AS/400 Advanced Series

IBM Access Class Library User's Guide

Version 3

SC41-4623-00

AS/400 Advanced Series

IBM Access Class Library User's Guide

Version 3

SC41-4623-00

Take Note!

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

First Edition (November 1996)

This edition applies to the licensed program IBM VisualAge for C++ for AS/400 (Program 5716-CX5), Version 3 Release 7 Modification 0, and to all subsequent releases and modifications until otherwise indicated in new editions.

Make sure that you are using the proper edition for the level of the product.

Order publications through your IBM representative or the IBM branch serving your locality. If you live in the United States, Puerto Rico, or Guam, you can order publications through the IBM Software Manufacturing Solutions at 800+879-2755. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for readers’ comments may be provided at the back of this publication. You can also mail your comments to the following address:

IBM Corporation
Attention Department 542
IDCLERK
3605 Highway 52 N
Rochester, MN 55901-7829 USA

or you can fax your comments to:

United States and Canada: 800+937-3430
Other countries: (+1)+507+253-5192

If you have access to Internet, you can send your comments electronically to IDCLERK@RCHVMW2.VNET.IBM.COM; IBMMAIL, to IBMMAIL(USIB56RZ).

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

Contents

Tables

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

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 the software interoperability coordinator. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Address your questions to:

IBM Corporation
Software Interoperability Coordinator
3605 Highway 52 N
Rochester, MN 55901-7829 USA

This publication could contain technical inaccuracies or typographical errors.

This publication may refer to products that are announced but not currently available in your country. This publication may also refer to products that have not been announced in your country. IBM makes no commitment to make available any unannounced products referred to herein. The final decision to announce any product is based on IBM's business and technical judgment.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This publication contains small programs that are furnished by IBM as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

Programming Interface Information

This publication is intended to help you develop applications that use the C++ class libraries provided with VisualAge for C++ for AS/400. This publication documents General-Use Programming Interface and Associated Guidance Information provided by VisualAge for C++ for AS/400.

General-Use programming interfaces allow the customer to write programs that obtain the services of VisualAge for C++ for AS/400.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AS/400	OS/2
DB2	OS/400
DB2/400	Presentation Manager
IBM	VisualAge
Integrated Language Environment	400

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

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

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About IBM Access Class Library User's Guide (SC41-4623)

This guide tells you how you can use the classes that are included in the IBM Access Class Library for OS/400.

IBM Access Class Library for OS/400 consists of classes in the following groups:

- Command and program
- Data area
- Database
- Data queue
- Data translation
- Data type
- User space

This guide provides information for the following:

- OS/2 client
- Windows 95 client
- Windows NT** client
- OS/400 server

Who Should Use This Guide?

This guide is intended for skilled C++ programmers who understand the concept of classes and who are familiar with using C++ templates when working with individual class libraries. Use this guide if you want to access OS/400 resources that are commonly used when constructing client/server or OS/400 applications.

For information about other AS/400 publications (except Advanced 36), see either of the following:

- The *Publications Reference* book, SC41-4003, in the AS/400 Softcopy Library.
- The *AS/400 Information Directory*, a unique, multimedia interface to a searchable database that contains descriptions of titles available from IBM or from selected other publishers. The *AS/400 Information Directory* is shipped with the OS/400 operating system at no charge.

For a list of related publications, see the “Bibliography” on page 79.

How to Use This Guide

This guide describes the IBM Access Class Library for OS/400, and it shows you how to write programs that make use of the AS/400 services provided by this library.

A Note about Examples

The examples in this guide explain elements of the C++ class libraries. They are coded in a simple style. They do not try to conserve storage, check for errors, achieve fast run times, or demonstrate all possible uses of a library, class, or member function.

Chapter 1. Introduction to the Access Class Library

The Access Class Library is an object-oriented class library that is used to access OS/400 resources that are commonly used when constructing client/server or OS/400 applications.

Library Organization

The Access Class Library provides the following groups of classes to help you access these resources on an AS/400 system:

Command and Program These classes are used to run AS/400 commands and programs. Command access is supported on OS/2, OS/2 Optimized, Windows 95, Windows NT, and AS/400. Program access is supported on OS/2 Optimized, Windows 95, and Windows NT.

See Chapter 2, “Command and Program Access Classes” on page 9.

Data area These classes are used to create or access an AS/400 data area. This class group supports character, logical, and decimal data areas through member functions such as `read()` and `write()`. Data area access classes are supported on OS/2 Optimized, Windows 95, Windows NT, and AS/400. See Chapter 3, “Data Area Access Classes” on page 15.

Database These classes are used to access data on an AS/400 system by using SQL commands. Database access classes are supported on OS/2, OS/2 Optimized, Windows 95, Windows NT, and AS/400. See Chapter 4, “Database Access Classes” on page 19.

Data queue These classes are used to create or access an AS/400 data queue. Both keyed and nonkeyed data queues are supported. This class group provides a full set of member functions such as `read()`, `write()`, and `peek()`. Data queue access classes are supported on OS/2, OS/2 Optimized, Windows 95, Windows NT, and AS/400. See Chapter 5, “Data Queue Access Classes” on page 37.

Data translation This class is used to convert data between PC and AS/400 formats, such as ASCII data to EBCDIC data. Data translation access classes are supported on OS/2, OS/2 Optimized, Windows 95, and Windows NT. See Chapter 6, “Data Translation Access Class” on page 43.

Data type These classes are used to manipulate data that is in AS/400 formats and to convert data between AS/400 formats and PC C++ formats. Data type access classes

Conventions

	are supported on OS/2 Optimized, Windows 95, and Windows NT. See Chapter 7, “Data Type Access Classes” on page 47.
User space	This class is used to create or access an AS/400 user space. This is a single class with member functions such as `read()` and `write()`. User space access classes are supported on OS/2 Optimized, Windows 95, Windows NT, and AS/400. See Chapter 8, “User Space Access Class” on page 71.

Conventions

File names, names of classes, member functions, and data members follow naming conventions, which are described in the following section.

File and Program Names

All file names provided for the Access Class Library have a maximum of eight characters. Header files start with the characters IC4 and have a file extension of either .HPP or .H.

The x in the name CTTACLxI.LIB is replaced with one of the following numbers:

0 For the OS/2 client
1 For the Optimized OS/2 client
4 For the Windows 95/NT client

The DLLs follow a similar naming convention.

