined in this class, that indicates whether the index type is key, RBA, or RRN.

Note: This is protected because you should not create **IccRecordIndex** objects; see subclasses **IccKey**, **IccRBA**, and **IccRRN**.

Public methods

length

unsigned short length() const

Returns the length of the record identifier.

type

Type type() const

Returns an enumeration, defined in this class, that indicates whether the index type is key, RBA, or RRN.

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

Type

Indicates the access method. Possible values are:
- key
- RBA
- RRN

Chapter 41. IccRequestId class

IccBase
  IccResourceId
    IccRequestId

An **IccRequestId** is used to hold the name of a request. This request identifier can subsequently be used to cancel a request—see, for example, **start** and **cancel** methods in **IccStartRequestQ** class.

Header file: ICCRIDEH

IccRequestId constructors

Constructor (1)

IccRequestId()

An empty **IccRequestId** object.

Constructor (2)

IccRequestId(const char* *requestName***)**

requestName
  The 8-character name of the request.

Constructor (3)

The copy constructor.

IccRequestId(const IccRequestId& *id***)**

id A reference to an **IccRequestId**.

Public methods

operator= (1)

IccRequestId& operator=(const IccRequestId& *id***)**

id A reference to an **IccRequestId** object whose properties are copied into this object.

operator= (2)

IccRequestId& operator=(const char* *reqestName***)**

requestName
  An 8-character string which is copied into this object.

Assigns new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 42. IccResource class

IccBase
  IccResource

IccResource class is a base class that is used to derive other classes. The methods associated with **IccResource** are described here although, in practise, they are only called on objects of derived classes.

IccResource is the parent class for all CICS resources—tasks, files, programs, etc. Every class inherits from **IccBase**, but only those that use CICS services inherit from **IccResource**.

Header file: ICCRESEH

Sample: ICC$RES1, ICC$RES2

IccResource constructor (protected)

Constructor

IccResource(IccBase::ClassType *classType***)**

classType
: An enumeration that indicates what the subclass type is. For example, for an **IccTempStore** object, the class type is cTempStore. The possible values are listed under **ClassType** in the description of the **IccBase** class.

Public methods

actionOnCondition

ActionOnCondition actionOnCondition(IccCondition::Codes *condition***)**

condition
: The name of the condition as an enumeration. See **IccCondition** structure for a list of the possible values.

Returns an enumeration that indicates what action the class will take in response to the specified condition being raised by CICS. The possible values are described in this class.

actionOnConditionAsChar

char actionOnConditionAsChar(IccCondition::Codes *condition***)**

This method is the same as **actionOnCondition** but returns a character, rather than an enumeration, as follows:

0 (zero)
: No action is taken for this CICS condition.

H
: The virtual method **handleEvent** is called for this CICS condition.

X
: An exception is generated for this CICS condition.

A
: This program is abended for this CICS condition.

actionsOnConditionsText

const char* actionsOnConditionsText()

Returns a string of characters, one character for each possible condition. Each character indicates the actions to be performed for that corresponding condition. The characters used in the string are described above in actionOnConditionAsChar” on page 177. For example, the string: `0X00H0A ...` shows the actions for the first seven conditions are as follows:

condition 0 (NORMAL)
action=0 (noAction)

condition 1 (ERROR)
action=X (throwException)

condition 2 (RDATT)
action=0 (noAction)

condition 3 (WRBRK)
action=0 (noAction)

condition 4 (ICCEOF)
action=H (callHandleEvent)

condition 5 (EODS)
action=0 (noAction)

condition 6 (EOC)
action=A (abendTask)

clear

virtual void clear()

Clears the contents of the object. This method is virtual and is implemented, wherever appropriate, in the derived classes. See Polymorphic Behavior˝ on page 59 for a description of polymorphism. The default implementation in this class throws an exception to indicate that it has not been overridden in a subclass.

condition

unsigned long condition(ConditionType *type* **=** majorCode**) const**

type
An enumeration, defined in this class, that indicates the type of condition requested. Possible values are majorCode (the default) and minorCode.

Returns a number that indicates the condition code for the most recent CICS call made by this object.

conditionText

const char* conditionText() const

Returns the symbolic name of the last CICS condition for this object.

get

virtual const IccBuf& get()

Gets data from the **IccResource** object and returns it as an **IccBuf** reference. This method is virtual and is implemented, wherever appropriate, in the derived classes. See Polymorphic Behavior˝ on page 59 for a description of polymorphism. The default implementation in this class throws an exception to indicate that it has not been overridden in a subclass.

handleEvent

virtual HandleEventReturnOpt handleEvent(IccEvent& *event***)**

event
: A reference to an **IccEvent** object that describes the reason why this method is being called.

This virtual function may be re-implemented in a subclass (by the application programmer) to handle CICS events (see **IccEvent** class on page 123).

id

const IccResourceId* id() const

Returns a pointer to the **IccResourceId** object associated with this **IccResource** object.

isEDFOn

Icc::Bool isEDFOn() const

Returns a boolean value that indicates whether EDF trace is active. Possible values are yes or no.

isRouteOptionOn

Icc::Bool isRouteOptionOn() const

Returns a boolean value that indicates whether the route option is active. Possible values are yes or no.

name

const char* name() const

Returns a character string that gives the name of the resource that is being used. For an **IccTempStore** object, the 8-character name of the temporary storage queue is returned. For an **IccTerminal** object, the 4-character terminal name is returned. This is equivalent to calling **id()name**.

put

virtual void put(const IccBuf& *buffer***)**

buffer
: A reference to an **IccBuf** object that contains data that is to be put into the object.

Puts information from the buffer into the **IccResource** object. This method is virtual and is implemented, wherever appropriate, in the derived classes. See Polymorphic Behavior˝ on page 59 for more information on polymorphism. The default implementation in this class throws an exception to indicate that it has not been overridden in a subclass.

routeOption

const IccSysId& routeOption() const

Returns a reference to an **IccSysId** object that represents the system to which all CICS requests are routed—explicit function shipping.

setActionOnAnyCondition

void setActionOnAnyCondition(ActionOnCondition *action***)**

action
: The name of the action as an enumeration. The possible values are listed under the description of this class.

Specifies the default action to be taken by the CICS foundation classes when a CICS condition occurs.

setActionOnCondition

void setActionOnCondition (ActionOnCondition *action***,**
IccCondition::Codes *condition***)**

action
: The name of the action as an enumeration. The possible values are listed under the description of this class.

condition
: See **IccCondition** structure.

Specifies what action is automatically taken by the CICS foundation classes when a given CICS condition occurs.

setActionsOnConditions

void setActionsOnConditions(const char* *actions* **= 0)**

actions
: A string that indicates what action is to be taken for each condition. The default is not to indicate any actions, in which case each condition is given a default **ActionOnCondition** of noAction. The string should have the same format as the one returned by the **actionsOnConditionsText** method.

setEDF

void setEDF(Icc::Bool *onOff***)**

onOff
: A boolean value that selects whether EDF trace is switched on or off.

Switches EDF on or off for this resource object. See Execution Diagnostic Facility” on page 50.

These methods force the object to route CICS requests to the named remote system. This is called explicit function shipping.

setRouteOption (1)

void setRouteOption(const IccSysId& *sysId***)**

The parameters are:

sysId
The **IccSysId** object that represents the remote system to which commands are routed.

setRouteOption (2)

void setRouteOption(const char* *sysName* **= 0)**

sysName
The 4-character name of the system to which commands are routed.

This option is only valid for the following classes:
- **IccDataQueue**
- **IccFile**
- **IccFileIterator**
- **IccProgram**
- **IccStartRequestQ**
- **IccTempStore**

Attempting to use this method on other subclasses of **IccResource** causes an exception to be thrown.

To turn off the route option specify no parameter, for example:

```
obj.setRouteOption()
```

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

ActionOnCondition

Possible values are:

noAction
Carry on as normal; it is the application programs responsibility to test CICS conditions using the **condition** method, after executing a method that calls CICS services.

callHandleEvent
Call the virtual **handleEvent** method.

throwException
An **IccException** object is created and thrown. This is typically used for more serious conditions or errors.

abendTask
Abend the CICS task.

HandleEventReturnOpt

Possible values are:

rContinue
The CICS event proceeded satisfactorily and normal processing is to resume.

rThrowException
The application program could not handle the CICS event and an exception is to be thrown.

rAbendTask
The application program could not handle the CICS event and the CICS task is to be abended.

ConditionType

Possible values are:

majorCode
The returned value is the CICS RESP value. This is one of the values in IccCondition::codes.

minorCode
The returned value is the CICS RESP2 value.

Chapter 43. IccResourceId class

IccBase
  IccResourceId

This is a base class from which **IccTransId** and other classes, whose names all end in Id, are derived. Many of these derived classes represent CICS resource names, such as a file control table (FCT) entry.

Header file: ICCRIDEH

IccResourceId constructors (protected)

Constructor (1)

IccResourceId (IccBase::ClassType *typ***,**
const IccResourceId& *id***)**

type
  An enumeration, defined in **IccBase** class, that indicates the type of class.

id
  A reference to an **IccResourceId** object that is used to create this object.

Constructor (2)

IccResourceId (IccBase::ClassType *type***,**
const char* *resName***)**

type
  An enumeration, defined in **IccBase** class, that indicates the type of class.

resName
  The name of a resource that is used to create this object.

Public methods

name

const char* name() const

Returns the name of the resource identifier as a string. Most **...Id** objects have 4- or 8-character names.

nameLength

unsigned short nameLength() const

Returns the length of the name returned by the **name** method.

Protected methods

operator=

IccResourceId& operator=(const IccResourceId& *id***)**

id
A reference to an **IccResourceId** object.

Set an **IccResourceId** object to be identical to *id*.

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 44. IccRRN class

IccBase
  IccRecordIndex
    IccRRN

An **IccRRN** object holds a relative record number and is used to identify records in VSAM RRDS files.

Header file: ICCRECEH

IccRRN constructors

Constructor

IccRRN(unsigned long *initRRN* **= 1)**

initRRN
The initial relative record number—an integer greater than 0. The default is 1.

Public methods

operator= (1)

IccRRN& operator=(const IccRRN& *rrn***)**

operator= (2)

IccRRN& operator=(unsigned long *num***)**

num
A relative record number—an integer greater than 0.

Assigns a new value for the relative record number.

operator== (1)

Icc::Bool operator== (const IccRRN& *rrn***) const**

operator== (2)

Icc::Bool operator== (unsigned long num) const

Tests equality

operator!= (1)

Icc::Bool operator!= (const IccRRN& *rrn***) const**

operator!= (2)

Icc::Bool operator!=(unsigned long num) const

Tests inequality

number

unsigned long number() const

Returns the relative record number.

Inherited public methods

Method	Class
className	IccBase
classType	IccBase
customClassNum	IccBase
length	IccRecordIndex
operator delete	IccBase
operator new	IccBase
type	IccRecordIndex
value	IccRecordIndex

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 45. IccSemaphore class

IccBase
  IccResource
    IccSemaphore

This class enables synchronization of resource updates.

Header file: ICCSEMEH

Sample: ICC$SEM

IccSemaphore constructor

Constructor (1)

IccSemaphore (const char* *resource*,
  LockType *type* **= byValue,**
  LifeTime *life* **= UOW)**

resource
  A text string, if *type* is byValue, otherwise an address in storage.
type
  An enumeration, defined in this class, that indicates whether locking is by value or by address. The default is by value.
life
  An enumeration, defined in this class, that indicates how long the semaphore lasts. The default is to last for the length of the UOW.

Constructor (2)

IccSemaphore (const IccLockId& *id*,
  LifeTime *life* **= UOW)**

id
  A reference to an **IccLockId** object
life
  An enumeration, defined in this class, that indicates how long the semaphore lasts. The default is to last for the length of the UOW.

Public methods

lifeTime

LifeTime lifeTime() const

Returns an enumeration, defined in this class, that indicates whether the lock lasts for the length of the current unit-of-work (UOW) or until the task terminates(task).

lock

void lock()

Attempts to get a lock. This method blocks if another task already owns the lock.

Conditions

ENQBUSY, LENGERR, INVREQ

tryLock

Icc::Bool tryLock()

Attempts to get a lock. This method does not block if another task already owns the lock. It returns a boolean that indicates whether it succeeded.

Conditions

ENQBUSY, LENGERR, INVREQ

type

LockType type() const

Returns an enumeration, defined in this class, that indicates what type of semaphore this is.

unlock

void unlock()

Release a lock.

Conditions

LENGERR, INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

LockType

byValue
The lock is on the contents (for example, name).

byAddress
The lock is on the memory address.

LifeTime

UOW The semaphore lasts for the length of the current unit of work.

task The semaphore lasts for the length of the task.

Chapter 46. IccSession class

IccBase
  IccResource
    IccSession

This class enables APPC and DTP programming.

Header file: ICCSESEH

Sample: ICC$SES1, ICC$SES2

IccSession constructors (public)

Constructor (1)

IccSession(const IccPartnerId& *id***)**

id
: A reference to an **IccPartnerId** object

Constructor (2)

IccSession (const IccSysId& *sysId***,**
const char* *profile* **= 0)**

sysId
: A reference to an **IccSysId** object that represents a remote CICS system

profile
: The 8-character name of the profile.

Constructor (3)

IccSession (const char* *sysName***,**
const char* *profile* **= 0)**

sysName
: The 4-character name of the remote CICS system with which this session is associated

profile
: The 8-character name of the profile.

IccSession constructor (protected)

Constructor

IccSession()

This constructor is for back end DTP CICS tasks that have a session as their principal facility. In this case the application program uses the **session** method on the **IccControl** object to gain access to their **IccSession** object.

Public methods

allocate

void allocate(AllocateOpt *option* **= queue)**

option
: An enumeration, defined in this class, that indicates what action CICS is to take if a communication channel is unavailable when this method is called.

Establishes a session (communication channel) to the remote system.

Conditions

INVREQ, SYSIDERR, CBIDERR, NETNAMEIDERR, PARTNERIDERR, SYSBUSY

connectProcess (1)

void connectProcess (SyncLevel *level***,**
const IccBuf* *PIP* **= 0)**

level
: An enumeration, defined in this class, that indicates what sync level is to be used for this conversation

PIP
: An optional pointer to an **IccBuf** object that contains the PIP data to be sent to the remote system

This method can only be used if an **IccPartnerId** object was used to construct this session object.

connectProcess (2)

void connectProcess (SyncLevel *level***,**
const IccTransId& *transId***,**
const IccBuf* *PIP* **= 0)**

level
: An enumeration, defined in this class, that indicates what sync level is to be used for this conversation

transId
: A reference to an **IccTransId** object that holds the name of the transaction to be started on the remote system

PIP
: An optional pointer to an **IccBuf** object that contains the PIP data to be sent to the remote system

connectProcess (3)

void connectProcess (SyncLevel *level***,**
const IccTPNameId& *TPName***,**
const IccBuf* *PIP* **= 0)**

level
: An enumeration, defined in this class, that indicates what sync level is to be used for this conversation

TPName
A reference to an **IccTPNameId** object that contains the 1–64 character TP name.

PIP
An optional pointer to an **IccBuf** object that contains the PIP data to be sent to the remote system

Starts a partner process on the remote system in preparation for sending and receiving information.

Conditions

INVREQ, LENGERR, NOTALLOC, PARTNERIDERR, NOTAUTH, TERMERR, SYSBUSY

converse

const IccBuf& converse(const IccBuf& *send***)**

send
A reference to an **IccBuf** object that contains the data that is to be sent.

converse sends the contents of *send* and returns a reference to an **IccBuf** object that holds the reply from the remote APPC partner.

Conditions

EOC, INVREQ, LENGERR, NOTALLOC, SIGNAL, TERMERR

convId

const IccConvId& convId()

Returns a reference to an **IccConvId** object that contains the 4-byte conversation identifier.

errorCode

const char* errorCode() const

Returns the 4-byte error code received when **isErrorSet** returns true. See the relevant DTP Guide for more information.

extractProcess

void extractProcess()

Retrieves information from an APPC conversation attach header and holds it inside the object. See **PIPList**, **process**, and **syncLevel** methods to retrieve the information from the object. This method should be used by the back end task if it wants access to the PIP data, the process name, or the synclevel under which it is running.

Conditions

INVREQ, NOTALLOC, LENGERR

flush

void flush()

Ensure that accumulated data and control information are transmitted on an APPC mapped conversation.

Conditions
INVREQ, NOTALLOC

free

void free()

Return the APPC session to CICS so that it may be used by other tasks.