All files are integrated into the VisualAge for C++ directory structure.

The Access Class Library run time for the AS/400 is provided by base operating system service programs in library QSYS. The names of the service programs start with the characters QZCC.

Names of Classes, Member Functions, and Data Members

Names of classes, member functions, and data members follow these rules:

- All class names are prefixed with `IC4`.
- Class names are mixed case with the first letter of each word in upper case, as in `IC4KeyedDataQueue`.
- Member function names and data member names are mixed case with the first letter always lower case, as in `getParameterDescription()`. There are a few exceptions to this, for consistency with similar functions in other VisualAge for C++ libraries.
- Acronyms within a name are uppercase, as in `IC4SQLStatement`.
- OS/400 object names must be specified as qualified names in the library file system. The following are examples:

- /QSYS.LIB/MYLIB.LIB/MYDQ.DTAQ identifies a data queue named MYDQ in a library named MYLIB.
- /QSYS.LIB/CHGUSRPRF.CMD specifies the Change User Profile (CHGUSRPRF) command in the QSYS library.

The library qualifier can be a special value to indicate that the current library or the library list should be used. For example, /QSYS.LIB/%LIBL%.LIB/MYCMD.CMD identifies the MYCMD command in the library list for the job. For programs targeted to a client, this job is the Client Access server job. For programs targeted to the OS/400 server, this job is the job the program is running under. For more information about OS/400 object names in the library file system, see the *Integrated File System Introduction* book, SC41-4711.

Exception Handling

Exceptions are thrown when serious errors are encountered. Therefore, programs should monitor for exceptions. The Access Class Library defines an IC4Exception class that derives from the IException class. Each class group derives a unique exception class from IC4Exception and throws exceptions of that type. These specialized exceptions allow a program to easily discriminate among errors generated by different class groups. All exceptions are thrown using the C++ exception support.

See Appendix A, “Error Codes” on page 75 for a description of the information contained in an IC4Exception.

Developing Client Applications with the Access Class Library

In this section, Client Access/400 clients, run-time support, linking and Dynamic Link Libraries (DLLs) are discussed. Each client is discussed separately.

The Access Class Library can be used to develop client/server or OS/400 applications. The Windows 95 client, Windows NT client, Optimized OS/2 client, and OS/2 client are supported; however, some functions are not supported on every client or server.

Client run-time support is provided either through static libraries or through a set of dynamic link libraries (DLLs). Certain class groups also require DLLs provided by Client Access.

Personal computers that run applications developed using the Access Class Library require client-specific run-time environments.

Your program can link either statically or dynamically to the Access Class Library. You can select the option that best meets your needs.

Static linking is the recommended option for linking to the Access Class Library. When using this option, your program must statically link to the Open Class Library.

Developing Windows 95 and Windows NT Applications with the Access Class Li

When linking a program with one of the static library files, you must invoke the linker through the compiler and specify the */Tdp* option, because some of the access classes use templates.

The Access Class Library DLLs that provide the client run-time support when an application links dynamically to the Access Class Library are not distributed to the end users of an IBM product. If you choose to dynamically link to the Access Class Library by using a CTTACLxI.LIB import library, you must rename and distribute the Access Class Library DLLs with your application. Licensing requirements state that when you distribute the DLLs as part of your application, you must rename the DLLs. To rename the DLLs, use the DLL Rename Utility (DLLRNAME) provided with VisualAge for C++. More information on renaming DLLs can be found online in the *IBM VisualAge for C++ for Windows User's Guide*.

When linking dynamically, your program must link dynamically to the Open Class Library, because the Access Class Library uses the DLLs supplied by the Open Class Library. For information about linking to the Open Class Library on the OS/2 clients, see “Linking an Application to the Open Class Library” in the *IBM VisualAge C++ for OS/2 Open Class Library User's Guide*. For information about linking to the Open Class Library on the Windows clients, see “Linking an Application to the Open Class Library” in the *IBM VisualAge C++ for Windows** Open Class Library User's Guide*.

When you distribute an application, the run-time support for the Access Class Library is included with your application, either statically linked or as a set of DLLs that you renamed. If an error is reported in the Access Class Library and is later fixed by IBM via a PTF, to incorporate the correction:

- You must relink and redistribute your application if you are using the static library.
- You may need to relink and redistribute your application if you are using the import library.

Developing Windows 95 and Windows NT Applications with the Access Class Library

To develop applications using the Access Class Library and the AS/400 Client Access Windows 95/NT client, the following software environment is required:

- Windows 95 or Windows NT
- AS/400 Client Access for Windows 95/NT
- VisualAge for C++ for Windows and its prerequisites
- An ODBC software developers toolkit for Windows 95 or Windows NT if you use the database access classes.

The following library files are provided for use with the Windows 95 or Windows NT client:

Library File	Description
CTTACL4.LIB	Code library for static linking
CTTACL4I.LIB	Import library for dynamic linking

Developing Optimized OS/2 Applications with the Access Class Library

When linking a Windows 95 or Windows NT program to the Access Class Library using a static library file, the program must also link with the appropriate library for each of the functions that are used but not provided by the Access Class Library. All Access Class Library functions must link using the Client Access library CWBAPI.LIB. When using the database access classes, your program must link using an ODBC software development kit for Windows 95 or Windows NT.

The Access Class Library provides the following DLLs:

Table 1. Access Class Library DLLs for Windows

Access Class	Windows 95 and Windows NT
Command	CTTACP4.DLL
Data Area	CTTADA4.DLL
Database	CTTASQ4.DLL
Data Queue	CTTADQ4.DLL
Data Translation	CTTAXL4.DLL
Data Type	CTTADT4.DLL
Program	CTTAPG4.DLL
User Space	CTTAUS4.DLL

To run an application, you need the following:

- Your application
- Renamed Access Class Library DLLs distributed with your application if linking dynamically
- Windows 95 or Windows NT
- AS/400 Client Access for Windows 95/NT
- The ODBC DLL from an ODBC software development kit for the appropriate Windows version if any database classes are used

Developing Optimized OS/2 Applications with the Access Class Library

To develop applications using the Access Class Library and Optimized OS/2 client, the following software environment is required:

- OS/2 2.1 or higher or OS/2 Warp
- AS/400 Client Access Optimized for OS/2
- VisualAge for C++ for OS/2 Version 3 and its prerequisites
- An ODBC software development kit for OS/2 if you use the database classes on the Optimized OS/2 client

Note: If you use the data queues classes on the Optimized OS/2 client, you must use the version of OS/2 Communications Manager that is shipped with the Optimized OS/2 client or a more recent version.

The following library files are provided for use with the Client Access Optimized for OS/2 client:

Developing OS/2 Applications with the Access Class Library

Library File	Description
CTTACL1.LIB	Code library for static linking
CTTACL1I.LIB	Import library for dynamic linking

When linking to the Optimized OS/2 client and using the database access classes, your program must link using an ODBC software development kit for OS/2. All Access Class Library functions must link using the library CWB.LIB.

The Access Class Library provides the following DLLs:

Table 2. Access Class Library DLLs for Optimized OS/2

Access Class	Optimized OS/2 Client
Command	CTTACP1.DLL
Data Area	CTTADA1.DLL
Database	CTTASQ1.DLL
Data Queue	CTTADQ1.DLL
Data Translation	CTTAXL1.DLL
Data Type	CTTADT1.DLL
Program	CTTAPG1.DLL
User Space	CTTAUS1.DLL

To run an application, you need the following:

- Your application
- Renamed Access Class Library DLLs distributed with your application if linking dynamically
- OS/2 2.1 or higher or OS/2 Warp
- Client Access Optimized for OS/2 Version 3 Release 1 Modification 1 or later
- The ODBC DLL from an ODBC software development kit for OS/2 if any database classes are used

Developing OS/2 Applications with the Access Class Library

To develop applications using the Access Class Library and OS/2 client, the following software environment is required:

- OS/2 2.1 or higher or OS/2 Warp
- AS/400 Client Access for OS/2 client
- VisualAge for C++ for OS/2 Version 3 and its prerequisites

The following library files are provided for use with the AS/400 Client Access for OS/2 client:

Library File	Description
CTTACL0.LIB	Code library for static linking
CTTACL0I.LIB	Import library for dynamic linking

When compiling for the OS/2 client, you must define the preprocessor variable __IC4PCS__. This preprocessor variable ensures that the correct member data is defined for the Access Class Library classes for use on this client.

When linking with a static library file, your program must also link to the AS/400 Client Access for OS/2 libraries for those OS/2 client functions that are used. The following libraries are required:

Table 3. Required Libraries When Linking an Application for the OS/2 Client

Access Class	Required Client Access Libraries
Command	EHNSRAPI.LIB EHNALAPI.LIB
Database	EHNRQAPI.LIB
Data Queue	EHNDQ.LIB EHNALAPI.LIB
Data Translation	EHNALAPI.LIB

The Access Class Library provides the following DLLs:

Table 4. Access Class Library DLLs for OS/2

Access Class	OS/2 Client
Command	CTTACP0.DLL
Database	CTTASQ0.DLL
Data Queue	CTTADQ0.DLL
Data Translation	CTTAXL0.DLL

To run an application, you need the following:

- Your application
- Renamed Access Class Library DLLs distributed with your application if linking dynamically
- AS/400 Client Access for OS/2 installed

Developing Native AS/400 Applications with the Access Class Library

This section discusses developing, distributing, and running applications that use the Access Class Library and run natively on the AS/400 system.

To develop native AS/400 applications that use the Access Class Library, the following software environment is required:

- VisualAge for C++ for AS/400 and prerequisites
- An AS/400 system at V3R6 or later
- Licensed program 57xx-SS1 option 13 (System Openness Includes) installed if you use the database access classes.

When specifying the *systemName* parameter on member functions, only the name of the AS/400 system that the program is running on is valid. The member functions default to the current system name when a NULL pointer is passed, so it is a good practice to use a NULL pointer instead of specifying a system name.

You must define the preprocessor variable `__IC40S400__` when using access classes.

Your program binds to the Access Class Library native run-time support by using the default binding directories for VisualAge C++ for OS/400. Depending on the target release, the bind step uses either binding directory QYPPLR or binding directory QYPPLR370. Refer to the *VisualAge for C++ for AS/400 C++ User's Guide*, SC09-2416, for more information on compiling and binding C++ programs for the AS/400 system.

The OS/400 server run-time support for the Access Class Library is shipped with the base operating system as a set of service programs (*SRVPGM) in the QSYS library for V3R7 and V3R2. For V3R6, the run-time support is available as a PTF to the base operating system.

To maintain release-to-release compatibility for programs that use the native Access Class Library, a new version of the Access Class Library run-time is required when the Access Class Library is enhanced. An additional set of run-time service programs is shipped with releases after V3R6 and V3R2. The run-time service program to use is determined during the compilation step depending on which release is targeted. The service programs and their target release are shown in the following table.

Table 5. Service Programs and Corresponding Target Releases

Class Group	**V3R6 and V3R2**	**V3R7**
Command	QZCCCP	QZCCCP370
Data Area	QZCCDA	QZCCDA370
Database	QZCCDB	QZCCDB370
Data Queue	QZCCDQ	QZCCDQ370
User Space	QZCCUS	QZCCUS370

Programs that are compiled and bound for V3R6 or V3R2 and then subsequently moved to V3R7 will use the run-time that is compatible with the release for which the programs were created.

To run an application, you need the following:

- Your application
- An AS/400 system at V3R6 or later (this does include V3R2)
- Licensed program 57xx-SS1 option 13 (System Openness Includes) installed if you use the database access classes.

Chapter 2. Command and Program Access Classes

The Access Class Library command and program access class group provides a VisualAge for C++ class interface for applications that submit commands to an AS/400 system or call programs that run on an AS/400 system.

Calling a command is supported on the following:

- OS/2
- OS/2 Optimized
- Windows 95
- Windows NT
- AS/400

Calling a program is supported on the following:

- OS/2 Optimized
- Windows 95
- Windows NT

Client Access is required for you to use command and program access classes.

When compiling for the OS/2 client, you must define the preprocessor variable `__IC4PCS__`. When compiling for the OS/400 server, you must define the preprocessor variable `__IC4OS400__`.