Conditions
INVREQ, NOTALLOC

get

virtual const IccBuf& get()

A synonym for **receive**. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

isErrorSet

Icc::Bool isErrorSet() const

Returns a boolean variable, defined in **Icc** structure, that indicates whether an error has been set.

isNoDataSet

Icc::Bool isNoDataSet() const

Returns a boolean variable, defined in **Icc** structure, that indicates if no data was returned on a **send**—just control information.

isSignalSet

Icc::Bool isSignalSet() const

Returns a boolean variable, defined in **Icc** structure, that indicates whether a signal has been received from the remote process.

issueAbend

void issueAbend()

Abnormally ends the conversation. The partner transaction sees the TERMERR condition.

Conditions

INVREQ, NOTALLOC, TERMERR

issueConfirmation

void issueConfirmation()

Sends positive response to a partner's **send** request that specified the confirmation option.

Conditions

INVREQ, NOTALLOC, TERMERR, SIGNAL

issueError

void issueError()

Signals an error to the partner process.

Conditions

INVREQ, NOTALLOC, TERMERR, SIGNAL

issuePrepare

void issuePrepare()

This only applies to DTP over APPC links. It enables a syncpoint initiator to prepare a syncpoint slave for syncpointing by sending only the first flow (prepare to commit) of the syncpoint exchange.

Conditions

INVREQ, NOTALLOC, TERMERR

issueSignal

void issueSignal()

Signals that a mode change is needed.

Conditions

INVREQ, NOTALLOC, TERMERR

PIPList

IccBuf& PIPList()

Returns a reference to an **IccBuf** object that contains the PIP data sent from the front end process. A call to this method should be preceded by a call to **extractProcess** on back end DTP processes.

process

const IccBuf& process() const

Returns a reference to an **IccBuf** object that contains the process data sent from the front end process. A call to this method should be preceded by a call to **extractProcess** on back end DTP processes.

put

virtual void put(const IccBuf& *data***)**

data
: A reference to an **IccBuf** object that holds the data to be sent to the remote process.

A synonym for **send**. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

receive

const IccBuf& receive()

Returns a reference to an **IccBuf** object that contains the data received from the remote system.

Conditions

EOC, INVREQ, LENGERR, NOTALLOC, SIGNAL, TERMERR

send (1)

void send (const IccBuf& *send***,**
SendOpt *option* **=** normal**)**

send
: A reference to an **IccBuf** object that contains the data that is to be sent.

option
: An enumeration, defined in this class, that affects the behavior of the **send** method. The default is normal.

send (2)

void send(SendOpt *option* **=** normal**)**

option
: An enumeration, defined in this class, that affects the behavior of the **send** method. The default is normal.

Sends data to the remote partner.

Conditions

INVREQ, LENGERR, NOTALLOC, SIGNAL, TERMERR

sendInvite (1)

void sendInvite (const IccBuf& *send***,**
SendOpt *option* **=** normal**)**

send
A reference to an **IccBuf** object that contains the data that is to be sent.

option
An enumeration, defined in this class, that affects the behavior of the **sendInvite** method. The default is normal.

sendInvite (2)

void sendInvite(SendOpt *option* **=** normal**)**

option
An enumeration, defined in this class, that afffects the behavior of the **sendInvite** method. The default is normal.

Sends data to the remote partner and indicates a change of direction, that is, the next method on this object will be **receive**.

Conditions

INVREQ, LENGERR, NOTALLOC, SIGNAL, TERMERR

sendLast (1)

void sendLast (const IccBuf& *send***,**
SendOpt *option* **=** normal**)**

send
A reference to an **IccBuf** object that contains the data that is to be sent.

option
An enumeration, defined in this class, that affects the behavior of the **sendLast** method. The default is normal.

sendLast (2)

void sendLast(SendOpt *option* **=** normal**)**

option
An enumeration, defined in this class, that affects the behavior of the **sendLast** method. The default is normal.

Sends data to the remote partner and indicates that this is the final transmission. The **free** method must be invoked next, unless the sync level is 2, when you must commit resource updates before the **free**. (See **commitUOW** on page 215 in **IccTaskClass**).

Conditions

INVREQ, LENGERR, NOTALLOC, SIGNAL, TERMERR

state

IccValue::CVDA state(StateOpt *option* **=** lastCommand**)**

option
: An enumeration, defined in this class, that indicates how to report the state of the conversation

Returns a CVDA, defined in **IccValue** structure, that indicates the current state of the APPC conversation. Possible values are:

ALLOCATED
CONFFREE
CONFSEND
FREE
PENDFREE
PENDRECEIVE
RECEIVE
ROLLBACK
SEND
SYNCFREE
SYNCRECEIVE
SYNCSEND
NOTAPPLIC

IccValue::NOTAPPLIC is returned if there is no APPC conversation state.

Conditions

INVREQ, NOTALLOC

stateText

const char* stateText(StateOpt *option* **=** lastCommand**)**

option
: An enumeration, defined in this class, that indicates how to report the state of the conversation

Returns the symbolic name of the state that **state** method would return. For example, if **state** returns IccValue::ALLOCATED, **stateText** would return ALLOCATED.

syncLevel

SyncLevel syncLevel() const

Returns an enumeration, defined in this class, that indicates the synchronization level that is being used in this session. A call to this method should be preceded by a call to **extractProcess** on back end DTP processes.

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase

Method	Class
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

AllocateOpt

queue
If all available sessions are in use, CICS is to queue this request (and block the method) until it can allocate a session.

noQueue
Control is returned to the application if it cannot allocate a session. CICS raises the SYSBUSY condition.

Indicates whether queuing is required on an **allocate** method.

SendOpt

normal
The default.

confirmation
Indicates that a program using SyncLevel level1 or level2 requires a response from the remote partner program. The remote partner can respond positively, using the **issueConfirmation** method, or negatively, using the **issueError** method. The sending program does not receive control back from CICS until the response is received.

wait
Requests that the data is sent and not buffered internally. CICS is free to buffer requests to improve performance if this option is not specified.

StateOpt

Used to indicate how the state of a conversation is to be reported.

lastCommand
Return the state at the time of the completion of the last operation on the session.

extractState
Return the explicitly extracted current state.

SyncLevel

level0
Sync level 0

level1
Sync level 1

level2
Sync level 2

Chapter 47. IccStartRequestQ class

IccBase
  IccResource
    IccStartRequestQ

This is a singleton class that enables the application programmer to request an asynchronous start of another CICS transaction (see the **start** method on page 204).

An asynchronously started transaction uses the **IccStartRequestQ** class method **retrieveData** to gain the information passed to it by the transaction that issued the **start** request.

An unexpired start request can be cancelled by using the **cancel** method.

Header file: ICCSRQEH

Sample: ICC$SRQ1, ICC$SRQ2

IccStartRequestQ constructor (protected)

Constructor

IccStartRequestQ()

Public methods

cancel

void cancel (const IccRequestId& *reqId*,
  const IccTransId* *transId* **= 0)**

reqId
  A reference to an **IccRequestId** object that represents the request to be cancelled
transId
  An optional pointer to an **IccTransId** object that represents the transaction that is to be cancelled.

Cancels a previously issued **start** request that has not yet expired.

Conditions

ISCINVREQ, NOTAUTH, NOTFND, SYSIDERR

clearData

void clearData()

clearData clears the current data that is to be passed to the started transaction. The data was set using **setData** or **registerData**.

If the data was set using **registerData**, only the pointer to the data is removed, the data in the buffer is left unchanged.

If the data was set using **setData**, then **clearData** releases the memory used by the buffer.

data

const IccBuf& data() const

Returns a reference to an **IccBuf** object that contains data passed on a start request. A call to this method should be preceded by a call to **retrieveData** method.

instance

static IccStartRequestQ* instance()

Returns a pointer to the single **IccStartRequestQ** object. If the object does not exist it is created. See also **startRequestQ** method on page 113 of **IccControl**.

queueName

const char* queueName() const

Returns the name of the queue that was passed by the start requester. A call to this method should be preceded by a call to **retrieveData** method.

registerData

void registerData(const IccBuf* *buffer***)**

buffer
A pointer to the **IccBuf** object that holds data to be passed on a **start** request.

Registers an **IccBuf** object to be interrogated for start data on each subsequent **start** method invocation.

This just stores the address of the **IccBuf** object within the **IccStartRequestQ** so that the **IccBuf** object can be found when using the **start** method. This differs from the **setData** method, which takes a copy of the data held in the **IccBuf** object during the time that it is invoked.

reset

void reset()

Clears any associations previously made by **set...** methods in this class.

retrieveData

void retrieveData(RetrieveOpt *option* **=** noWait**)**

option
: An enumeration, defined in this class, that indicates what happens if there is no start data available.

Used by a task that was started, via an async start request, to gain access to the information passed by the start requester. The information is returned by the **data**, **queueName**, **returnTermId**, and **returnTransId** methods.

Conditions

ENDDATA, ENVDEFERR, IOERR, LENGERR, NOTFND, INVREQ

Note: The ENVDEFERR condition will be raised if all the possible options (**setData**, **setQueueName**, **setReturnTermId**, and **setReturnTransId**) are not used before issuing the **start** method. This condition is therefore not necessarily an error condition and your program should handle it accordingly.

returnTermId

const IccTermId& returnTermId() const

Returns a reference to an **IccTermId** object that identifies which terminal is involved in the session. A call to this method should be preceded by a call to **retrieveData** method.

returnTransId

const IccTransId& returnTransId() const

Returns a reference to an **IccTransId** object passed on a start request. A call to this method should be preceded by a call to **retrieveData** method.

setData

void setData(const IccBuf& *buf***)**

Copies the data in *buf* into the **IccStartRequestQ**, which passes it to the started transaction when the **start** method is called. See also **registerData** on page 202 for an alternative way to pass data to started transactions.

setQueueName

void setQueueName(const char* *queueName***)**

queueName
: An 8-character queue name.

Requests that this queue name be passed to the started transaction when the **start** method is called.

setReturnTermId (1)

void setReturnTermId(const IccTermId& *termId***)**

termId
A reference to an **IccTermId** object that identifies which terminal is involved in the session.

setReturnTermId (2)

void setReturnTermId(const char* *termName***)**

termName
The 4-character name of the terminal that is involved in the session.

Requests that this return terminal ID be passed to the started transaction when the **start** method is called.

setReturnTransId (1)

void setReturnTransId(const IccTransId& *transId***)**

transId
A reference to an **IccTransId** object.

setReturnTransId (2)

void setReturnTransId(const char* *transName***)**

transName
The 4-character name of the return transaction.

Requests that this return transaction ID be passed to the started transaction when the **start** method is called.

setStartOpts

void setStartOpts (ProtectOpt *popt* **=** none**,**
CheckOpt *copt* **=** check**)**

popt
An enumeration, defined in this class, that indicates whether start requests are to be protected

copt
An enumeration, defined in this class, that indicates whether start requests are to be checked.

Sets whether the started transaction is to have protection and whether it is to be checked.

start

const IccRequestId& start (const IccTransId& *transId***,**
const IccTermId* *termId***,**
const IccTime* *time* **= 0,**
const IccRequestId* *reqId* **= 0)**

or

const IccRequestId& start (const IccTransId& *transId***,**
const IccUserId* *userId***,**
const IccTime* *time* **= 0,**
const IccRequestId* *reqId* **= 0)**

or

const IccRequestId& start (const IccTransId& *transId***,**
const IccTime* *time* **= 0,**
const IccRequestId* *reqId* **= 0)**

transId
A reference to an **IccTransId** object that represents the transaction to be started

termId
A reference to an **IccTermId** object that identifies which terminal is involved in the session.

userId
A reference to an **IccUserId** object that represents the user ID.

time
An (optional) pointer to an **IccTime** object that specifies when the task is to be started. The default is for the task to be started immediately.

reqId
An (optional) pointer to an **IccRequestId** object that is used to identify this start request so that the **cancel** can cancel the request.

Asynchronously starts the named CICS transaction. The returned reference to an **IccRequestId** object identifies the **start** request and can be used subsequently to **cancel** the **start** request.

Conditions

INVREQ, IOERR, ISCINVREQ, LENGERR, NOTAUTH, SYSIDERR, TERMIDERR, TRANSIDERR, USERIDERR

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
className	IccBase
classType	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
isRouteOptionOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
routeOption	IccResource
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource

Method	Class
setEDF	IccResource
setRouteOption	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

RetrieveOpt

noWait
wait

ProtectOpt

none
protect

CheckOpt

check
noCheck

Chapter 48. IccSysId class

IccBase
  IccResourceId
    IccSysId

IccSysId class is used to identify a remote CICS system.

Header file: ICCRIDEH

IccSysId constructors

Constructor (1)

IccSysId(const char* *name*)

name
The 4-character name of the CICS system.

Constructor (2)

IccSysId(const IccSysId& *id*)

id A reference to an **IccSysId** object.

The copy constructor.

Public methods

operator= (1)

IccSysId& operator=(const IccSysId& *id*)

id A reference to an existing **IccSysId** object.

operator= (2)

IccSysId& operator=(const char* *name*)

name
The 4-character name of the CICS system.

Sets the name of the CICS system held in the object.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase

Method	Class
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 49. IccSystem class

IccBase
  IccResource
    IccSystem

This is a singleton class that represents the CICS system. It is used by an application program to discover information about the CICS system on which it is running.

Header file: ICCSYSEH

Sample: ICC$SYS

IccSystem constructor (protected)

Constructor

IccSystem()

Public methods

applName

const char* applName()

Returns the 8-character name of the CICS region.

Conditions

INVREQ

beginBrowse (1)

void beginBrowse (ResourceType *resource*,
const IccResourceId* *resId* **= 0)**

resource
: An enumeration, defined in this class, that indicates the type of resource to be browsed within the CICS system.

resId
: An optional pointer to an **IccResourceId** object that indicates the starting point for browsing through the resources.

beginBrowse (2)

void beginBrowse (ResourceType *resource*,
const char* *resName***)**

resource
: An enumeration, defined in this class, that indicates the type of resource to be browsed within the CICS system.

resName
The name of the resource that is to be the starting point for browsing the resources.

Signals the start of a browse through a set of CICS resources.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

dateFormat

const char* dateFormat()

Returns the default dateFormat for the CICS region.

Conditions

INVREQ

endBrowse

void endBrowse(ResourceType *resource***)**

Signals the end of a browse through a set of CICS resources.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

freeStorage

void freeStorage(void* *pStorage***)**

Releases the storage obtained by the **IccSystem getStorage** method.

Conditions

INVREQ

getFile (1)

IccFile* getFile(const IccFileId& *id***)**

id
A reference to an **IccFileId** object that identifies a CICS file.

getFile (2)

IccFile* getFile(const char* *fileName***)**

fileName
The name of a CICS file.

Returns a pointer to the **IccFile** object identified by the argument.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

getNextFile

IccFile* getNextFile()

This method is only valid after a successful **beginBrowse(IccSystem::file)** call. It returns the next file object in the browse sequence in the CICS system.

Conditions

END, FILENOTFOUND, ILLOGIC, NOTAUTH

getStorage

void* getStorage (unsigned long *size*,
char *initByte* **= -1,**
unsigned long *storageOpts* **= 0)**

size
The amount of storage being requested, in bytes

initByte
The initial setting of all bytes in the allocated storage

storageOpts
An enumeration, defined in **IccTask** class, that affects the way that CICS allocates storage.

Obtains a block of storage of the requested size and returns a pointer to it. The storage is not released automatically at the end of task; it is only released when a **freeStorage** operation is performed.

Conditions

LENGERR, NOSTG

instance

static IccSystem* instance()

Returns a pointer to the singleton **IccSystem** object. The object is created if it does not already exist.

operatingSystem

char operatingSystem()

Returns a 1-character value that identifies the operating system under which CICS is running:

A AIX
N Windows NT
P OS/2
X MVS/ESA

Conditions

NOTAUTH

operatingSystemLevel

unsigned short operatingSystemLevel()

Returns a halfword binary field giving the release number of the operating system under which CICS is running. The value returned is ten times the formal release number (the version number is not represented). For example, MVS/ESA Version 3 Release 2.1 would produce a value of 21.

Conditions

NOTAUTH

release

unsigned long release()

Returns the level of the CICS system as an integer set to 100 multiplied by the version number plus 10 multiplied by the release level. For example, CICS Transaction Server for z/OS [Version 1] Release 3 would return 130.

Conditions

NOTAUTH

releaseText

const char* releaseText()

Returns the same as **release**, except as a 4-character string. For example, CICS Transaction Server for z/OS [Version 1] Release 3 would return 0130.