Introduction to the Command and Program Access Classes

The command and program access class group includes these classes:

Command processor	There is a single command processor class, `IC4CommandProcessor`. `IC4CommandProcessor` provides an interface to run commands and programs on an AS/400 system. All calls to run programs or commands using the same command processor object occur sequentially. Command strings can be specified directly on a call to run a command, or the OS/400 command class can be used to construct a command string.
OS/400 command class	`IC4OS400Command` represents a command that is run on an AS/400 system. You can use this class to construct a command string for a command that is run on an AS/400 system, and then pass this object to the `IC4CommandProcessor::run()` member function.

Using the Command Access Class

AS/400 program class	`IC4Program` represents a program that is run on an AS/400 system. It can be used by client applications that need to call a program on the AS/400 system and received output when the program completes.

Exception Classes

The Command Exception class, `IC4CommandException`, is derived from IC4Exception. All exceptions that are thrown by `IC4OS400Command` are instances of `IC4CommandException`.

The Program Exception class, `IC4ProgramException`, is derived from `IC4Exception`. All exceptions that are thrown by `IC4Program` are instances of `IC4ProgramException`.

The Command Processor Exception class, `IC4CommandProcessorException`, is also derived from `IC4Exception`. All exceptions that are thrown by `IC4CommandProcessor` are instances of `IC4CommandProcessorException`.

Using the Command Access Class

Following are examples of how the command access class can be used.

Running an OS/400 Command

Submitting an OS/400 command can be done in two ways: with a command string or with a command object.

A command can be submitted without creating a command object by passing a command string to the `IC4CommandProcessor::run()` member function. Following is an example of submitting an OS/400 command using a command string:

```
IC4CommandProcessor myAS400("SystemA");
myAS400.open();
myAS400.run("CRTLIB TESTLIB TEXT('test library')");
myAS400.close();
```

When using the command object, the order of the parameters does not matter. However, you must specify a keyword for each parameter. The command object is then passed to the `IC4CommandProcessor::run()` member function. Following is an example of submitting an AS/400 command using a command object:

```
IC4CommandProcessor myAS400("SystemA");
IC4OS400Command myCmd("/QSYS.LIB/%LIBL%.LIB/crtlib.CMD");
myCmd.addParameter("LIB", "TESTLIB");
myCmd.addParameter("TEXT", "'test library'");
myAS400.open();
myAS400.run(myCmd);
myAS400.close();
```

Accessing Messages

When you submit a command, OS/400 may return messages. These messages may be error messages, or they may be informational or completion messages. There are two ways to access messages:

- When an exception is thrown, messages are in the exception text. Exceptions are thrown if there are any error messages.
- Messages can always be accessed by using the `message()` member function of the command processor object.

Following is an example of calling a command and printing the returned messages:

```
IC4CommandProcessor myAS400("SystemA");
IC4OS400Command myCmd("/QSYS.LIB/%LIBL%.LIB/crtlib.CMD");
myCmd.addParameter("LIB", "TESTLIB");
myCmd.addParameter("TEXT", "'test library'");
myAS400.open();
myAS400.run(myCmd);

cout << "Messages:\n";     // print out message heading
IString msg;

// print out all messages
while((msg = myAS400.message()) != "")
  cout << msg << endl;

myAS400.close();
```

Using the Program Access Class

To run an AS/400 program, you must first define the calling syntax of the AS/400 program if the program has any arguments. Defining the calling syntax consists of describing the type, I/O mode and order of each program argument. Each program argument is represented by a **data item**. See Chapter 7, “Data Type Access Classes” on page 47 for more information on data items. The advantage of using data items to represent arguments is that the arguments can be manipulated as normal C++ types and are converted between C++ types and AS/400 system types automatically. The following section, “Defining a Program,” describes how to define the calling syntax of an AS/400 program.

After the calling syntax is defined, the program can be run using the `IC4CommandProcessor::run()` function. When a program is run, arguments are converted as needed between C++ types and AS/400 system types. The section “Running a Program” on page 13 describes how to run a program.

Defining a Program

The first step in defining a program is to construct an `IC4Program` object as follows:

```
IC4Program pgm("/QSYS.LIB/MYLIB.LIB/MYPROGRAM.PGM");
```

where MYLIB and MYPROGRAM are the library name and program name of the program to be run.

Using the Program Access Class

The next step is to define the arguments to the program in the order that the program expects them. For example, assume that the AS/400 program MYPROGRAM in library MYLIB takes the following arguments:

Number	Description	Mode	AS/400 Type
1	Output argument	Output	Char(*)
2	Input argument	Input	Binary(4)
3	System API error code	Input/Output	Char(*)

Further assume that the data returned in the first argument has the following form:

Decimal Offset	Hexadecimal Offset	AS/400 Type
0	0	Binary(4)
4	4	Char(7)
11	B	Char(1)

Also assume that the data of the third argument has the following form. Note that this is the form of the **error code parameter** that is returned by many OS/400 APIs.

Decimal Offset	Hexadecimal Offset	AS/400 Type	Description
0	0	Binary(4)	Bytes provided
4	4	Binary(4)	Bytes available
8	8	Char(7)	Exception ID
15	F	Char(1)	Reserved
16	10	Char(128)	Exception data

Because arguments 1 and 3 are complex, the `IC4Struct` aggregate data type is used to define them. A simple type can then be used for the second argument. In the following example, the arguments are defined in terms of data types, and then those definitions are provided to the program.

Next, `IC4Struct` objects for the first and third arguments should be defined. A simple type for the second argument is used. The following example shows you this:

```
// Declare constants for initialization.  See note after example.
   const char blanks7[]  = "       ";
   const char blanks128[]= "                                        "
         "                                                 "
         "                                                 ";
// Construct the first argument.  Declare a structure with 3 members
// and add the members to the structure.
   IC4Struct arg1;
   IC4Bin4 arg1Field1;
   IC4Char arg1Field2(7,7);
   IC4Byte arg1Field3(1);
   arg1 << arg1Field1 << arg1Field2 << arg1Field3;

// Construct a simple type for the second argument.
   IC4Bin4 arg2;
```

```
// Construct the third argument.  Declare a structure with 5 members
// that are initialized, and add the members to the structure.
   IC4Struct arg3;
   IC4Bin4 bytesProvided(144);
   IC4Bin4 bytesAvailable;
   IC4Char exceptionID(blanks7,7);
   IC4Byte reserved(1);
   IC4Char exceptionData(blanks128,128);
   arg3 << bytesProvided << bytesAvailable << exceptionID
        << reserved << exceptionData;
```

Note: Because the third argument is an input argument (as well as an output argument), the program object converts the value of the input argument to the external format before making the call. The conversion fails (an exception is thrown) if any IC4Char data item that is part of the argument contains a character, such as a null character, that cannot be converted successfully. The constant character arrays in the preceding example are used to initialize the IC4Char items to avoid an exception condition.