Conditions

NOTAUTH

sysId

IccSysId& sysId()

Returns a reference to the **IccSysId** object that identifies this CICS system.

Conditions

INVREQ

workArea

const IccBuf& workArea()

Returns a reference to the **IccBuf** object that holds the work area for the CICS system.

Conditions

INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

ResourceType

autoInstallModel
connection
dataQueue
exitProgram
externalDataSet
file
journal
modename
partner
profile
program
requestId
systemDumpCode
tempStore
terminal
transactionDumpCode
transaction
transactionClass

Chapter 50. IccTask class

IccBase
  IccResource
    IccTask

IccTask is a singleton class used to invoke task related CICS services.

Header file: ICCTSKEH

Sample: ICC$TSK

IccTask Constructor (protected)

Constructor

IccTask()

Public methods

The opt parameter
Many methods have the same parameter, *opt*, which is described under the **abendCode** method inabendCode” on page 71.

abend

void abend (const char* *abendCode* **= 0,**
    AbendHandlerOpt *opt1* **=** respectAbendHandler**,**
    AbendDumpOpt *opt2* **=** createDump**)**

abendCode
  The 4-character abend code

opt1
  An enumeration, defined in this class, that indicates whether to respect or ignore any abend handling program specified by **setAbendHandler** method in **IccControl** class

opt2
  An enumeration, defined in this class, that indicates whether a dump is to be created.

Requests CICS to abend this task.

abendData

IccAbendData* abendData()

Returns a pointer to an **IccAbendData** object that contains information about the program abends, if any, that relate to this task.

commitUOW

void commitUOW()

Commit the resource updates within the current UOW for this task. This also causes a new UOW to start for subsequent resource update activity.

Conditions

INVREQ, ROLLEDBACK

delay

void delay (const IccTime& *time***,**
const IccRequestId* *reqId* **= 0)**

time
: A reference to an object that contains information about the delay time. The object can be one of these types:
  IccAbsTime
  : Expresses time as the number of milliseconds since the beginning of the year 1900.
  IccTimeInterval
  : Expresses an interval of time, such as 3 hours, 2 minutes, and 1 second.
  IccTimeOfDay
  : Expresses a time of day, such as 13 hours, 30 minutes (1-30 pm).

reqId
: An optional pointer to an **IccRequestId** object that can be used to cancel an unexpired delay request.

Requests that this task be delayed for an interval of time, or until a specific time.

Conditions

EXPIRED, INVREQ

dump

const char* dump (const char* *dumpCode***,**
const IccBuf* *buf* **= 0)**

dumpCode
: A 4-character label that identifies this dump

buf
: A pointer to the **IccBuf** object that contains additional data to be included in the dump.

Requests CICS to take a dump for this task. (See also **setDumpOpts**.) Returns the character identifier of the dump.

Conditions

INVREQ, IOERR, NOSPACE, NOSTG, NOTOPEN, OPENERR, SUPPRESSED

enterTrace

void enterTrace (unsigned short *traceNum***,**
const char* *resource* **= 0,**
IccBuf* *data* **= 0,**
TraceOpt *opt* **=** normal**)**

traceNum
The trace identifier for a user trace table entry; a value in the range 0 through 199.

resource
An 8-character name to be entered in the resource field of the trace table entry.

data
A pointer to the **IccBuf** object containing data to be included in the trace record.

opt
An enumeration, defined in this class, that indicates whether tracing should be normal or whether only exceptions should be traced.

Writes a user trace entry in the CICS trace table.

Conditions

INVREQ, LENGERR

facilityType

FacilityType facilityType()

Returns an enumeration, defined in this class, that indicates what type of principal facility this task has. This is usually a terminal, such as when the task was started by someone keying a transaction name on a CICS terminal. It is a session if the task is the back end of a mapped APPC conversation.

Conditions

INVREQ

freeStorage

void freeStorage(void* pStorage)

Releases the storage obtained by the **IccTask getStorage** method.

Conditions

INVREQ

getStorage

void* getStorage (unsigned long *size***,**
char *initByte* **= -1,**
unsigned short *storageOpts* **= 0)**

size
The amount of storage being requested, in bytes

initByte
The initial setting of all bytes in the allocated storage

storageOpts
An enumeration, defined in this class, that affects the way that CICS allocates storage.

Obtains a block of storage of the requested size. The storage is released automatically at the end of task, or when the **freeStorage** operation is performed. See also **getStorage** on page 211 in **IccSystem**class.

Conditions

LENGERR, NOSTG

instance

static IccTask* instance();

Returns a pointer to the singleton **IccTask** object. The object is created if it does not already exist.

isCommandSecurityOn

Icc::Bool isCommandSecurityOn()

Returns a boolean, defined in **Icc** structure, that indicates whether this task is subject to command security checking.

Conditions

INVREQ

isCommitSupported

Icc::Bool isCommitSupported()

Returns a boolean, defined in **Icc** structure that indicates whether this task can support the **commit** method. This method returns true in most environments; the exception to this is in a DPL environment (see **link** on page 166 in **IccProgram**).

Conditions

INVREQ

isResourceSecurityOn

Icc::Bool isResourceSecurityOn()

Returns a boolean, defined in **Icc** structure, that indicates whether this task is subject to resource security checking.

Conditions

INVREQ

isRestarted

Icc::Bool isRestarted()

Returns a boolean, defined in **Icc** structure, that indicates whether this task has been automatically restarted by CICS.

Conditions

INVREQ

isStartDataAvailable

Icc::Bool isStartDataAvailable()

Returns a boolean, defined in **Icc** structure, that indicates whether start data is available for this task. See the **retrieveData** method in **IccStartRequestQ** class if start data is available.

Conditions

INVREQ

number

unsigned long number() const

Returns the number of this task, unique within the CICS system.

principalSysId

IccSysId& principalSysId(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a reference to an **IccSysId** object that identifies the principal system identifier for this task.

Conditions

INVREQ

priority

unsigned short priority(Icc::GetOpt *opt* **=** Icc::object**)**

Returns the priority for this task.

Conditions

INVREQ

rollBackUOW

void rollBackUOW()

Roll back (backout) the resource updates associated with the current UOW within this task.

Conditions

INVREQ, ROLLEDBACK

setDumpOpts

void setDumpOpts(unsigned long *opts* **= dDefault)**

opts
An integer, made by adding or logically ORing values from the **DumpOpts** enumeration, defined in this class.

Set the dump options for this task. This method affects the behavior of the **dump** method defined in this class.

setPriority

void setPriority(unsigned short *pri***)**

pri
The new priority.

Changes the dispatch priority of this task.

Conditions

INVREQ

setWaitText

void setWaitText(const char* *name***)**

name
The 8-character string label that indicates why this task is waiting.

Sets the text that will appear when someone inquires on this task while it is suspended as a result of a **waitExternal** or **waitOnAlarm** method call.

startType

StartType startType()

Returns an enumeration, defined in this class, that indicates how this task was started.

Conditions

INVREQ

suspend

void suspend()

Suspend this task, allowing other tasks to be dispatched.

transId

const IccTransId& transId()

Returns the **IccTransId** object representing the transaction name of this CICS task.

triggerDataQueueId

const IccDataQueueId& triggerDataQueueId()

Returns a reference to the **IccDataQueueId** representing the trigger queue, if this task was started as a result of data arriving on an **IccDataQueue**. See **startType** method.

Conditions

INVREQ

userId

const IccUserId& userId(Icc::GetOpt *opt* **= Icc::object)**

opt
: An enumeration, defined in **Icc** structure, that indicates whether the information already existing in the object is to be used or whether it is to be refreshed from CICS.

Returns the ID of the user associated with this task.

Conditions

INVREQ

waitExternal

void waitExternal (long** *ECBList***,**
unsigned long *numEvents***,**
WaitPurgeability *opt* **=** purgeable**,**
WaitPostType *type* **=** MVSPost**)**

ECBList
: A pointer to a list of ECBs that represent events.

numEvents
: The number of events in *ECBList*.

opt
: An enumeration, defined in this class, that indicates whether the wait is purgeable.

type
: An enumeration, defined in this class, that indicates whether the post type is a standard MVS POST.

Waits for events that post ECBs - Event Control Blocks. The call causes the issuing task to be suspended until one of the ECBs has been posted—that is, one of the events has occurred. The task can wait on more than one ECB and can be dispatched as soon as any of them are posted.

See **waitExternal** in the *CICS Application Programming Reference* for more information about ECBs.

Conditions

INVREQ

waitOnAlarm

void waitOnAlarm(const IccAlarmRequestId& *id***)**

id
A reference to the **IccAlarmRequestId** object that identifies a particular alarm request.

Suspends the task until the alarm goes off (expires). See also **setAlarm** on page 101 in **IccClock**.

Conditions

INVREQ

workArea

IccBuf& workArea()

Returns a reference to the **IccBuf** object that holds the work area for this task.

Conditions

INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

AbendHandlerOpt

respectAbendHandler
Allows control to be passed to an abend handling program if one is in effect.

ignoreAbendHandler
Does not allow control to be passed to any abend handling program that may be in effect.

AbendDumpOpt

createDump
Take a transaction dump when servicing an abend request.
suppressDump
Do not take a transaction dump when servicing an abend request.

DumpOpts

The values may be added, or bitwise ORed, together to get the desired combination. For example `IccTask::dProgram + IccTask::dDCT + IccTask::dSIT`.
dDefault
dComplete
dTask
dStorage
dProgram
dTerminal
dTables
dDCT
dFCT
dPCT
dPPT
dSIT
dTCT
dTRT

FacilityType

none The task has no principal facility, that is, it is a background task.
terminal
This task has a terminal as its principal facility.
session
This task has a session as its principal facility, that is, it was probably started as a backend DTP program.
dataqueue
This task has a transient data queue as its principal facility.

StartType

DPL Distributed program link request
dataQueueTrigger
Trigger by data arriving on a data queue
startRequest
Started as a result of an asynchronous start request. See **IccStartRequestQ** class.
FEPIRequest
Front end programming interface. See *CICS/ESA: Front End Programming Interface User's Guide*, SC33-1175.
terminalInput
Started via a terminal input
CICSInternalTask
Started by CICS.

StorageOpts

ifSOSReturnCondition
: If insufficient space is available, return NOSTG condition instead of blocking the task.

below
: Allocate storage below the 16Mb line.

userDataKey
: Allocate storage in the USER data key.

CICSDataKey
: Allocate storage in the CICS data key.

TraceOpt

normal
: The trace entry is a standard entry.

exception
: The trace entry is an exception entry.

WaitPostType

MVSPost
: ECB is posted using the MVS POST service.

handPost
: ECB is hand posted (that is, using some method other than the MVS POST service).

WaitPurgeability

purgeable
: Task can be purged via a system call.

notPurgeable
: Task cannot be purged via a system call.

Chapter 51. IccTempStore class

IccBase
  IccResource
    IccTempStore

IccTempStore objects are used to manage the temporary storage of data. (**IccTempStore** data can exist between transaction calls.)

Header file: ICCTMPEH

Sample: ICC$TMP

IccTempStore constructors

Constructor (1)

IccTempStore (const IccTempStoreId& *id***,**
  Location *loc* **= auxStorage)**

id
  Reference to an **IccTempStoreId** object

loc
  An enumeration, defined in this class, that indicates where the storage is to be located when it is first created. The default is to use auxiliary storage (disk).

Constructor (2)

IccTempStore (const char* *storeName***,**
  Location *loc* **= auxStorage)**

storeName
  Specifies the 8-character name of the queue to be used. The name must be unique within the CICS system.

loc
  An enumeration, defined in this class, that indicates where the storage is to be located when it is first created. The default is to use auxiliary storage (disk).

Public methods

The opt parameter

Many methods have the same parameter, *opt*, which is described under the **abendCode** method inabendCode” on page 71.

clear

virtual void clear()

A synonym for **empty**. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

empty

void empty()

Deletes all the temporary data associated with the **IccTempStore** object and deletes the associated TD queue.

Conditions

INVREQ, ISCINVREQ, NOTAUTH, QIDERR, SYSIDERR

get

virtual const IccBuf& get()

A synonym for **readNextItem**. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

numberOfItems

unsigned short numberOfItems() const

Returns the number of items in temporary storage. This is only valid after a successful **writeItem** call.

put

virtual void put(const IccBuf& *buffer***)**

buffer
: A reference to an **IccBuf** object that contains the data that is to be added to the end of the temporary storage queue.

A synonym for **writeItem**. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

readItem

const IccBuf& readItem(unsigned short *itemNum***)**

itemNum
: Specifies the item number of the logical record to be retrieved from the queue.

Reads the specified item from the temporary storage queue and returns a reference to the **IccBuf** object that contains the information.

Conditions

INVREQ, IOERR, ISCINVREQ, ITEMERR, LENGERR, NOTAUTH, QIDERR, SYSIDERR

readNextItem

const IccBuf& readNextItem()

Reads the next item from a temporary storage queue and returns a reference to the **IccBuf** object that contains the information.

Conditions

INVREQ, IOERR, ISCINVREQ, ITEMERR, LENGERR, NOTAUTH, QIDERR, SYSIDERR

rewriteItem

void rewriteItem (unsigned short *itemNum***,**
const IccBuf& *item***,**
NoSpaceOpt *opt* **=** suspend**)**

The parameters are:

itemNum
Specifies the item number of the logical record that is to be modified

item
The name of the **IccBuf** object that contains the update data.

opt
An enumeration, defined in this class, that indicates whether the application program is to be suspended if a shortage of space in the queue prevents the record being added. suspend is the default.

This method updates the specified item in the temporary storage queue.

Conditions

INVREQ, IOERR, ISCINVREQ, ITEMERR, LENGERR, NOSPACE, NOTAUTH, QIDERR, SYSIDERR

writeItem (1)

unsigned short writeItem (const IccBuf& *item***,**
NoSpaceOpt *opt* **=** suspend**)**

item
The name of the **IccBuf** object that contains the data that is to added to the end of the temporary storage queue.

opt
An enumeration, defined in this class, that indicates whether the application program is to be suspended if a shortage of space in the queue prevents the record being added. suspend is the default.

writeItem (2)

unsigned short writeItem (const char* *text***,**
NoSpaceOpt *opt* **=** suspend**)**

text
The text string that is to added to the end of the temporary storage queue.

opt
An enumeration, defined in this class, that indicates whether the application program is to be suspended if a shortage of space in the queue prevents the record being added. suspend is the default.

This method adds a new record at the end of the temporary storage queue. The returned value is the item number that was created (if this was done successfully).

Conditions

INVREQ, IOERR, ISCINVREQ, ITEMERR, LENGERR, NOSPACE, NOTAUTH, QIDERR, SYSIDERR

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
className	IccBase
classType	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
isRouteOptionOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
routeOption	IccResource
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource
setRouteOption	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

Location

auxStorage
: Temporary store data is to reside in auxiliary storage (disk).

memory
: Temporary store data is to reside in memory.

NoSpaceOpt

What action to take if a shortage of space in the queue prevents the record being added immediately.

suspend
: Suspend the application program.

returnCondition
: Do not suspend the application program, but raise the NOSPACE condition instead.

Chapter 52. IccTempStoreId class

IccBase
    IccResourceId
        IccTempStoreId

IccTempStoreId class is used to identify a temporary storage name in the CICS system. This is an entry in the TST (temporary storage table).

Header file: ICCRIDEH

IccTempStoreId constructors

Constructor (1)

IccTempStoreId(const char* *name***)**

name
: The 8-character name of the temporary storage entry.

Constructor (2)

IccTempStoreId(const IccTempStoreId& *id***)**

id
: A reference to an **IccTempStoreId** object.

The copy constructor.

Public methods

operator= (1)

IccTempStoreId& operator=(const char* *name***)**

name
: The 8-character name of the temporary storage entry.

operator= (2)

IccTempStoreId& operator=(const IccTempStoreId& *id***)**

id
: A reference to an **IccTempStoreId** object.

Assigns a new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase

Method	Class
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 53. IccTermId class

IccBase
IccResourceId
IccTermId

IccTermId class is used to identify a terminal name in the CICS system. This is an entry in the TCT (terminal control table).

Header file: ICCRIDEH

IccTermId constructors

Constructor (1)

IccTermId(const char* *name***)**

name
The 4-character name of the terminal

Constructor (2)

IccTermId(const IccTermId& *id***)**

id
A reference to an **IccTermId** object.

The copy constructor.

Public methods

operator= (1)

IccTermId& operator=(const char* *name***)**

name
The 4-character name of the terminal

operator= (2)

IccTermId& operator=(const IccTermId& *id***)**

id
A reference to an **IccTermId** object.