At this point, each argument is defined as a data type. Next, the arguments are provided to the program in the proper order. The `IC4Program` class defines three member functions for defining AS/400 program call arguments: `in()`, `inOut()` and `out()`. The argument definitions can be provided to the program in a single statement:

```
pgm.out(arg1).in(arg2).inOut(arg3);
```

or in multiple statements:

```
pgm.out(arg1);
pgm.in(arg2);
pgm.inOut(arg3);
```

Running a Program

Running a program requires an `IC4CommandProcessor` object that has been opened. The following example assumes the following:

- An `IC4Program` object called *pgm* has already been constructed
- The calling syntax for *pgm* has been defined
- The arguments for *pgm* have been initialized

The example shows how to construct the command processor object, open a conversation to the default AS/400 system, and run the program:

```
IC4CommandProcessor defaultAS400; // Construct the
                                  // command processor.
defaultAS400.open();              // Open a conversation.

defaultAS400.run(pgm);            // Run the program
```

When a program is run, the following sequence of steps takes place:

1. Input and input/output arguments are converted to the appropriate AS/400 system format. The conversion is done by the `translateToExternal()` member function for each argument.

2. The AS/400 program is called with the converted arguments.
3. The return values of the output and input/output arguments are converted from AS/400 system format to the appropriate C++ format. The conversion is done by the `translateFromExternal()` member function for each argument.

You need to be aware of the following:

1. **The input and input/output arguments of the `translateToExternal()` member function must be able to complete.**

   Because input and input/output arguments are converted from the local (C++) format to the external (AS/400 system) format when a program is run, the local format data of the arguments must be in a valid state for the conversion. Otherwise an exception is thrown. You are likely to see this situation when you have a structure, such as the OS/400 error code parameter, that is passed as an input/output argument and a portion of the structure is an `IC4Char` data item that is used only to return data. The program object does not know that the data for the `IC4Char` object is not needed as input. It assumes that the entire argument is needed and attempts to convert the data.

2. **Output arguments must have their external data size set.**

   The program needs to know how much storage space is available for the return value in an output argument. The program returns the value in the external (AS/400 system) format data of the argument before converting the data to the local (C++) format. You may see this situation with `IC4Char` and `IC4Byte` arguments. If the external data size of an argument is zero when a program is run, an exception is thrown.

Chapter 3. Data Area Access Classes

A **data area** is an AS/400 system object used for sharing type-specific data, such as character data, decimal data, or logical data, between programs. The object type identifier for the data area is *DTAARA. The Access Class Library data area classes provide a VisualAge for C++ class interface to help build applications that use these AS/400 objects.

The interface supports three types of data areas: decimal, character, and logical. These three types of data areas have important differences. For example, a decimal data area can have only decimal data, a character data area can have only character data, and a logical data area can have only logical data. All three data areas have `read()` and `write()` member functions to access the data.

Data area access classes are supported on the following:

- OS/2 Optimized
- Windows 95
- Windows NT
- AS/400

Introduction to the Data Area Access Classes

The classes provided for data areas are as follows:

Base Data Area	The Base Data Area class, `IC4BaseDataArea`, provides a common interface to all types of data areas. This includes the following: • Opening or closing a data area • Resetting the contents of a data area • Deleting a data area
Data Area Exception Class	The Data Area Exception class, `IC4DataAreaException`, is derived from `IC4Exception`. All exceptions thrown by the data area classes are instances of IC4DataAreaException.

The following classes are derived from Base Data Area and provide extensions for specific types of data areas:

Character Data Area	The Character Data Area class, `IC4CharacterDataArea`, provides the extended interface to a character data area, such as creating a character data area or writing character data to it. It also provides the extensions to the base class for character data areas.
Decimal Data Area	The Decimal Data Area class, `IC4DecimalDataArea`, provides the extended interface to a decimal data area, such as creating a decimal data area or writing

	decimal data to it. It also provides the extensions to the base class for decimal data areas.
Logical Data Area	The Logical Data Area class, `IC4LogicalDataArea`, provides the extended interface to a logical data area, such as creating a logical data area or writing logical data to it. It also provides the extensions to the base class for logical data areas.

Using the Data Area Access Classes

Following are examples of the how data area classes can be used.

Creating and Deleting a Data Area

To create and delete data areas on an AS/400 system, member functions `create()` and `destroy()` are used. The `create()` member function does an implicit `open()` function call, and the `destroy()` member function does an implicit `close()` function call. Following is an example of creating a decimal data area:

```
IC4DecimalDataArea myDDA("/QSYS.LIB/mylib.LIB/mydda.DTAARA", "SystemA");
myDDA.create(10, 2, 0.0, "My Data Area", "*EXCLUDE");
...
myDDA.destroy();
```

The above example specifies all parameters. The following example of creating a character data area uses the minimum number of parameters:

```
IC4CharacterDataArea myCDA("/QSYS.LIB/mylib.LIB/mycda.DTAARA", "SystemA");
myCDA.create();
...
myCDA.destroy();
```

Writing Data to a Data Area

Writing data to a data area is quite simple. Following is an example of writing data to each type of data area:

```
IC4DecimalDataArea myDDA("/QSYS.LIB/mylib.LIB/mydda.DTAARA", "SystemA");
IC4CharacterDataArea myCDA("/QSYS.LIB/mylib.LIB/mycda.DTAARA", "SystemA");
IC4LogicalDataArea myLDA("/QSYS.LIB/mylib.LIB/mylda.DTAARA", "SystemA");

myDDA.open();
myDDA.write(111.11);
myDDA.close();

myCDA.open();
myCDA.write("character data area test data");
myCDA.close();

myLDA.open();
myLDA.write('1');
myLDA.close();
```