Assigns a new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase

Method	Class
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 54. IccTerminal class

IccBase
  IccResource
    IccTerminal

This is a singleton class that represents the terminal that belongs to the CICS task. It can only be created if the transaction has a 3270 terminal as its principal facility, otherwise an exception is thrown.

Header file: ICCTRMEH

Sample: ICC$TRM

IccTerminal constructor (protected)

Constructor

IccTerminal()

Public methods

The opt parameter

Many methods have the same parameter, *opt*, which is described under the **abendCode** method inabendCode” on page 71.

AID

AIDVal AID()

Returns an enumeration, defined in this class, that indicates which AID (action identifier) key was last pressed at this terminal.

clear

virtual void clear()

A synonym for **erase**. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

cursor

unsigned short cursor()

Returns the current cursor position as an offset from the top left corner of the screen.

data

IccTerminalData* data()

Returns a pointer to an **IccTerminalData** object that contains information about the characteristics of the terminal. The object is created if it does not already exist.

erase

void erase()

Erase all the data displayed at the terminal.

Conditions
INVREQ, INVPARTN

freeKeyboard

void freeKeyboard()

Frees the keyboard so that the terminal can accept input.

Conditions
INVREQ, INVPARTN

get

virtual const IccBuf& get()

A synonym for **receive**. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

height

unsigned short height(Icc::getopt *opt* **= Icc::object)**

Returns how many lines the screen holds.

Conditions
INVREQ

inputCursor

unsigned short inputCursor()

Returns the position of the cursor on the screen.

instance

static IccTerminal* instance()

Returns a pointer to the single **IccTerminal** object. The object is created if it does not already exist.

line

unsigned short line()

Returns the current line number of the cursor from the top of the screen.

netName

const char* netName()

Returns the 8-byte string representing the network logical unit name of the principal facility.

operator<< (1)

IccTerminal& operator << (Color *color***)**

Sets the foreground color for data subsequently sent to the terminal.

operator<< (2)

IccTerminal& operator << (Highlight *highlight***)**

Sets the highlighting used for data subsequently sent to the terminal.

operator<< (3)

IccTerminal& operator << (const IccBuf& *buffer***)**

Writes another buffer.

operator<< (4)

IccTerminal& operator << (char *ch***)**

Writes a character.

operator<< (5)

IccTerminal& operator << (signed char *ch***)**

Writes a character.

operator<< (6)

IccTerminal& operator << (unsigned char *ch***)**

Writes a character.

operator<< (7)

IccTerminal& operator << (const char* *text***)**

Writes a string.

operator<< (8)

IccTerminal& operator << (const signed char* *text***)**

Writes a string.

operator<< (9)

IccTerminal& operator << (const unsigned char* *text***)**

Writes a string.

operator<< (10)

IccTerminal& operator << (short *num***)**

Writes a short.

operator<< (11)

IccTerminal& operator << (unsigned short *num***)**

Writes an unsigned short.

operator<< (12)

IccTerminal& operator << (long *num***)**

Writes a long.

operator<< (13)

IccTerminal& operator << (unsigned long *num***)**

Writes an unsigned long.

operator<< (14)

IccTerminal& operator << (int *num***)**

Writes an integer.

operator<< (15)

IccTerminal& operator << (float *num***)**

Writes a float.

operator<< (16)

IccTerminal& operator << (double *num***)**

Writes a double.

operator<< (17)

IccTerminal& operator << (long double *num***)**

Writes a long double.

operator<< (18)

IccTerminal& operator << (IccTerminal& (*f)(IccTerminal&))

Enables the following syntax:

```
Term << Hello World << endl;
Term << Hello again << flush;
```

put

virtual void put(const IccBuf& *buf***)**

A synonym for **sendLine**. See Polymorphic Behavior˝ on page 59 for information on polymorphism.

receive

const IccBuf& receive(Case *caseOpt* **=** upper**)**

caseOpt
An enumeration, defined in this class, that indicates whether text is to be converted to upper case or left as it is.

Receives data from the terminal

Conditions

EOC, INVREQ, LENGERR, NOTALLOC, SIGNAL, TERMERR

receive3270Data

const IccBuf& receive3270Data(Case *caseOpt* **=** upper**)**

caseOpt
An enumeration, defined in this class, that indicates whether text is to be converted to upper case or left as it is.

Receives the 3270 data buffer from the terminal

Conditions

INVREQ, LENGERR, TERMERR

send (1)

void send(const IccBuf& *buffer***)**

buffer
A reference to an **IccBuf** object that holds the data that is to be sent.

send (2)

void send (const char* *format***,**
...)

format
A format string, as in the **printf** standard library function.

...
The optional arguments that accompany *format*.

send (3)

void send (unsigned short *row***,**
unsigned short *col***,**
const IccBuf& *buffer***)**

row
The row where the writing of the data is started.

col
The column where the writing of the data is started.

buffer
A reference to an **IccBuf** object that holds the data that is to be sent.

send (4)

void send (unsigned short *row***,**
unsigned short *col***,**
const char* *format***,**
...)

row
The row where the writing of the data is started.

col
The column where the writing of the data is started.

format
A format string, as in the **printf** standard library function.

...
The optional arguments that accompany *format*.

Writes the specified data to either the current cursor position or to the cursor position specified by the arguments.

Conditions

INVREQ, LENGERR, TERMERR

send3270Data (1)

void send3270Data(const IccBuf& *buffer***)**

buffer
A reference to an **IccBuf** object that holds the data that is to be sent.

send3270Data (2)

void send3270 Data(const char* *format***,**
...)

format
A format string, as in the **printf** standard library function

...
The optional arguments that accompany *format*.

send3270Data (3)

void send3270Data (unsigned short *col***,**
const IccBuf& *buf***)**

col
The column where the writing of the data is started

buffer
A reference to an **IccBuf** object that holds the data that is to be sent.

send3270Data (4)

void send3270Data (unsigned short *col***,**
const char* *format***,**
...)

col
The column where the writing of the data is started

format
A format string, as in the **printf** standard library function

...
The optional arguments that accompany *format*.

Writes the specified data to either the next line of the terminal or to the specified column of the current line.

Conditions

INVREQ, LENGERR, TERMERR

sendLine (1)

void sendLine(const IccBuf& *buffer***)**

buffer
A reference to an **IccBuf** object that holds the data that is to be sent.

sendLine (2)

void sendLine (const char* *format***,**
...)

format
A format string, as in the **printf** standard library function

...
The optional arguments that accompany *format*.

sendLine (3)

void sendLine (unsigned short *col***,**
const IccBuf& *buf***)**

col
The column where the writing of the data is started

buffer
A reference to an **IccBuf** object that holds the data that is to be sent.

sendLine (4)

void sendLine (unsigned short *col***,**
const char* *format***,**
...)

col
The column where the writing of the data is started

format
A format string, as in the **printf** standard library function

...
The optional arguments that accompany *format*.

Writes the specified data to either the next line of the terminal or to the specified column of the current line.

Conditions

INVREQ, LENGERR, TERMERR

setColor

void setColor(Color *color***=defaultColor)**

color
An enumeration, defined in this class, that indicates the color of the text that is written to the screen.

Changes the color of the text subsequently sent to the terminal.

setCursor (1)

void setCursor(unsigned short *offset***)**

offset
The position of the cursor where the top left corner is 0.

setCursor (2)

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

row
: The row number of the cursor where the top row is 1

col
: The column number of the cursor where the left column is 1

Two different ways of setting the position of the cursor on the screen.

Conditions

INVREQ, INVPARTN

setHighlight

void setHighlight(Highlight *highlight* = normal)

highlight
: An enumeration, defined in this class, that indicates the highlighting of the text that is written to the screen.

Changes the higlighting of the data subsequently sent to the terminal.

setLine

void setLine(unsigned short *lineNum* = 1)

lineNum
: The line number, counting from the top.

Moves the cursor to the start of line *lineNum*, where 1 is the top line of the terminal. The default is to move the cursor to the start of line 1.

Conditions

INVREQ, INVPARTN

setNewLine

void setNewLine(unsigned short *numLines* = 1)

numLines
: The number of blank lines.

Requests that *numLines* blank lines be sent to the terminal.

Conditions

INVREQ, INVPARTN

setNextCommArea

void setNextCommArea(const IccBuf& *commArea*)

commArea
: A reference to the buffer that is to be used as a COMMAREA.

Specifies the COMMAREA that is to be passed to the next transaction started on this terminal.

setNextInputMessage

void setNextInputMessage(const IccBuf& *message***)**

message
: A reference to the buffer that holds the input message.

Specifies data that is to be made available, by the **receive** method, to the next transaction started at this terminal.

setNextTransId

void setNextTransId (const IccTransId& *transid***,**
NextTransIdOpt *opt* **=** queue**)**

transid
: A reference to the **IccTransId** object that holds the name of a transaction

opt
: An enumeration, defined in this class, that indicates whether *transId* should be queued or started immediately (that is, it should be the very next transaction) at this terminal.

Specifies the next transaction that is to be started on this terminal.

signoff

void signoff()

Signs off the user who is currently signed on. Authority reverts to the default user.

Conditions

INVREQ

signon (1)

void signon (const IccUserId& *id***,**
const char* *password* **= 0,**
const char* *newPassword* **= 0)**

id
: A reference to an **IccUserId** object

password
: The 8-character existing password.

newPassword
: An optional 8-character new password.

signon (2)

void signon (IccUser& *user***,**
const char* *password* **= 0,**
const char* *newPassword* **= 0)**

user
A reference to an **IccUser** object

password
The 8-character existing password.

newPassword
An optional 8-character new password. This method differs from the first **signon** method in that the **IccUser** object is interrogated to discover **IccGroupId** and language information. The object is also updated with language and ESM return and response codes.

Signs the user on to the terminal.

Conditions

INVREQ, NOTAUTH, USERIDERR

waitForAID (1)

AIDVal waitForAID()

Waits for any input and returns an enumeration, defined in this class, that indicates which AID key is expected.

waitForAID (2)

void waitForAID(AIDVal *aid***)**

aid
An enumeration, defined in this class, that indicates which AID key was last pressed.

Waits for the specified AID key to be pressed, before returning control. This method loops, receiving input from the terminal, until the correct AID key is pressed by the operator.

Conditions

EOC, INVREQ, LENGERR, NOTALLOC, SIGNAL, TERMERR

width

unsigned short width(Icc::getopt *opt* **=** Icc::object**)**

Returns the width of the screen in characters.

Conditions

INVREQ

workArea

IccBuf& workArea()

Returns a reference to the **IccBuf** object that holds the terminal work area.

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

AIDVal

ENTER
CLEAR
PA1 to PA3
PF1 to PF24

Case

upper
mixed

Color

defaultColor
blue
red
pink
green
cyan
yellow
neutral

Highlight

defaultHighlight
blink
reverse
underscore

NextTransIdOpt

queue
: Queue the transaction with any other outstanding starts queued on the terminal.

immediate
: Start the transaction immediately, that is, before any other outstanding starts queued on the terminal.

Chapter 55. IccTerminalData class

IccBase
    IccResource
        IccTerminalData

IccTerminalData is a singleton class owned by **IccTerminal** (see **data** on page 233 in **IccTerminal** class). **IccTerminalData** contains information about the terminal characteristics.

Header file: ICCTMDEH

Sample: ICC$TRM

IccTerminalData constructor (protected)

Constructor

IccTerminalData()

Public methods

> **The opt parameter**
> Many methods have the same parameter, *opt*, which is described under the **abendCode** method inabendCode” on page 71.

alternateHeight

unsigned short alternateHeight(Icc::GetOpt *opt* **=** Icc::object**)**

opt
: An enumeration that indicates whether the information in the object should be refreshed from CICS before being extracted. The default is not to refresh.

Returns the alternate height of the screen, in lines.

Conditions
INVREQ

alternateWidth

unsigned short alternateWidth(Icc::GetOpt *opt* **=** Icc::object**)**

Returns the alternate width of the screen, in characters.

Conditions
INVREQ

defaultHeight

unsigned short defaultHeight(Icc::GetOpt *opt* **=** Icc::object**)**

Returns the default height of the screen, in lines.

Conditions

INVREQ

defaultWidth

unsigned short defaultWidth(Icc::GetOpt *opt* **=** Icc::object**)**

Returns the default width of the screen, in characters.

Conditions

INVREQ

graphicCharCodeSet

unsigned short graphicCharCodeSet(Icc::GetOpt *opt* **=** Icc::object**)**

Returns the binary code page global identifier as a value in the range 1 to 65534, or 0 for a non-graphics terminal.

Conditions

INVREQ

graphicCharSetId

unsigned short graphicCharSetId(Icc::GetOpt *opt* **=** Icc::object**)**

Returns the graphic character set global identifier as a number in the range 1 to 65534, or 0 for a non-graphics terminal.

Conditions

INVREQ

isAPLKeyboard

Icc::Bool isAPLKeyboard(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal has the APL keyboard feature.

Conditions

INVREQ

isAPLText

Icc::Bool isAPLText(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal has the APL text feature.

Conditions
INVREQ

isBTrans

Icc::Bool isBTrans(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal has the background transparency capability.

Conditions
INVREQ

isColor

Icc::Bool isColor(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal has the extended color capability.

Conditions
INVREQ

isEWA

Icc::Bool isEWA(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a Boolean that indicates whether the terminal supports Erase Write Alternative.

Conditions
INVREQ

isExtended3270

Icc::Bool isExtended3270(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a Boolean that indicates whether the terminal supports the 3270 extended data stream.

Conditions
INVREQ

isFieldOutline

Icc::Bool isFieldOutline(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal supports field outlining.

Conditions
INVREQ

isGoodMorning

Icc::Bool isGoodMorning(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal has a good morning message.

Conditions

INVREQ

isHighlight

Icc::Bool isHighlight(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal has extended highlight capability.

Conditions

INVREQ

isKatakana

Icc::Bool isKatakana(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal supports Katakana.

Conditions

INVREQ

isMSRControl

Icc::Bool isMSRControl(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal supports magnetic slot reader control.

Conditions

INVREQ

isPS

Icc::Bool isPS(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal supports programmed symbols.

Conditions

INVREQ

isSOSI

Icc::Bool isSOSI(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal supports mixed EBCDIC/DBCS fields.

Conditions
INVREQ

isTextKeyboard

Icc::Bool isTextKeyboard(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal supports TEXTKYBD.

Conditions
INVREQ

isTextPrint

Icc::Bool isTextPrint(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal supports TEXTPRINT.

Conditions
INVREQ

isValidation

Icc::Bool isValidation(Icc::GetOpt *opt* **=** Icc::object**)**

Returns a boolean that indicates whether the terminal supports validation.

Conditions
INVREQ

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 56. IccTime class

IccBase
  IccResource
    IccTime

IccTime is used to contain time information and is the base class from which **IccAbsTime**, **IccTimeInterval**, and **IccTimeOfDay** classes are derived.

Header file: ICCTIMEH

IccTime constructor (protected)

Constructor

IccTime (unsigned long *hours* **= 0,**
  unsigned long *minutes* **= 0,**
  unsigned long *seconds* **= 0)**

hours
  The number of hours

minutes
  The number of minutes

seconds
  The number of seconds

Public methods

hours

virtual unsigned long hours() const

Returns the hours component of time—the value specified in the constructor.

minutes

virtual unsigned long minutes() const

Returns the minutes component of time—the value specified in the constructor.

seconds

virtual unsigned long seconds() const

Returns the seconds component of time—the value specified in the constructor.

timeInHours

virtual unsigned long timeInHours()

Returns the time in hours.

timeInMinutes

virtual unsigned long timeInMinutes()

Returns the time in minutes.

timeInSeconds

virtual unsigned long timeInSeconds()

Returns the time in seconds.

type

Type type() const

Returns an enumeration, defined in this class, that indicates what type of subclass of **IccTime** this is.

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
className	IccBase
classType	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
isEDFOn	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Enumerations

Type

absTime
The object is of **IccAbsTime** class. It is used to represent a current date and time as the number of milliseconds that have elapsed since the beginning of the year 1900.

timeInterval
The object is of **IccTimeInterval** class. It is used to represent a length of time, such as 5 minutes.

timeOfDay
The object is of **IccTimeOfDay** class. It is used to represent a particular time of day, such as midnight.

Chapter 57. IccTimeInterval class

IccBase
  IccResource
    IccTime
      IccTimeInterval

This class holds information about a time interval.