Reading Data from a Data Area

Reading data from a data area is also quite simple. Following is an example of reading data from each type of data area:

```
IC4DecimalDataArea myDDA("/QSYS.LIB/mylib.LIB/mydda.DTAARA", "SystemA");
IC4CharacterDataArea myCDA("/QSYS.LIB/mylib.LIB/mycda.DTAARA", "SystemA");
IC4LogicalDataArea myLDA("/QSYS.LIB/mylib.LIB/mylda.DTAARA", "SystemA");

double num;
char charData[25];
char logicalData;

myDDA.open();
myDDA.read(num);

myCDA.open();
myCDA.read(charData, sizeof(charData), 1);

myLDA.open();
myLDA.read(logicalData);
...
myDDA.close();
myCDA.close();
myLDA.close();
```

Overwriting a Data Area

Only the Character Data Area class provides a member function to replace all the data in the data area. Following is an example of overwriting character data in a data area:

```
IC4CharacterDataArea myCDA("/QSYS.LIB/mylib.LIB/mycda.DTAARA", "SystemA");

myCDA.open();
IString newData = "character data area overwrite.";
myCDA.overwrite(newData);
myCDA.close();
```

Using the Data Area Access Classes

Chapter 4. Database Access Classes

The Access Class Library database access class group provides a VisualAge C++ interface for applications that work with the AS/400 database. The database access classes use a call-level interface to simplify creating an SQL environment, managing database connections, preparing and executing SQL statements, and working with data in the result set.

Database access classes are supported on the following:

- OS/2
- OS/2 Optimized
- Windows 95
- Windows NT
- AS/400

When compiling for the OS/2 client, you must define the preprocessor variable `__IC4PCS__`. When compiling for the OS/400 server, you must define the preprocessor variable `__IC4OS400__`.

Introduction to the Database Access Classes

The database access classes provide the following classes:

- `IC4SQLEnvironment`
- `IC4SQLDatabase`
- `IC4SQLStatement`
- `IC4SQLResultSet`
- `IC4SQLVariable`
- `IC4SQLVariableList`
- `IC4SQLRow`
- `IC4SQLException`

For readability, the following documentation refers to instances of classes as “database objects” or “statement objects.”

Environment Class

The Environment class, `IC4SQLEnvironment`, creates the SQL environment. It is used as input to the constructor for `IC4SQLDatabase`. One environment object must be created before any database operation can be performed. Only one environment object can be instantiated for an application. The Environment class provides the ability to obtain information about data sources and to commit or rollback all open transactions across all connections with one member function call.

Database Class

The Database class, `IC4SQLDatabase`, provides the interface to manage a database connection to the AS/400 system. This includes the following:

Database Access Classes

- Starting and ending sessions
- Transaction support (commit, rollback)
- Retrieving information about the AS/400 database
- Determining function support
- Functions for setting attributes of the connection
- Catalog functions

Statement Class

The Statement class, `IC4SQLStatement`, handles the preparation and execution of SQL statements. This includes the following:

- Setting parameters (binding a parameter marker to a user variable)
- Preparing an SQL text string
- Executing a prepared SQL text string
- Executing an SQL text string directly (no preparation)
- Setting and retrieving various statement attributes that affect statement behavior and the behavior of result sets that are generated.

When creating a statement object, you must specify an existing, connected database object as a parameter. The statement object can be used to perform SQL operations only on the connection provided by this database object. If a database object is disconnected or deleted, all statement objects created under it become unusable.

Result Set Class

The Result Set class, `IC4SQLResultSet`, manages and provides access to the result sets generated by SQL SELECT statements and by the catalog functions of database objects. This includes the following:

- Binding and unbinding result columns to user variables or row objects
- Fetching rows from the result set
- Getting column data from a fetched row into user variables
- Retrieving the column attributes or number of columns
- Closing the result set (this function deallocates resources used for the SELECT statement)

Variable and Variable List Classes

The Variable class, `IC4SQLVariable`, is a convenience class that allows programs to store information needed for binding a column, binding a parameter, and getting column data after the row has been fetched.

The Variable List class, `IC4SQLVariableList`, allows grouping a set of `IC4SQLVariable` objects into a collection and passing the set as a single parameter to the `bindColumns()`, `bindParameters()`, and `getData()` member functions.

Row Class

The Row class, `IC4SQLRow`, is a convenience class that allows you to fetch and store an entire row of data. The Row class provides stream functions to print out the contents of a row. It also allows access to the data in the format in which it is stored.

Database Exception Class

The Database Exception class, `IC4SQLException`, is derived from IC4Exception. All exceptions thrown by the database access classes are instances of `IC4SQLException`.

Developing Client Applications with the Database Access Classes

The Database Access Classes use both an Open Database Connectivity (ODBC) driver and AS/400 Client Access to access data on the AS/400. Therefore, to make full use of the database access classes, knowledge of ODBC and SQL are essential. Below is a brief overview of ODBC. For a complete understanding of the ODBC functions, refer to the *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*.

Microsoft Open Database Connectivity (ODBC) is an industry-standard interface defined by the Microsoft Corporation for database connectivity. This standard allows applications to access data using the Structured Query Language (SQL). The four main components in the ODBC environment used by the database access classes are as follows: Access Class Library Database Access Classes, ODBC driver manager, Client Access ODBC driver, and the AS/400 database.

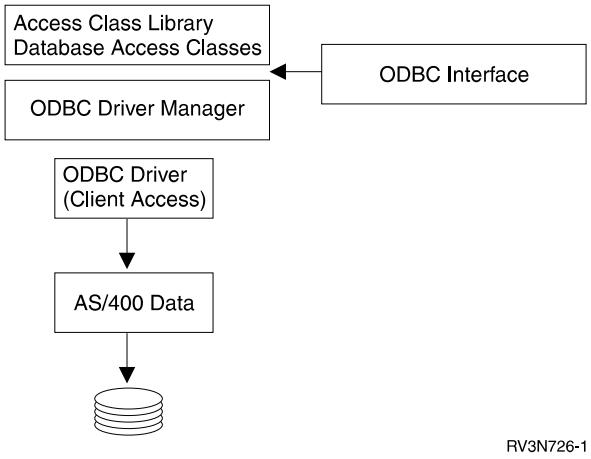

Figure 1. ODBC Components

The ODBC components are as follows:

Access Class Library Database Access Classes
Database Access Classes call ODBC functions to submit SQL requests and retrieve results.

ODBC Driver Manager
The ODBC Driver Manager loads ODBC drivers and routes function calls from the Database Access Classes to the proper ODBC environment. The Database Access Classes require the Client Access ODBC driver.

Database Access Classes

Client Access ODBC Driver The ODBC driver processes ODBC function calls, submits requests to the database management system, and returns results to the driver manager.

AS/400 Data Source The data source is the component to which applications connect. The data source contains the data the user of the applications wants to access, the database management system and its associated operating system, and any network used to access the database management system.

ODBC API Conformance Levels

Three conformance levels are defined for ODBC: core functions, level 1 functions, and level 2 functions. The core functions correspond to the X/Open** and SQL Access Group Call Level Interface specifications. Level 1 and level 2 functions correspond to Microsoft extensions for completeness. Refer to Table 6 for the level support for your platform.

ODBC Versions

Currently, two versions of ODBC are available. You need to know which version you are using because the format of the function may change. Refer to the *IBM Access Class Library for Windows Reference*, SC41-4622, and to the individual header files for more details on the these changes. Refer to Table 6 for the version supported on your platform.

Table 6 (Page 1 of 2). Database Access ODBC Support

Client Access Version	Level of Support	Version of ODBC
AS/400 Client Access for OS/2	Core functions except: SQLCancel SQLGetCursorName SQLRowCount SQLSetCursorName Level 1 SQLGetFunctions SQLSetParam SQLGetData	1
AS/400 Client Access Optimized for OS/2	Core Level 1 except: SQLDriverConnect Level 2 except: SQLBrowseConnect SQLDrivers	1

Table 6 (Page 2 of 2). Database Access ODBC Support

Client Access Version	Level of Support	Version of ODBC
AS/400 Client Access for Windows 95/NT	Core Level 1 except: SQLDriverConnect Level 2 except: SQLBrowseConnect SQLDrivers	1 and 2

ODBC Constants

Many database access member functions have parameters that take predefined constant values. For the constant names, see *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*. File IC4SQDEF.H, for use with the OS/2 client, provides the core constant values. The full set of ODBC constant values can be obtained by purchasing an ODBC software development kit. To use all the ODBC constants, the header file from that kit must be included before any Access Class Library header file.

Database Access Mapping to ODBC

In most cases, a one-to-one mapping exists between the Database Access member functions and the ODBC functions. However, cases do exist where the names are different.

Because of the close parallels between the Access Class Library database access objects and ODBC, many behaviors and restrictions of ODBC carry over to the database access classes.

Refer to the *IBM Access Class Library for Windows Reference*, SC41-4622, or to the individual header files for the ODBC API used for each member function.

Developing Native AS/400 Applications with the Database Access Classes

The Database Access Classes use the DB2 for OS/400 SQL Call Level Interface to access data on the AS/400 system. Therefore, to make full use of the database access classes, you need to have knowledge of DB2 for OS/400 and SQL. For more information, refer to the *DB2 for OS/400 SQL Call Level Interface*, SC41-4806.

The following sections provide a brief overview of the native AS/400 database access classes, and an explanation of the major differences between developing client applications using the database access classes and developing native AS/400 applications using the database access classes.

Developing Native AS/400 Applications with the Database Access Classes

Software Requirements

To use the native AS/400 database access classes, the following software environment is required:

- The licensed program 5716-SS1 option 13 (System Openness Includes) must be installed
- The preprocessor macro `__IC4OS400__` must be defined for when you compile the program. If this define is omitted, the database access classes will fail to compile.
- The database name must be in the remote database directory. To add the database to the directory, issue the AS/400 CL command, Add Relational Database Directory Entry: `ADDRDBDIRE RDB(<database name>) RMTLOCNAME(*LOCAL)`

Porting Client Applications to Native AS/400

For programs being ported from the OS/2 client, surround any ODBC header file includes with `#ifndef __IC4OS400__ ... #endif` to prevent duplicate definitions.

Additional Member Functions

You can set environment options such as system naming mode. See the `setEnvironmentOption()` member function in the *IBM Access Class Library for OS/400 Reference*, SC41-4620.

Unsupported Member Functions

The following member functions are not supported by the native AS/400 database access classes. An `IC4SQLException` object is thrown; this exception returns 1 from errorId() and contains 0003 in the first text line. For cases where some signature forms of the function are supported and some are not, the signature forms that are not supported are shown.

```
IC4SQLDatabase::columnPrivileges
IC4SQLDatabase::connectOption(unsigned short option,
                              char *         value)
IC4SQLDatabase::connectOption(unsigned short option,
                              IString&       value)
IC4SQLDatabase::foreignKeys
IC4SQLDatabase::primaryKeys
IC4SQLDatabase::procedureColumns
IC4SQLDatabase::procedures
IC4SLDatabase::setConnectOption(const unsigned short option,
                                const char *         value)
IC4SQLDatabase::tablePrivileges
IC4SQLDatabase::typeInformation
IC4SQLEnvironment::dataSources
IC4SQLResultSet::bindColumn(unsigned short column,
                            DATE_STRUCT&   variable,
                            unsigned long *bytesAvailable)
IC4SQLResultSet::bindColumn(unsigned short column,
                            TIME_STRUCT&   variable,
                            unsigned long *bytesAvailable)
IC4SQLResultSet::bindColumn(unsigned short      column,
                            TIMESTAMP_STRUCT&   variable,
```

```
                            unsigned long *     bytesAvailable)
IC4SQLResultSet::extendedFetch
IC4SQLResultSet::getData(unsigned short column,
                         DATE_STRUCT&   variable,
                         unsigned long *bytesAvailable)
IC4SQLResultSet::getData(unsigned short column,
                         TIME_STRUCT&   variable,
                         unsigned long *bytesAvailable)
IC4SQLResultSet::getData(unsigned short      column,
                         TIMESTAMP_STRUCT&   variable,
                         unsigned long *     bytesAvailable)
IC4SQLResultSet::moreResults
IC4SQLResultSet::setPosition
IC4SQLStatement::bindParameter(short         pnumber,
                               DATE_STRUCT&  value,
                               short         sqlType)
IC4SQLStatement::bindParameter(short         pnumber,
                               TIME_STRUCT&  value,
                               short         sqlType)
IC4SQLStatement::bindParameter(short              pnumber,
                               TIMESTAMP_STRUCT&  value,
                               short              sqlType)
IC4SQLStatement::bindParameter(short         pnumber,
                               short         parameterType,
                               short         cType,
                               short         sqlType,
                               unsigned long precision,
                               short         scale,
                               void *        value,
                               long          maxSize,
                               unsigned long valueLength)
IC4SQLStatement::getParameterDescription
IC4SQLStatement::moreResults
IC4SQLStatement::numberParameters
IC4SQLStatement::setParameterOptions
IC4SQLVariable::IC4SQLVariable(DATE_STRUCT&  variable,
                               short         position,
                               short         sqlType,
                               long *        bytesAvailable)
IC4SQLVariable::IC4SQLVariable(TIME_STRUCT&  variable,
                               short         position,
                               short         sqlType,
                               long *        bytesAvailable)
IC4SQLVariable::IC4SQLVariable(TIMESTAMP_STRUCT & variable,
                               short              position,
                               short              sqlType,
                               long *             bytesAvail
```