Header file: ICCTIMEH

IccTimeInterval constructors

Constructor (1)

IccTimeInterval (unsigned long *hours* **= 0,**
unsigned long *minutes* **= 0,**
unsigned long *seconds* **= 0)**

hours
  The initial hours setting. The default is 0.
minutes
  The initial minutes setting. The default is 0.
seconds
  The initial seconds setting. The default is 0.

Constructor (2)

IccTimeInterval(const IccTimeInterval& *time***)**

The copy constructor.

Public methods

operator=

IccTimeInterval& operator=(const IccTimeInterval& *timeInterval***)**

Assigns one **IccTimeInterval** object to another.

set

void set (unsigned long *hours***,**
unsigned long *minutes***,**
unsigned long *seconds***)**

hours
  The new hours setting
minutes
  The new minutes setting
seconds
  The new seconds setting

Changes the time held in the **IccTimeInterval** object.

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
hours	IccTime
isEDFOn	IccResource
minutes	IccTime
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource
timeInHours	IccTime
timeInMinutes	IccTime
timeInSeconds	IccTime
type	IccTime

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 58. IccTimeOfDay class

IccBase
  IccResource
    IccTime
      IccTimeOfDay

This class holds information about the time of day.

Header file: ICCTIMEH

IccTimeOfDay constructors

Constructor (1)

IccTimeOfDay (unsigned long *hours* **= 0,**
unsigned long *minutes* **= 0,**
unsigned long *seconds* **= 0)**

hours
The initial hours setting. The default is 0.

minutes
The initial minutes setting. The default is 0.

seconds
The initial seconds setting. The default is 0.

Constructor (2)

IccTimeOfDay(const IccTimeOfDay& *time***)**

The copy constructor

Public methods

operator=

IccTimeOfDay& operator=(const IccTimeOfDay& *timeOfDay***)**

Assigns one **IccTimeOfDay** object to another.

set

void set (unsigned long *hours***,**
unsigned long *minutes***,**
unsigned long *seconds***)**

hours
The new hours setting

minutes
The new minutes setting

seconds
The new seconds setting

Changes the time held in the **IccTimeOfDay** object.

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
hours	IccTime
isEDFOn	IccResource
minutes	IccTime
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource
timeInHours	IccTime
timeInMinutes	IccTime
timeInSeconds	IccTime
type	IccTime

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 59. IccTPNameId class

IccBase
  IccResourceId
    IccTPNameId

IccTPNameId class holds a 1-64 byte TP partner name.

Header file: ICCRIDEH

IccTPNameId constructors

Constructor (1)

IccTPNameId(const char* *name***)**

name
  The 1- to 64-character TP name.

Constructor (2)

IccTPNameId(const IccTPNameId& *id***)**

id A reference to an **IccTPNameId** object.
The copy constructor.

Public methods

operator= (1)

IccTPNameId& operator=(const char* *name***)**

name
  The 1- to 64-character TP name.

operator= (2)

IccTPNameId& operator=(const IccTPNameId& *id***)**

id A reference to an **IccTPNameId** object.
Assigns a new value.

Inherited public methods

Method	**Class**
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase

Method	Class
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 60. IccTransId class

IccBase
  IccResourceId
    IccTransId

IccTransId class identifies a transaction name in the CICS system. This is an entry in the PCT (Program Control Table).

Header file: ICCRIDEH

IccTransId constructors

Constructor (1)

IccTransId(const char* *name***)**

name
  The 4-character transaction name.

Constructor (2)

IccTransId(const IccTransId& *id***)**

id
  A reference to an **IccTransId** object.

The copy constructor.

Public methods

operator= (1)

IccTransId& operator=(const char* *name***)**

name
  The 4-character transaction name.

operator= (2)

IccTransId& operator=(const IccTransId& *id***)**

id
  A reference to an **IccTransId** object.

Assigns a new value.

Inherited public methods

Method	**Class**
classType	IccBase
className	IccBase
customClassNum	IccBase

Method	Class
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 61. IccUser class

IccBase
    IccResource
        IccUser

This class represents a CICS user.

Header file: ICCUSREH

Sample: ICC$USR

IccUser constructors

Constructor (1)

IccUser (const IccUserId& *id*,
    const IccGroupId* *gid* **= 0)**

id
    A reference to an **IccUserId** object that contains the user ID name
gid
    An optional pointer to an **IccGroupId** object that contains information about the users group ID.

Constructor (2)

IccUser (const char* *userName*,
    const char* *groupName* **= 0)**

userName
    The 8-character user ID
gid
    The optional 8-character group ID.

Public methods

changePassword

void changePassword (const char* *password*,
    const char* *newPassword***)**

password
    The users existing password—a string of up to 8 characters
newPassword
    The users new password—a string of up to 8 characters.

Attempts to change the user's password.

Conditions

INVREQ, NOTAUTH, USERIDERR

daysUntilPasswordExpires

unsigned short daysUntilPasswordExpires() const

Returns the number of days before the password expires. This method is valid after a successful **verifyPassword** method call in this class.

ESMReason

unsigned long ESMReason() const

Returns the external security reason code of interest if a **changePassword** or **verifyPassword** method call is unsuccessful.

ESMResponse

unsigned long ESMResponse() const

Returns the external security response code of interest if a **changePassword** or **verifyPassword** method call is unsuccessful.

groupId

const IccGroupId& groupId() const

Returns a reference to the **IccGroupId** object that holds information on the users group ID.

invalidPasswordAttempts

unsigned long invalidPasswordAttempts() const

Returns the number of times the wrong password has been entered for this user since the last successful signon. This method should only be used after a successful **verifyPassword** method.

language

const char* language() const

Returns the user's language after a successful call to **signon** in **IccTerminal**.

lastPasswordChange

const IccAbsTime& lastPasswordChange() const

Returns a reference to an **IccAbsTime** object that holds the time when the password was last changed. This method should only be used after a successful **verifyPassword** method.

lastUseTime

const IccAbsTime& lastUseTime() const

Returns a reference to an **IccAbsTime** object that holds the time when the user ID was last used. This method should only be used after a successful **verifyPassword** method.

passwordExpiration

const IccAbsTime& passwordExpiration() const

Returns a reference to an **IccAbsTime** object that holds the time when the password will expire. This method should only be used after a successful **verifyPassword** method.

setLanguage

void setLanguage(const char* *language***)**

Sets the IBM-defined national language code that is to be associated with this user. This should be a three character value.

verifyPassword

void verifyPassword(const char* *password***)**

Checks that the supplied password matches the password recorded by the external security manager for this **IccUser**.

Conditions

INVREQ, NOTAUTH, USERIDERR

Inherited public methods

Method	Class
actionOnCondition	IccResource
actionOnConditionAsChar	IccResource
actionsOnConditionsText	IccResource
classType	IccBase
className	IccBase
condition	IccResource
conditionText	IccResource
customClassNum	IccBase
handleEvent	IccResource
id	IccResource
isEDFOn	IccResource
name	IccResource
operator delete	IccBase
operator new	IccBase
setActionOnAnyCondition	IccResource
setActionOnCondition	IccResource
setActionsOnConditions	IccResource
setEDF	IccResource

Inherited protected methods

Method	Class
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 62. IccUserId class

IccBase
  IccResourceId
    IccUserId

IccUserId class represents an 8-character user name.

Header file: ICCRIDEH

IccUserId constructors

Constructor (1)

IccUserId(const char* *name*)

name
  The 8-character name of the user ID.

Constructor (2)

IccUserId(const IccUserId& *id*)

id A reference to an **IccUserId** object.
The copy constructor.

Public methods

operator= (1)

IccUserId& operator=(const char* *name*)

name
  The 8-character name of the user ID.

operator= (2)

IccUserId& operator=(const IccUserId& *id*)

id A reference to an **IccUserId** object.
Assigns a new value.

Inherited public methods

Method	Class
classType	IccBase
className	IccBase
customClassNum	IccBase
name	IccResourceId
nameLength	IccResourceId
operator delete	IccBase

Method	Class
operator new	IccBase

Inherited protected methods

Method	Class
operator=	IccResourceId
setClassName	IccBase
setCustomClassNum	IccBase

Chapter 63. IccValue structure

This structure contains CICS-value data areas (CVDAs) as an enumeration.

Header file: ICCVALEH

Enumeration

CVDA

Valid CVDAs are:

ACQFAIL	ACQUIRED	ACQUIRING	ACTIVE
ADD	ADDABLE	ADDFAIL	ADVANCE
ALARM	ALLCONN	ALLOCATED	ALLQUERY
ALTERABLE	ALTERNATE	ALTPRTCOPY	ANY
APLKYBD	APLTEXT	APPC	APPCPARALLEL
APPCSINGLE	APPLICATION	ASACTL	ASCII7
ASCII8	ASSEMBLER	ATI	ATTENTION
AUDALARM	AUTOACTIVE	AUTOARCH	AUTOCONN
AUTOINACTIVE	AUTOPAGEABLE	AUTOSTART	AUXILIARY
AUXPAUSE	AUXSTART	AUXSTOP	AVAILABLE
BACKOUT	BACKTRANS	BACKUPNONBWO	BASE
BASESPACE	BATCHLU	BDAM	BEGINSESSION
BELOW	BGAM	BIPROG	BISYNCH
BLK	BLOCKED	BROWSABLE	BSAM
	BUSY	C	CACHE
CANCEL	CANCELLED	CD	CDRDLPRT
CEDF	CICS	CICSDATAKEY	CICSEXECKEY
CICSSECURITY	CICSTABLE	CLEAR	CLOSED
CLOSEFAILED	CLOSELEAVE	CLOSEREQUEST	CLOSING
CMDPROT	CMDSECEXT	CMDSECNO	CMDSECYES
COBOL	COBOLII	COLDACQ	COLDQUERY
COLDSTART	COLOR	COMMIT	COMMITFAIL
CONFFREE	CONFRECEIVE	CONFSEND	CONNECTED
CONNECTION	CONSISTENT	CONSOLE	CONTNLU
CONTROLSHUT	CONVERSE	CONVIDLE	COORDINATOR
COPY	CREATE	CRITICAL	CTLGALL
CTLGMODIFY	CTLGNONE	CTRLABLE	CURRENT
DAE	DATA	DATASET	DATASETFULL
DATASTREAM	DB2®	DEADLOCK	DEC
DEFAULT	DEFRESP1	DEFRESP1OR2	DEFRESP2
DEFRESP3	DELAY	DELETABLE	DELETEFAIL
DELEXITERROR	DEREGERROR	DEREGISTERED	DEST
DISABLED	DISABLING	DISCARDFAIL	DISCREQ
DISK1	DISK2	DISK2PAUSE	DISPATCHABLE
DPLSUBSET	DS3270	DUALCASE	DUMMY
DYNAMIC	EB	EMERGENCY	EMPTY
EMPTYREQ	ENABLED	ENDAFFINITY	ESDS
EVENT	EVENTUAL	EXCEPT	EXCEPTRESP
EXCI	EXCTL	EXECENQ	EXECENQADDR
EXITTRACE	EXTENDEDDS	EXTRA	EXTSECURITY
FAILED	FAILEDBKOUT	FAILINGBKOUT	FCLOSE
FILE	FINALQUIESCE	FINPUT	FIRSTINIT

FIRSTQUIESCE	FIXED	FLUSH	FMH
FMHPARM	FOPEN	FORCE	FORCECANCEL
FORCECLOSE	FORCECLOSING	FORCEPURGE	FORCEUOW
FORMATEDF	FORMATTED	FORMFEED	FOUTPUT
FREE	FREEING	FULL	FULLAPI
FWDRECOVABLE	GENERIC	GMT	GOINGOUT
GTFSTART	GTFSTOP	HARDCOPY	HEURBACKOUT
HEURCOMMIT	HEX	HFORM	HILIGHT
HOLD	IGNORE	IGNORERR	IMMCLOSE
IMMCLOSING	IMMEDIATE	IMMQUIESCED	INACTIVE
INBOUND	INDEXRECFULL	INDIRECT	INDOUBT
INFLIGHT	INITCOMPLETE	INOUT	INPUT
INSERVICE	INSTALLED	INSTALLFAIL	INTACTLU
INTRA	INTSTART	INTSTOP	INVALID
IOERROR	IRC	ISCMMCONV	ISOLATE
KATAKANA	KEYED	KSDS	LE370
LEAVE	LIC	LIGHTPEN	LOCAL
LOG	LOGICAL	LOGTERM	LOSE
LPA	LU61	LUCMODGRP	LUCSESS
LUP	LUSTAT	LUTYPE4	LUTYPE6
LUW	MAGTAPE	MAIN	MAP
MAPSET	MCHCTL	MDT	MOD
MODE24	MODE31	MODEANY	MODEL
MORE	MSRCONTROL	MVS	NEGATIVE
NEW	NEWCOPY	NEWSESSION	NOALARM
NOALTPRTCOPY	NOAPLKYBD	NOAPLTEXT	NOATI
NOAUDALARM	NOAUTOARCH	NOBACKOUT	NOBACKTRANS
NOCEDF	NOCLEAR	NOCMDPROT	NOCOLOR
NOCONV	NOCONVERSE	NOCOPY	NOCREATE
NOCTL	NODAE	NODISCREQ	NODUALCASE
NOEMPTYREQ	NOEVENT	NOEXCEPT	NOEXCTL
NOEXITTRACE	NOEXTENDEDDS	NOFMH	NOFMHPARM
NOFORMATEDF	NOFORMFEED	NOHFORM	NOHILIGHT
NOHOLD	NOISOLATE	NOKATAKANA	NOLIGHTPEN
NOLOG	NOLOSTLOCKS	NOMDT	NOMSGJRNL
NOMSRCONTROL	NONAUTOCONN	NONCICS	NONE
NOOBFORMAT	NOOBOPERID	NOOUTLINE	NOPARTITIONS
NOPERF	NOPRESETSEC	NOPRINTADAPT	NOPROGSYMBOL
NOPRTCOPY	NOQUERY	NORECOVDATA	NOREENTPROT
NORELREQ	NORETAINED	NORMALBKOUT	NORMALRESP
NOSECURITY	NOSHUTDOWN	NOSOSI	NOSPI
NOSTSN	NOSWITCH	NOSYNCPOINT	NOSYSDUMP
NOSYSLOG	NOTADDABLE	NOTALTERABLE	NOTAPPLIC
NOTASKSTART	NOTBROWSABLE	NOTBUSY	NOTCDEB
NOTCONNECTED	NOTCTRLABLE	NOTDEFINED	NOTDELETABLE
NOTEMPTY	NOTERMINAL	NOTEXTKYBD	NOTEXTPRINT
NOTFWDRCVBLE	NOTINBOUND	NOTINIT	NOTINSTALLED
NOTKEYED	NOTLPA	NOTPENDING	NOTPURGEABLE
NOTRANDUMP	NOTREADABLE	NOTREADY	NOTRECOVABLE
NOTREQUIRED	NOTRLS	NOTSOS	NOTSUPPORTED
NOTTABLE	NOTTI	NOTUPDATABLE	NOUCTRAN
NOVALIDATION	NOVFORM	NOWAIT	NOWRITE
NOZCPTRACE	OBFORMAT	OBOPERID	OBTAINING
OFF	OK	OLD	OLDCOPY
OLDSESSION	ON	OPEN	OPENERROR

OPENING	OPENINPUT	OPENOUTPUT	OUTLINE
OUTPUT	OUTSERVICE	OWNER	PAGEABLE
PARTITIONS	PARTITIONSET	PATH	PENDBEGIN
PENDDATA	PENDFREE	PENDING	PENDPASS
PENDRECEIVE	PENDRELEASE	PENDSTART	PENDSTSN
PENDUNSOL	PERF	PHASEIN	PHYSICAL
PL1	PLI	POSITIVE	POST
PRESETSEC	PRIMARY	PRINTADAPT	PRIVATE
PROFILE	PROGRAM	PROGSYMBOL	PROTECTED
PRTCOPY	PURGE	PURGEABLE	QUEUE
QUIESCED	QUIESCING	READABLE	READBACK
READONLY	READY	RECEIVE	RECOVDATA
RECOVERABLE	RECOVERED	RECOVERLOCKS	REENTPROT
REGERROR	REGISTERED	REJECT	RELATED
RELEASE	RELEASED	RELEASING	RELREQ
REMLOSTLOCKS	REMOTE	REMOVE	REMSESSION
REPEATABLE	REQUIRED	REREAD	RESET
RESETLOCKS	RESSECEXT	RESSECNO	RESSECYES
RESSYS	RESYNC	RETAINED	RETRY
REVERTED	RLS	RLSACTIVE	RLSGONE
RLSINACTIVE	RLSSERVER	RMI	ROLLBACK
ROUTE	RPG	RRDS	RTR
RU	RUNNING	SCS	SDLC
SECONDINIT	SEND	SEQDISK	SESSION
SESSIONFAIL	SESSIONLOST	SETFAIL	SFS
SHARE	SHARED	SHUNTED	SHUTDISABLED
SHUTDOWN	SHUTENABLED	SIGNEDOFF	SIGNEDON
SINGLEOFF	SINGLEON	SKIP	SMF
SNA	SOS	SOSABOVE	SOSBELOW
SOSI	SPECIFIC	SPECTRACE	SPI
SPRSTRACE	STANDBY	STANTRACE	START
STARTED	STARTING	STARTUP	STATIC
STOPPED	STSN	STSNSET	STSNTEST
SUBORDINATE	SUBSPACE	SURROGATE	SUSPENDED
SWITCH	SWITCHALL	SWITCHING	SWITCHNEXT
SYNCFREE	SYNCPOINT	SYNCRECEIVE	SYNCSEND
SYS370	SYS7BSCA	SYSDUMP	SYSLOG
SYSTEM3	SYSTEM	SYSTEM7	SYSTEMOFF
SYSTEMON	T1050	T1053	T2260L
T2260R	T2265	T2740	T2741BCD
T2741COR	T2770	T2780	T2980
T3275R	T3277L	T3277R	T3278M2
T3278M3	T3278M4	T3278M5	T3279M2
T3279M3	T3279M4	T3279M5	T3284L
T3284R	T3286L	T3286R	T3600BI
T3601	T3614	T3650ATT	T3650PIPE
T3650USER	T3653HOST	T3735	T3740
T3780	T3790	T3790SCSP	T3790UP
T7770	TAKEOVER	TAPE1	TAPE2
TASK	TASKSTART	TCAM	TCAMSNA
TCEXITALL	TCEXITALLOFF	TCEXITNONE	TCEXITSYSTEM
TCLASS	TCONSOLE	TDQ	TELETYPE
TERM	TERMINAL	TEXTKYBD	TEXTPRINT
THIRDINIT	TIME	TIMEOUT	TPS55M2
TPS55M3	TPS55M4	TPS55M5	TRANDUMP

TRANIDONLY	TSQ	TTCAM	TTI
TWX3335	UCTRAN	UNAVAILABLE	UNBLOCKED
UNCOMMITTED	UNCONNECTED	UNDEFINED	UNDETERMINED
UNENABLED	UNENABLING	UNKNOWN	UNPROTECTED
UNQUIESCED	UNREGISTERED	UNSOLDATA	UOW
UPDATABLE	USER	USERDATAKEY	USEREXECKEY
USEROFF	USERON	USERTABLE	VALID
VALIDATION	VARIABLE	VFORM	VIDEOTERM
VRRDS	VSAM	VTAM®	WAIT
WAITCOMMIT	WAITER	WAITFORGET	WAITING
WAITRMI	WARMSTART	WIN	XCF
XM	XNOTDONE	XOK	ZCPTRACE

Chapter 64. main function

You are recommended to include this code in your application. It initializes the CICS Foundation Classes correctly, provides default exception handling, and releases allocated memory after it is finished. You may substitute your own variation of this **main** function, provided you know what you are doing, but this should rarely be necessary.

Source file: ICCMAIN

The stub has three functions:
1. It initializes the Foundation Classes environment. You can customize the way it does this by using `#define`s that control:
   - memory management (see page 61)
   - Family Subset enforcement (see page 70)
   - EDF enablement (see page 50)
2. It provides a default definition of a class **IccUserControl**, derived from **IccControl**, that includes a default constructor and **run** method.
3. It invokes the **run** method of the user's control object using a try-catch construct.

The functional part of the **main** code is shown below.

```
int main()                                                  1

{
    Icc::initializeEnvironment(ICC_CLASS_MEMORY_MGMT,           2

                               ICC_FAMILY_SUBSET,
                               ICC_EDF_BOOL);
    try                                                         3

    {
        ICC_USER_CONTROL control;                               4

        control.run();                                          5

    }
    catch(IccException& exc)                                    6

    {
        Icc::catchException(exc);                               7

    }
    catch(...)                                                  8

    {
        Icc::unknownException();                                9

    }
    Icc::returnToCICS();                                        10

}
```

1 This is the main C++ entry point.

2 This call initializes the environment and is essential. The three parameters have previously been defined to the defaults for the platform.

3 Run the users application code, using **try** and **catch**, in case the application code does not catch exceptions.
4 Create control object.
5 Invoke **run** method of control object (defined as pure virtual in **IccControl**.
6 Catch any **IccException** objects not caught by the application.
7 Call this function to abend task.
8 Catch any other exceptions not caught by application.
9 Call this function to abend task.
10 Return control to CICS.

Part 4. Appendixes

Appendix A. Mapping EXEC CICS calls to Foundation Class methods

The following table shows the correspondence between CICS calls made using the EXEC CICS API and the equivalent calls from the Foundation Classes.

EXEC CICS	Class	Method
ABEND	IccTask	abend
ADDRESS COMMAREA	IccControl	commArea
ADDRESS CWA	IccSystem	workArea
ADDRESS EIB	No direct access to EIB: please use appropriate method on appropriate class.	
ADDRESS TCTUA	IccTerminal	workArea
ADDRESS TWA	IccTask	workArea
ALLOCATE	IccSession	allocate
ASKTIME	IccClock	update
ASSIGN ABCODE	IccAbendData	abendCode
ASSIGN ABDUMP	IccAbendData	isDumpAvaliable
ASSIGN ABPROGRAM	IccAbendData	programName
ASSIGN ALTSCRNHT	IccTerminalData	alternateHeight
ASSIGN ALTSCRNWD	IccTerminalData	alternateWidth
ASSIGN APLKYBD	IccTerminalData	isAPLKeyboard
ASSIGN APLTEXT	IccTerminalData	isAPLText
ASSIGN ASRAINTRPT	IccAbendData	ASRAInterrupt
ASSIGN ASRAKEY	IccAbendData	ASRAKeyType
ASSIGN ASRAPSW	IccAbendData	ASRAPSW
ASSIGN ASRAREGS	IccAbendData	ASRARegisters
ASSIGN ASRASPC	IccAbendData	ASRASpaceType
ASSIGN ASRASTG	IccAbendData	ASRAStorageType
ASSIGN APPLID	IccSystem	applName
ASSIGN BTRANS	IccTerminalData	isBTrans
ASSIGN CMDSEC	IccTask	isCommandSecurityOn
ASSIGN COLOR	IccTerminalData	isColor
ASSIGN CWALENG	IccSystem	workArea
ASSIGN DEFSCRNHT	IccTerminalData	defaultHeight
ASSIGN DEFSCRNWD	IccTerminalData	defaultWidth
ASSIGN EWASUPP	IccTerminalData	isEWA
ASSIGN EXTDS	IccTerminalData	isExtended3270
ASSIGN FACILITY	IccTerminal	name
ASSIGN FCI	IccTask	facilityType
ASSIGN GCHARS	IccTerminalData	graphicCharSetId
ASSIGN GCODES	IccTerminalData	graphicCharCodeSet

EXEC CICS	Class	Method
ASSIGN GMMI	IccTerminalData	isGoodMorning
ASSIGN HILIGHT	IccTerminalData	isHighlight
ASSIGN INITPARM	IccControl	initData
ASSIGN INITPARMLEN	IccControl	initData
ASSIGN INVOKINGPROG	IccControl	callingProgramId
ASSIGN KATAKANA	IccTerminalData	isKatakana
ASSIGN NETNAME	IccTerminal	netName
ASSIGN OUTLINE	IccTerminalData	isFieldOutline
ASSIGN ORGABCODE	IccAbendData	originalAbendCode
ASSIGN PRINSYSID	IccTask	principalSysId
ASSIGN PROGRAM	IccControl	programId
ASSIGN PS	IccTerminalData	isPS
ASSIGN QNAME	IccTask	triggerDataQueueId
ASSIGN RESSEC	IccTask	isResourceSecurityOn
ASSIGN RESTART	IccTask	isRestarted
ASSIGN SCRNHT	IccTerminal	height
ASSIGN SCRNWD	IccTerminal	width
ASSIGN SOSI	IccTerminalData	isSOSI
ASSIGN STARTCODE	IccTask	startType, isCommitSupported, isStartDataAvailable
ASSIGN SYSID	IccSystem	sysId
ASSIGN TASKPRIORITY	IccTask	priority
ASSIGN TCTUALENG	IccTerminal	workArea
ASSIGN TEXTKYBD	IccTerminalData	isTextKeyboard
ASSIGN TEXTPRINT	IccTerminalData	isTextPrint
ASSIGN TWALENG	IccTask	workArea
ASSIGN USERID	IccTask	userId
ASSIGN VALIDATION	IccTerminalData	isValidation
CANCEL	IccClock	cancelAlarm
CANCEL	IccStartRequestQ	cancel
CHANGE PASSWORD	IccUser	changePassword
CHANGE TASK	IccTask	setPriority
CONNECT PROCESS	IccSession	connectProcess
CONVERSE	IccSession	converse
DELAY	IccTask	delay
DELETE	IccFile	deleteRecord
DELETE	IccFile	deleteLockedRecord
DELETEQ TD	IccDataQueue	empty
DELETEQ TS	IccTempStore	empty
DEQ	IccSemaphore	unlock

EXEC CICS	Class	Method
DUMP TRANSACTION	IccTask	dump
DUMP TRANSACTION	IccTask	setDumpOpts
ENDBR	IccFileIterator	IccFileIterator (destructor)
ENQ	IccSemaphore	lock
ENQ	IccSemaphore	tryLock
ENTER TRACENUM	IccTask	enterTrace
EXTRACT ATTRIBUTES	IccSession	state, stateText
EXTRACT PROCESS	IccSession	extractProcess
FORMATTIME YYDDD, YYMMDD, etc	IccClock	date
FORMATTIME DATE	IccClock	date
FORMATTIME DATEFORM	IccSystem	dateFormat
FORMATTIME DAYCOUNT	IccClock	daysSince1900
FORMATTIME DAYOFWEEK	IccClock	dayOfWeek
FORMATTIME DAYOFMONTH	IccClock	dayOfMonth
FORMATTIME MONTHOFYEAR	IccClock	monthOfYear
FORMATTIME TIME	IccClock	time
FORMATTIME YEAR	IccClock	year
FREE	IccSession	free
FREEMAIN	IccTask	freeStorage
GETMAIN	IccTask	getStorage
HANDLE ABEND	IccControl	setAbendHandler, cancelAbendHandler, resetAbendHandler
INQUIRE FILE ACCESSMETHOD	IccFile	accessMethod
INQUIRE FILE ADD	IccFile	isAddable
INQUIRE FILE BROWSE	IccFile	isBrowsable
INQUIRE FILE DELETE	IccFileControl	isDeletable
INQUIRE FILE EMPTYSTATUS	IccFile	isEmptyOn
INQUIRE FILE ENABLESTATUS	IccFile	enableStatus
INQUIRE FILE KEYPOSITION	IccFile	keyPosition
INQUIRE FILE OPENSTATUS	IccFile	openStatus
INQUIRE FILE READ	IccFile	isReadable
INQUIRE FILE RECORDFORMAT	IccFile	recordFormat
INQUIRE FILE RECORDSIZE	IccFile	recordLength

EXEC CICS	Class	Method
INQUIRE FILE RECOVSTATUS	IccFile	isRecoverable
INQUIRE FILE TYPE	IccFile	type
INQUIRE FILE UPDATE	IccFile	isUpdatable
ISSUE ABEND	IccSession	issueAbend
ISSUE CONFIRMATION	IccSession	issueConfirmation
ISSUE ERROR	IccSession	issueError
ISSUE PREPARE	IccSession	issuePrepare
ISSUE SIGNAL	IccSession	issueSignal
LINK	IccProgram	link
LINK INPUTMSG INPUTMSGLEN	IccProgram	setInputMessage
LOAD	IccProgram	load
POST	IccClock	setAlarm
READ	IccFile	readRecord
READNEXT	IccFileIterator	readNextRecord
READPREV	IccFileIterator	readPreviousRecord
READQ TD	IccDataQueue	readItem
READQ TS	IccTempStore	readItem
RECEIVE (APPC)	IccSession	receive
RECEIVE (3270)	IccTerminal	receive, receive3270Data
RELEASE	IccProgram	unload
RESETBR	IccFileIterator	reset
RETRIEVE	IccStartRequestQ	retrieveData [1]
Note: The **retrieveData** method gets the start information from CICS and stores it in the IccStartRequestQ object: the information can then be accessed using **data, queueName, returnTermId and returnTransId** methods.		
RETRIEVE INTO, LENGTH	IccStartRequestQ	data
RETRIEVE QUEUE	IccStartRequestQ	queueName
RETRIEVE RTRANSID	IccStartRequestQ	returnTransId
RETRIEVE RTERMID	IccStartRequestQ	returnTermId
RETURN	IccControl	main [2]
Note: Returning (using C++ reserved word **return**) from method **run** in class **IccControl** results in an EXEC CICS RETURN.		
RETURN TRANSID	IccTerminal	setNextTransId [3]
RETURN IMMEDIATE	IccTerminal	setNextTransId [3]
RETURN COMMAREA LENGTH	IccTerminal	setNextCommArea [3]
RETURN INPUTMSG, INPUTMSGLEN	IccTerminal	setNextInputMessage [3]
Note: Issue this call before returning from **IccControl::run**.		
REWRITE	IccFile	rewriteRecord
SEND (APPC)	IccSession	send, sendInvite, sendLast

EXEC CICS	Class	Method
SEND (3270)	IccTerminal	send, sendLine
SEND CONTROL CURSOR	IccTerminal	setCursor setLine, setNewLine
SEND CONTROL ERASE	IccTerminal	erase
SEND CONTROL FREEKB	IccTerminal	freeKeyboard
SET FILE ADD|BROWSE|DELETE|...	IccFile	setAccess
SET FILE EMPTYSTATUS	IccFile	setEmptyOnOpen
SET FILE OPEN STATUS|ENABLESTATUS	IccFile	setStatus
SIGNOFF	IccTerminal	signoff
SIGNON	IccTerminal	signon
START TRANSID AT/AFTER	IccStartRequestQ	start [4]
START TRANSID FROM LENGTH	IccStartRequestQ	setData, registerDataBuffer [4]
START TRANSID NOCHECK	IccStartRequestQ	setStartOpts [4]
START TRANSID PROTECT	IccStartRequestQ	setStartOpts [4]
START TRANSID QUEUE	IccStartRequestQ	setQueueName [4]
START TRANSID REQID	IccStartRequestQ	start [4]
START TRANSID TERMID	IccStartRequestQ	start [4]
START TRANSID USERID	IccStartRequestQ	start [4]
START TRANSID RTERMID	IccStartRequestQ	setReturnTermId [4]
START TRANSID RTRANSID	IccStartRequestQ	setReturnTransId [4]
Note: Use methods **setData, setQueueName, setReturnTermId, setReturnTransId, setStartOpts** to set the state of the **IccStartRequestQ** object before issuing start requests with the **start** method.		
STARTBR	IccFileIterator	IccFileIterator (constructor)
SUSPEND	IccTask	suspend
SYNCPOINT	IccTask	commitUOW
SYNCPOINT ROLLBACK	IccTask	rollBackUOW
UNLOCK	IccFile	unlockRecord
VERIFY PASSWORD	IccUser	verifyPassword
WAIT CONVID	IccSession	flush
WAIT EVENT	IccTask	waitOnAlarm
WAIT EXTERNAL	IccTask	waitExternal
WAIT JOURNALNUM	IccJournal	wait
WRITE	IccFile	writeRecord
WRITE OPERATOR	IccConsole	write, writeAndGetReply
WRITEQ TD	IccDataQueue	writeItem
WRITEQ TS	IccTempStore	writeItem, rewriteItem

Appendix B. Mapping Foundation Class methods to EXEC CICS calls

The following table shows the correspondence between CICS calls made using the Foundation Classes and the equivalent EXEC CICS API calls.

IccAbendData Class	
Method	EXEC CICS
abendCode	ASSIGN ABCODE
ASRAInterrupt	ASSIGN ASRAINTRPT
ASRAKeyType	ASSIGN ASRAKEY
ASRAPSW	ASSIGN ASRAPSW
ASRARegisters	ASSIGN ASRAREGS
ASRASpaceType	ASSIGN ASRASPC
ASRAStorageType	ASSIGN ASRASTG
isDumpAvailable	ASSIGN ABDUMP
originalAbendCode	ASSIGN ORGABCODE
programName	ASSIGN ABPROGRAM
IccAbsTime Class	
Method	EXEC CICS
date	FORMATTIME YYDDD/YYMMDD/etc.
dayOfMonth	FORMATTIME DAYOFMONTH
dayOfWeek	FORMATTIME DAYOFWEEK
daysSince1900	FORMATTIME DAYCOUNT
monthOfYear	FORMATTIME MONTHOFYEAR
time	FORMATTIME TIME
year	FORMATTIME YEAR
IccClock Class	
Method	EXEC CICS
cancelAlarm	CANCEL
date	FORMATTIME YYDDD/YYMMDD/etc.
dayOfMonth	FORMATTIME DAYOFMONTH
dayOfWeek	FORMATTIME DAYOFWEEK
daysSince1900	FORMATTIME DAYCOUNT
monthOfYear	FORMATTIME MONTHOFYEAR
setAlarm	POST
time	FORMATTIME TIME
update	ASKTIME
year	FORMATTIME YEAR
IccConsole Class	
Method	EXEC CICS
write	WRITE OPERATOR

writeAndGetReply	WRITE OPERATOR
IccControl Class	
Method	EXEC CICS
callingProgramId	ASSIGN INVOKINGPROG
cancelAbendHandler	HANDLE ABEND CANCEL
commArea	ADDRESS COMMAREA
initData	ASSIGN INITPARM & INITPARMLEN
programId	ASSIGN PROGRAM
resetAbendHandler	HANDLE ABEND RESET
setAbendHandler	HANDLE ABEND PROGRAM
IccDataQueue Class	
Method	EXEC CICS
empty	DELETEQ TD
readItem	READQ TD
writeItem	WRITEQ TD
IccFile Class	
Method	EXEC CICS
access	INQUIRE FILE ADD|BROWSE|DELETE|READ|UPDATE
accessMethod	INQUIRE FILE ACCESSMETHOD
deleteRecord	DELETE FILE RIDFLD
deleteLockedRecord	DELETE FILE
enableStatus	INQUIRE FILE ENABLESTATUS
isAddable	INQUIRE FILE ADD
isBrowsable	INQUIRE FILE BROWSE
isDeletable	INQUIRE FILE DELETE
isEmptyOnOpen	INQUIRE FILE EMPTYSTATUS
isReadable	INQUIRE FILE READ
isRecoverable	INQUIRE FILE RECOVSTATUS
isUpdatable	INQUIRE FILE UPDATE
keyPosition	INQUIRE FILE KEYPOSITION
openStatus	INQUIRE FILE OPENSTATUS
readRecord	READ FILE
recordFormat	INQUIRE FILE RECORDFORMAT
recordLength	INQUIRE FILE RECORDSIZE
rewriteRecord	REWRITE FILE
setAccess	SET FILE ADD BROWSE DELETE etc.
setEmptyOnOpen	SET FILE EMPTYSTATUS
setStatus	SET FILE OPENSTATUS ENABLESTATUS
type	INQUIRE FILE TYPE
unlockRecord	UNLOCK FILE
writeRecord	WRITE FILE
IccFileIterator Class	

Method	EXEC CICS
IccFileIterator (constructor)	STARTBR FILE
~IccFileIterator (destructor)	ENDBR FILE
readNextRecord	READNEXT FILE
readPreviousRecord	READPREV FILE
reset	RESETBR FILE
IccJournal Class	
Method	EXEC CICS
wait	WAIT JOURNALNUM
writeRecord	WRITE JOURNALNUM
IccProgram Class	
Method	EXEC CICS
link	LINK PROGRAM
load	LOAD PROGRAM
unload	RELEASE PROGRAM
IccResource Class	
Method	EXEC CICS
condition	(RESP & RESP2)
setRouteOption	(SYSID)
IccSemaphore Class	
Method	EXEC CICS
lock	ENQ RESOURCE
tryLock	ENQ RESOURCE NOSUSPEND
unlock	DEQ RESOURCE
IccSession Class	
Method	EXEC CICS
allocate	ALLOCATE
connectProcess	CONNECT PROCESS CONVID
converse	CONVERSE CONVID
extractProcess	EXTRACT PROCESS CONVID
flush	WAIT CONVID
free	FREE CONVID
issueAbend	ISSUE ABEND CONVID
issueConfirmation	ISSUE CONFIRMATION CONVID
issueError	ISSUE ERROR CONVID
issuePrepare	ISSUE PREPARE CONVID
issueSignal	ISSUE SIGNAL CONVID
receive	RECEIVE CONVID
send	SEND CONVID
sendInvite	SEND CONVID INVITE
sendLast	SEND CONVID LAST
state	EXTRACT ATTRIBUTES

Foundation Class methods to EXEC CICS

IccStartRequestQ Class	
Method	EXEC CICS
cancel	CANCEL
retrieveData	RETRIEVE
start	START TRANSID
IccSystem Class	
Method	EXEC CICS
applName	ASSIGN APPLID
beginBrowse	INQUIRE (FILE, TDQUEUE, etc) START
dateFormat	FORMATTIME DATEFORM
endBrowse	INQUIRE (FILE, TDQUEUE, etc) END
freeStorage	FREEMAIN
getFile	INQUIRE FILE
getNextFile	INQUIRE FILE NEXT
getStorage	GETMAIN SHARED
operatingSystem	INQUIRE SYSTEM OPSYS
operatingSystemLevel	INQUIRE SYSTEM OPREL
release	INQUIRE SYSTEM RELEASE
releaseText	INQUIRE SYSTEM RELEASE
sysId	ASSIGN SYSID
workArea	ADDRESS CWA
IccTask Class	
Method	EXEC CICS
abend	ABEND
commitUOW	SYNCPOINT
delay	DELAY
dump	DUMP TRANSACTION
enterTrace	ENTER TRACENUM
facilityType	ASSIGN STARTCODE, TERMCODE, PRINSYSID, FCI
freeStorage	FREEMAIN
isCommandSecurityOn	ASSIGN CMDSEC
isCommitSupported	ASSIGN STARTCODE
isResourceSecurityOn	ASSIGN RESSEC
isRestarted	ASSIGN RESTART
isStartDataAvailable	ASSIGN STARTCODE
principalSysId	ASSIGN PRINSYSID
priority	ASSIGN TASKPRIORITY
rollBackUOW	SYNCPOINT ROLLBACK
setPrioity	CHANGE TASK PRIORITY
startType	ASSIGN STARTCODE
suspend	SUSPEND
triggerDataQueueId	ASSIGN QNAME

userId	ASSIGN USERID
waitExternal	WAIT EXTERNAL / WAITCICS
waitOnAlarm	WAIT EVENT
workArea	ADDRESS TWA
IccTempStore Class	
Method	EXEC CICS
empty	DELETEQ TS
readItem	READQ TS ITEM
readNextItem	READQ TS NEXT
rewriteItem	WRITEQ TS ITEM REWRITE
writeItem	WRITEQ TS ITEM
IccTerminal Class	
Method	EXEC CICS
erase	SEND CONTROL ERASE
freeKeyboard	SEND CONTROL FREEKB
height	ASSIGN SCRNHT
netName	ASSIGN NETNAME
receive	RECEIVE
receive3270Data	RECEIVE BUFFER
send	SEND
sendLine	SEND
setCursor	SEND CONTROL CURSOR
setLine	SEND CONTROL CURSOR
setNewLine	SEND CONTROL CURSOR
signoff	SIGNOFF
signon	SIGNON
waitForAID	RECEIVE
width	ASSIGN SCRNWD
workArea	ADDRESS TCTUA
IccTerminalData Class	
Method	EXEC CICS
alternateHeight	ASSIGN ALTSCRNHT
alternateWidth	ASSIGN ALTSCRNWD
defaultHeight	ASSIGN DEFSCRNHT
defaultWidth	ASSIGN DEFSCRNWD
graphicCharSetId	ASSIGN GCHARS
graphicCharCodeSet	ASSIGN GCODES
isAPLKeyboard	ASSIGN APLKYBD
isAPLText	ASSIGN APLTEXT
isBTrans	ASSIGN BTRANS
isColor	ASSIGN COLOR
isEWA	ASSIGN ESASUPP

isExtended3270	ASSIGN EXTDS
isGoodMorning	ASSIGN GMMI
isHighlight	ASSIGN HILIGHT
isKatakana	ASSIGN KATAKANA
isMSRControl	ASSIGN MSRCONTROL
isFieldOutline	ASSIGN OUTLINE
isPS	ASSIGN PS
isSOSI	ASSIGN SOSI
isTextKeyboard	ASSIGN TEXTKYBD
isTextPrint	ASSIGN TEXTPRINT
isValidation	ASSIGN VALIDATION
IccUser Class	
Method	EXEC CICS
changePassword	CHANGE PASSWORD
verifyPassword	VERIFY PASSWORD

Appendix C. Output from sample programs

This section shows the typical screen output from the supplied sample programs (see Sample source code˝ on page 6).

ICC$BUF (IBUF)

```
This is program 'icc$buf'...
IccBuf buf1                          dal= 0 dl= 0 E+I []
IccBuf buf2(50)                      dal=50 dl= 0 E+I []
IccBuf buf3(30,fixed)                dal=30 dl= 0 F+I []
IccBuf buf4(sizeof(AStruct),&aStruc) dal=24 dl=24 F+E [!Some text for aStruc]
IccBuf buf5(A String Literal)      dal=19 dl=19 E+I [Some data somewhere]
IccBuf buf6(buf5)                    dal=19 dl=19 E+I [Some data somewhere]
buf1 = Some XXX data for buf1      dal=22 dl=22 E+I [Some XXX data for buf1]
buf2.assign(strlen(data),data)       dal=50 dl=19 E+I [Some data somewhere]
buf1.cut(4,5)                        dal=22 dl=18 E+I [Some data for buf1]
buf5.insert(5,more,5)                dal=24 dl=24 E+I [Some more data somewhere]
buf5.replace(4,xtra,5)               dal=24 dl=24 E+I [Some xtra data somewhere]
buf2 << .ext                       dal=50 dl=23 E+I [Some data somewhere.ext]
buf3 = buf4                          dal=30 dl=24 F+I [!Some text for aStruc]
(buf3 == buf4) returns true (OK).
buf3 = garbage                     dal=30 dl= 7 F+I [garbage]
(buf3 != buf4) returns true (OK).
Program 'icc$buf' complete: Hit PF12 to End
```

ICC$CLK (ICLK)

```
This is program 'icc$clk' ...
date() = [220296  ]
date(DDMMYY) = [220296  ]
date(DDMMYY,':') = [22:02:96]
date(MMDDYY) = [022296  ]
date(YYDDD) = [96053   ]
daysSince1900() = 35116
dayOfWeek() = 4                        Today is NOT Friday
dayOfMonth() = 22
monthOfYear() = 2
time() = [143832  ]
time('-') = [14-38-32]
year() = [1996]
Program 'icc$clk' complete: Hit PF12 to End
```

ICC$DAT (IDAT)

```
This is program 'icc$dat'...
Writing records to 'ICCQ'...
 - writing record #1: 'Hello World - item 1' <NORMAL>
 - writing record #2: 'Hello World - item 2' <NORMAL>
 - writing record #3: 'Hello World - item 3' <NORMAL>
Reading records back in...
 - reading record #1: 'Hello World - item 1' <NORMAL>
 - reading record #2: 'Hello World - item 2' <NORMAL>
 - reading record #3: 'Hello World - item 3' <NORMAL>
Program 'icc$dat' complete: Hit PF12 to End
```

ICC$EXC1 (IEX1)

```
This is program 'icc$exc1' ...
Number passed = 1
Number passed = 7
Number passed = 11
>>Out of Range - throwing exception
Exception caught: !!Number is out of range!!
Program 'icc$exc1' complete: Hit PF12 to End
```

ICC$EXC2 (IEX2)

```
This is program 'icc$exc2'...
Creating IccTermId id1...
Creating IccTermId id2...
IccException: 112 IccTermId::IccTermId type=invalidArgument (IccMessage: 030 Ic
cTermId::IccTermId <Invalid string length passed to 'IccTermId' constructor.
Spec ified: 5, Maximum allowed: 4>)
Program 'icc$exc2' complete: Hit PF12 to End
```

ICC$EXC3 (IEX3)

```
This is program 'icc$exc3'...
About to read Temporary Storage 'UNKNOWN!'...
IccException: 094 IccTempStore::readNextItem type=CICSCondition (IccMessage: 008
 IccTempStore::readNextItem <CICS returned the 'QIDERR' condition.>)
Program 'icc$exc3' complete: Hit PF12 to End
```

ICC$FIL (IFIL)

```
This is program 'icc$fil'...
Deleting records in file 'ICCKFILE...
5 records were deleted.
Writing records to file 'ICCKFILE'...
- writing record number 1.    <NORMAL>
- writing record number 2.    <NORMAL>
- writing record number 3.    <NORMAL>
- writing record number 4.    <NORMAL>
- writing record number 5.    <NORMAL>
Browsing records...
 - record read: [BACH, J S      003  00-1234   BACH      ]
 - record read: [CHOPIN, F      004  00-3355   CHOPIN    ]
 - record read: [HANDEL, G F    005  00-4466   HANDEL    ]
 - record read: [BEETHOVEN, L   007  00-2244   BEET      ]
 - record read: [MOZART, W A    008  00-5577   WOLFGANG  ]
 - record read: [MOZART, W A    008  00-5577   WOLFGANG  ]
 - record read: [BEETHOVEN, L   007  00-2244   BEET      ]
 - record read: [HANDEL, G F    005  00-4466   HANDEL    ]
 - record read: [CHOPIN, F      004  00-3355   CHOPIN    ]
 - record read: [BACH, J S      003  00-1234   BACH      ]
Updating record 1...
readRecord(update)<NORMAL>    rewriteRecord()<NORMAL>
 - record read: [MOZART, W A    008  00-5678   WOLFGANG  ]
Program 'icc$fil' complete: Hit PF12 to End
```

ICC$HEL (IHEL)

```
                                 Hello World
```

ICC$JRN (IJRN)

```
This is program 'icc$jrn'...
Writing  3 records to  journal number 77...
 - writing record 1: [Hello World - item 1]      <NORMAL>
 - writing record 2: [Hello World - item 2]      <NORMAL>
 - writing record 3: [Hello World - item 3]      <NORMAL>
Program 'icc$jrn' complete: Hit PF12 to End
```

ICC$PRG1 (IPR1)

First Screen

```
This is program 'icc$prg1'...
Loaded program: ICC$PRG2  <NORMAL> Length=0 Address=ff000000
Unloading program: ICC$PRG2        <NORMAL>
 - Hit ENTER to continue...
```

Second Screen

```
About to link to program 'ICC$PRG2 '
 - commArea before link is [DATA SET BY ICC$PRG1]
 - Hit ENTER to continue...
   This is program 'icc$prg2'...
   commArea received from caller =[DATA SET BY ICC$PRG1]
   Changed commArea to [DATA RETURNED BY ICC$PRG2]
   - Hit ENTER to return to caller...
 - link call returned <NORMAL>
 - commArea after link is [DATA RETURNED BY ICC$PRG2]
About to link to program 'ICC$PRG3 ' on system 'ICC2'
 - commArea before link is [DATA SET BY ICC$PRG1]
 - Hit ENTER to continue...
 - link call returned <NORMAL>
 - commArea after link is [DATA RETURNED BY ICC$PRG3]
Program 'icc$prg1' complete: Hit PF12 to End
```

ICC$RES1 (IRS1)

```
This is program 'icc$res1'...
Writing items to CustomDataQueue 'ICCQ' ...
 - writing item #1: 'Hello World - item 1'  <NORMAL>
 - writing item #2: 'Hello World - item 2'  <NORMAL>
 - writing item #3: 'Hello World - item 3'  <NORMAL>
Reading items from CustomDataQueue 'ICCQ' ...
 - item = 'Hello World - item 1'
 - item = 'Hello World - item 2'
 - item = 'Hello World - item 3'
Reading loop complete.
> In handleEvent().
Summary=IccEvent: CustomDataQueue::readItem condition=23 (QZ ERO) minor=0
Program 'icc$res1' complete: Hit PF12 to End
```

ICC$RES2 (IRS2)

```
This is program 'icc$res2'...
invoking clear() method for IccDataQueue object
invoking clear() method for IccTempStore object
put() item #1 in IccDataQueue object
put() item #2 in IccDataQueue object
put() item #3 in IccDataQueue object
put() item #1 in IccTempStore object
put() item #2 in IccTempStore object
put() item #3 in IccTempStore object
Now get items from IccDataQueue object
get() from IccDataQueue object returned 'Hello World - item 1'
get() from IccDataQueue object returned 'Hello World - item 2'
get() from IccDataQueue object returned 'Hello World - item 3'
Now get items from IccTempStore object
get() from IccTempStore object returned 'Hello World - item 1'
get() from IccTempStore object returned 'Hello World - item 2'
get() from IccTempStore object returned 'Hello World - item 3'
Program 'icc$res2' complete: Hit PF12 to End
```

ICC$SEM (ISEM)

```
This is program 'icc$sem'...
Constructing IccSemaphore object (lock by value)...
Issuing lock request...     <NORMAL>
Issuing unlock request...   <NORMAL>
Constructing Semaphore object (lock by address)...
Issuing tryLock request...  <NORMAL>
Issuing unlock request...   <NORMAL>


Program 'icc$sem' complete: Hit PF12 to End
```

ICC$SES1 (ISE1)

```
This is program 'icc$ses1'...
allocate session...                     <NORMAL>
STATE=81 ALLOCATED ERR=0 connectProcess...<NORMAL>
STATE=90 SEND ERR=0 sendInvite ...      <NORMAL>
STATE=87 PENDRECEIVE ERR=0 receive ...  <NORMAL>
STATE=85 FREE ERR=0  - data from back end=[Hi there this is from backEnd
TIME=14:49:18 on 22/02/96]
free...                                 <NORMAL>
STATE=1 NOTAPPLIC ERR=0

Program 'icc$ses1' complete: Hit PF12 to End
```

ICC$SES2 (ISE2)

This screen is typical output after running "CEBR DTPBKEND" on the back-end CICS system:

```
 CEBR  TSQ DTPBKEND       SYSID ABCD REC     1 OF    11    COL     1 OF    78
 ENTER COMMAND ===>
     ************************  TOP OF QUEUE  *****************************
00001 Transaction 'ISE2' starting.
00002 extractProcess...
00003 <NORMAL> STATE=88 RECEIVE ERR=0
00004 process=[ISE2] syncLevel=1 PIP=[Hello World]
00005 receive...
00006 <NORMAL> STATE=90 SEND ERR=0 NoData=0
00007 data from front end=[Hi there this is from frontEnd TIME=16:03:18 on 04/0
00008 sendLast ...
00009 <NORMAL>        STATE=86 PENDFREE ERR=0
00010 free...
00011 <NORMAL>        STATE=1 NOTAPPLIC ERR=0
     ***********************  BOTTOM OF QUEUE  ***************************
PF1 : HELP                PF2 : SWITCH HEX/CHAR     PF3 : TERMINATE BROWSE
PF4 : VIEW TOP            PF5 : VIEW BOTTOM         PF6 : REPEAT LAST FIND
PF7 : SCROLL BACK HALF    PF8 : SCROLL FORWARD HALF PF9 : VIEW RIGHT
PF10: SCROLL BACK FULL    PF11: SCROLL FORWARD FULL PF12: UNDEFINED
```

ICC$SRQ1 (ISR1)

```
This is program 'icc$srq1'...
Starting Tran 'ISR2' on terminal 'PE12' after 5 seconds... - <NORMAL>
request='DF!U0000'
Issuing cancel for start request='DF!U0000'...             - <NORMAL>
request='DF!U0000'
Starting Tran 'ISR2' on terminal 'PE12' after 5 seconds... - <NORMAL>
request='REQUEST1'
Program 'icc$srq1' complete.
```

ICC$SRQ2 (ISR2)

```
This is program 'icc$srq2'...
retrieveData()...                                        <NORMAL>
Start buffer contents = [This is a greeting from program 'icc$srq1'!!]
Start queue= [startqnm]
Start rtrn = [ITMP]
Start rtrm = [PE11]
Sleeping for 5 seconds...
Starting tran 'ITMP' on terminal 'PE11' on system ICC1...<NORMAL>

Program 'icc$srq2' complete: Hit PF12 to end
```

ICC$SYS (ISYS)

```
This is program 'icc$sys'...
applName=ICC$REG01 operatingSystem=A operatingSystemLevel=41
releaseText=[0210] sysidnt=ICC1
getStorage( 5678, 'Y')...                                   <NORMAL>
freeStorage( p )...                                         <NORMAL>
Checking attributes of a named file (ICCKFILE)...
>ICCKFILE< Add=true Brw=true Del=true Read=true Upd=true op=18 en=23
accessMethod=3 isRecoverable=true keyLength=3 keyPosition=16
setStatus( closed ) ...                                     <NORMAL>
setStatus( disabled ) ...                                   <NORMAL>
setAccess( notUpdatable ) ...                               <NORMAL>
>ICCKFILE< Add=true Brw=true Del=true Read=true Upd=false op=19 en=24
setAccess( updateable ) & setStatus( enabled, open ) ...
>ICCKFILE< Add=true Brw=true Del=true Read=true Upd=true op=18 en=23
Beginning browse of all file objects in CICS system...      <NORMAL>
 - >ICCEFILE< type=1                                        <NORMAL>
 - >ICCKFILE< type=6                                        <NORMAL>
 - >ICCRFILE< type=1                                        <NORMAL>
Program 'icc$sys' complete: Hit PF12 to End
```

ICC$TMP (ITMP)

```
This is program 'icc$tmp'...
Writing 3 records to IccTempStore object 'ICCSTORE'...
 - writing record #1: 'Hello World - item 1'     <NORMAL>
 - writing record #2: 'Hello World - item 2'     <NORMAL>
 - writing record #3: 'Hello World - item 3'     <NORMAL>
Reading records back in & rewriting new buffer contents...
 - record #1 = [Hello World - item 1]       - rewriteItem #1   <NORMAL>
 - record #2 = [Hello World - item 2]       - rewriteItem #2   <NORMAL>
 - record #3 = [Hello World - item 3]       - rewriteItem #3   <NORMAL>
Reading records back in one last time...
 - record #1 = [Modified Hello World - item 1]
 - record #1 = [Modified Hello World - item 2]
 - record #1 = [Modified Hello World - item 3]
Program 'icc$tmp' complete: Hit PF12 to end
```

ICC$TRM (ITRM)

```
This is program 'icc$trm'...
First part of the line...... a continuation of the line.
Start this on the next line           Send this to col 40 of current line

         Send this to row 5, column 10
                                      Send this to row 6, column 40

A Red line!
A Blue, reverse video line!

A cout style interface...
you can chain input together; use different types, eg numbers: 123 4567890 12345
6.789123
... and everything is buffered till you issue a flush.



Program 'icc$trm' complete: Hit PF12 to End
```

ICC$TSK (ITSK)

```
This is program 'icc$tsk'...
startType() = terminalInput
number() = 0598
isStartDataSupplied() = true
isCommitSupported() = true
userId() = [rabcics ]
enterTrace( 77, ICCENTRY, buffer )        <NORMAL>
suspend()...                                <NORMAL>
delay( ti ) (for 2 seconds)...              <NORMAL>
getStorage( 1234, 'X')...                   <NORMAL>
freeStorage( p )...                         <NORMAL>
commitUOW()...                              <NORMAL>
rollBackUOW()...                            <NORMAL>


Program 'icc$tsk' complete: Hit PF12 to End OR PF24 to ABEND
```

Glossary

abstract class. A class that is used as a base class for other classes and has at least one pure virtual function. It is not possible to create an instance of this class.

base class. A class from which other classes are derived.

CICS program. A program that runs in the CICS environment as part of a transaction.

class. A group of objects that share a common definition and common properties, operations and behavior.

class definition. How a class is defined in C++.

class implementation. How a class is implemented in C++.

const. In C++, the **const** attribute explicitly declares a data object as a data item that cannot be changed. Its value is set at initialization.

constructor. In C++, a special class member function (method) that has the same name as the class and is used to initialize class objects.

default argument. In C++, a default is used when an argument in a method call is not explicitly provided.

delete. A C++ operator that deallocates dynamic storage to destroy an object.

destructor. In C++, a special class member function (method) that has the same name as the class, preceded by (tilde), and is executed when an object is destroyed.

distributed program link. A technique where a program running on one CICS system links to a program running on another system.

encapsulation. The means whereby the inner workings of an object are hidden. An application programmer only has direct access to the external features.

function shipping. A technique whereby a transaction running on one CICS system accesses resources held on another system.

inheritance. The passing of class resources or attributes from a base class to a subclass.

method. An operator or function that is declared as a member of a class.

new. A C++ operator that allocates dynamic storage to create an object.

object. An abstraction consisting of data and the operations associated with that data.

overloading. The redefinition of functions and most standard C++ operators. This typically extends the operations that the function or operator performs to different data types.

polymorphism. The application of a method or function to objects of more than one data type.

subclass. A class that is derived from another class. The subclass inherits the data and methods of the base class and can define new methods or over-ride existing methods to define new behavior not inherited from the parent class.

task. One instance of the execution of a particular CICS transaction.

transaction. One or more programs on a CICS server that can be initiated on request by a CICS user.

transaction routing. A technique whereby a transaction initiated on one CICS system is actually run on another system.

UOW. A CICS unit of work is a set of resource updates.

virtual function. In C++, a class member function that is defined with the keyword **virtual**. The code that is executed when you make a call to a virtual function depends on the type of object for which it is called.

Bibliography

The CICS Transaction Server for z/OS library

The published information for CICS Transaction Server for z/OS is delivered in the following forms:

The CICS Transaction Server for z/OS Information Center
The CICS Transaction Server for z/OS Information Center is the primary source of user information for CICS Transaction Server. The Information Center contains:

- Information for CICS Transaction Server in HTML format.
- Licensed and unlicensed CICS Transaction Server books provided as Adobe Portable Document Format (PDF) files. You can use these files to print hardcopy of the books. For more information, see PDF-only books."
- Information for related products in HTML format and PDF files.

One copy of the CICS Information Center, on a CD-ROM, is provided automatically with the product. Further copies can be ordered, at no additional charge, by specifying the Information Center feature number, 7014.

Licensed documentation is available only to licensees of the product. A version of the Information Center that contains only unlicensed information is available through the publications ordering system, order number SK3T-6945.

Entitlement hardcopy books
The following essential publications, in hardcopy form, are provided automatically with the product. For more information, see The entitlement set."

The entitlement set

The entitlement set comprises the following hardcopy books, which are provided automatically when you order CICS Transaction Server for z/OS, Version 3 Release 1:

Memo to Licensees, GI10-2559
CICS Transaction Server for z/OS Program Directory, GI10-2586
CICS Transaction Server for z/OS Release Guide, GC34-6421
CICS Transaction Server for z/OS Installation Guide, GC34-6426
CICS Transaction Server for z/OS Licensed Program Specification, GC34-6608

You can order further copies of the following books in the entitlement set, using the order number quoted above:

CICS Transaction Server for z/OS Release Guide
CICS Transaction Server for z/OS Installation Guide
CICS Transaction Server for z/OS Licensed Program Specification

PDF-only books

The following books are available in the CICS Information Center as Adobe Portable Document Format (PDF) files:

CICS books for CICS Transaction Server for z/OS

General

CICS Transaction Server for z/OS Program Directory, GI10-2586
CICS Transaction Server for z/OS Release Guide, GC34-6421
CICS Transaction Server for z/OS Migration from CICS TS Version 2.3, GC34-6425

CICS Transaction Server for z/OS Migration from CICS TS Version 1.3, GC34-6423
CICS Transaction Server for z/OS Migration from CICS TS Version 2.2, GC34-6424
CICS Transaction Server for z/OS Installation Guide, GC34-6426

Administration

CICS System Definition Guide, SC34-6428
CICS Customization Guide, SC34-6429
CICS Resource Definition Guide, SC34-6430
CICS Operations and Utilities Guide, SC34-6431
CICS Supplied Transactions, SC34-6432

Programming

CICS Application Programming Guide, SC34-6433
CICS Application Programming Reference, SC34-6434
CICS System Programming Reference, SC34-6435
CICS Front End Programming Interface User's Guide, SC34-6436
CICS C++ OO Class Libraries, SC34-6437
CICS Distributed Transaction Programming Guide, SC34-6438
CICS Business Transaction Services, SC34-6439
Java Applications in CICS, SC34-6440
JCICS Class Reference, SC34-6001

Diagnosis

CICS Problem Determination Guide, SC34-6441
CICS Messages and Codes, GC34-6442
CICS Diagnosis Reference, GC34-6899
CICS Data Areas, GC34-6902
CICS Trace Entries, SC34-6443
CICS Supplementary Data Areas, GC34-6905

Communication

CICS Intercommunication Guide, SC34-6448
CICS External Interfaces Guide, SC34-6449
CICS Internet Guide, SC34-6450

Special topics

CICS Recovery and Restart Guide, SC34-6451
CICS Performance Guide, SC34-6452
CICS IMS Database Control Guide, SC34-6453
CICS RACF Security Guide, SC34-6454
CICS Shared Data Tables Guide, SC34-6455
CICS DB2 Guide, SC34-6457
CICS Debugging Tools Interfaces Reference, GC34-6908

CICSPlex SM books for CICS Transaction Server for z/OS

General

CICSPlex SM Concepts and Planning, SC34-6459
CICSPlex SM User Interface Guide, SC34-6460
CICSPlex SM Web User Interface Guide, SC34-6461

Administration and Management

CICSPlex SM Administration, SC34-6462
CICSPlex SM Operations Views Reference, SC34-6463
CICSPlex SM Monitor Views Reference, SC34-6464
CICSPlex SM Managing Workloads, SC34-6465
CICSPlex SM Managing Resource Usage, SC34-6466
CICSPlex SM Managing Business Applications, SC34-6467

Programming

CICSPlex SM Application Programming Guide, SC34-6468
CICSPlex SM Application Programming Reference, SC34-6469

Diagnosis

CICSPlex SM Resource Tables Reference, SC34-6470
CICSPlex SM Messages and Codes, GC34-6471
CICSPlex SM Problem Determination, GC34-6472

CICS family books

Communication

CICS Family: Interproduct Communication, SC34-6473
CICS Family: Communicating from CICS on System/390, SC34-6474

Licensed publications

The following licensed publications are not included in the unlicensed version of the Information Center:

CICS Diagnosis Reference, GC34-6899
CICS Data Areas, GC34-6902
CICS Supplementary Data Areas, GC34-6905
CICS Debugging Tools Interfaces Reference, GC34-6908

Other CICS books

The following publications contain further information about CICS, but are not provided as part of CICS Transaction Server for z/OS, Version 3 Release 1.

Designing and Programming CICS Applications	SR23-9692
CICS Application Migration Aid Guide	SC33-0768
CICS Family: API Structure	SC33-1007
CICS Family: Client/Server Programming	SC33-1435
CICS Transaction Gateway for z/OS Administration	SC34-5528
CICS Family: General Information	GC33-0155
CICS 4.1 Sample Applications Guide	SC33-1173
CICS/ESA 3.3 XRF Guide	SC33-0661

Related books

Here are some more 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.

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

CICS client manuals

CICS Clients: Administration	SC33-1792
CICS Clients: Messages	SC33-1793
CICS Clients: Gateways	SC33-1821
CICS Family: OO Programming in C++ for CICS Clients	SC33-1923

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 usually in step. However, due to the time required to print and distribute hardcopy books, the BookManager version is more likely to have had last-minute changes made to it before publication.

Subsequent updates will probably be available in softcopy before they are available in hardcopy. This means that at any time from the availability of a release, softcopy versions should be regarded as the most up-to-date.

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 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.

Updates to the softcopy are clearly marked by revision codes (usually a # character) to the left of the changes.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully.

You can perform most tasks required to set up, run, and maintain your CICS system in one of these ways:

- using a 3270 emulator logged on to CICS
- using a 3270 emulator logged on to TSO
- using a 3270 emulator as an MVS system console

IBM Personal Communications provides 3270 emulation with accessibility features for people with disabilities. You can use this product to provide the accessibility features you need in your CICS system.

Index

Special characters

Numerics

A

B

C

D

F

G

H

I

N

O

P

Q

R

S

T

X

Y

Notices

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

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

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

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

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

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

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

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

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

Trademarks

Sending your comments to IBM

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

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

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

- By mail, to this address:

  IBM United Kingdom Limited
  User Technologies Department (MP095)
  Hursley Park
  Winchester
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44–1962–816151
  - From within the U.K., use 01962–816151
- Electronically, use the appropriate network ID:
  - IBMLink: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

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

Product Number: 5655-M15

SC34-6437-03

Spine information:

CICS TS for z/OS

C++ OO Class Libraries

Version 3
Release 1