Unsupported Types

The following types are not supported when using the native AS/400 database access classes:

- SQL_LONGVARCHAR
- SQL_LONGVARGRAPHIC
- DATE_STRUCT
- TIME_STRUCT
- TIMESTAMP_STRUCT

Note: Use a `char *` to represent date, time, or timestamp types.

Additional Related Information

You may only connect to the database (`db.connect(...)`) using the user ID under which the application is running.

The default for commitment control (SQL_ATTR_COMMIT) is SQL_COMMIT_CHG. When using the default, you need to ensure that all transactions (for DB2 for OS/400 this includes SELECT statements) are committed using either `IC4SQLEnvironment::commitDatabases()` or `IC4SQLDatabase::commit()`.

When binding parameters to a statement object for data that is provided at execution time, the *valueLength* argument, which must be set to SQL_DATA_AT_EXEC prior to statement execution, must point to a different variable for each of the `bindParameter(...)` invocations. The following example applies only to native OS/400 applications and not to client applications:

```
unsigned long retSize1; // for valueLength argument of 1st
                        // bindParameter(...) call.
unsigned long retSize2; // for valueLength argument of 2nd
                        // bindParameter(...) call.
unsigned long retSize3; // for valueLength argument of 3rd
                        // bindParameter(...) call.
SQLPOINTER parm1tok = new long(1);
SQLPOINTER parm2tok = new long(2);
SQLPOINTER parm3tok = new long(3);

short val = 1;
float val2 = 123.123;
char val3[] = "Example";

IC4SQLStatement stmt(myDatabase,
   "INSERT INTO IC4SQCT.TABLE2 (FIRST, SECONDFLD, SIXTH) VALUES(?,?,?)");

try
{
  // Specify data types and declare params as SQL_DATA_AT_EXEC
  stmt.bindParameter(1, SQL_C_SHORT, SQL_SMALLINT, 0, 0,
                     (long *)parm1tok, &retSize1);
  stmt.bindParameter(2, SQL_C_CHAR, SQL_CHAR, 50, 0,
                     (long *)parm2tok,
                     &retSize2);
  stmt.bindParameter(3, SQL_C_FLOAT, SQL_DOUBLE, 0, 0,
                     (long *)parm3tok,
                     &retSize3);
}
catch(IC4SQLException &excp)
{
  cout << "Failed on bindParameter\n";
}

retSize1 = SQL_DATA_AT_EXEC;
retSize2 = SQL_DATA_AT_EXEC;
retSize3 = SQL_DATA_AT_EXEC;

RETCODE rc = stmt.execute();
```

Using the Database Access Classes

The AS/400 does not support asynchronous processing of SQL statements. This means that SQL_STILL_EXECUTING is not returned from any member function. However, should asynchronous processing be supported in a future release, this chapter addresses the topic wherever it is applicable. The Access Class Library database access header files also list SQL_STILL_EXECUTING as a return code for those member functions where it would be supported.

Following are examples of how the database access classes can be used.

Using the Environment Class

The Environment class, IC4SQLEnvironment, manages the SQL environment. After you construct an IC4SQLEnvironment object, it must be referenced when constructing database objects.

Following is an example of committing all outstanding transactions for all database objects instantiated under a particular environment:

```
IC4SQLEnvironment env;  // Establish the SQL environment for this
                        // application.  NOTE: Only one environment
                        // may be instantiated per application.
IC4SQLDatabase db1(env);// For the native platform and the OS/2
                        // platform, there can only be one database
                        // object connected at a time.
db1.connect("myas400"); // Establish connection to a data source.

IC4SQLStatement myStmt(dbl, "INSERT INTO mylib.mytable values('ME', 1)");
                        // Instantiate and prepare a statement.
myStmt.execute();       // Execute the statement.

// Commit changes using the environment object
env.commitDatabases();  // All outstanding transactions for all
                        // database objects instantiated under this
                        // environment object will be committed.
```

Using the Database Class

The Database class, IC4SQLDatabase, manages a connection to an AS/400 system and provides catalog functions to query information about the AS/400 database. Following is an example of using a database object catalog function:

```
IC4SQLEnvironment    myEnvironment;             // Create Environment
IC4SQLDatabase       myDatabase(myEnvironment); // Create Database
IC4SQLResultSet      myResults;                 // Create Result Set
try
{
   myDatabase.connect("SystemA");               // Connect to AS/400
   unsigned short supported;
   myDatabase.functionSupported(SQL_API_SQLTABLES, &supported);
   if (supported)
   {
      char *qualifier = "";                     // Generate a result
      char *owner     = "MYSQLCOL";
      char *table     = "MYTABLE";
```

```
      char *tableType = "'TABLE' 'VIEW'";
      myDatabase.tables(myResults,
                        (unsigned char *)qualifier, strlen (qualifier),
                        (unsigned char *)owner, strlen(owner),
                        (unsigned char *)table, strlen(table),
                        (unsigned char *)tableType, strlen(tableType));
      RETCODE rc = myResults.fetch();           // Fetch first row
      while (rc == SQL_SUCCESS ||
             rc == SQL_SUCCESS_WITH_INFO)
      {
        cout << myResults << endl;              // Print the data
        rc = myResults.fetch();                 // Fetch next row
      }
   }
   else
      cout << "tables function not supported." << endl;
}
catch (IC4SQLException)
{
   cout << "Something failed" << endl;
}
myResults.close();
```

This example illustrates the following points:

1. **myDatabase.connect("SystemA");**

   The database object tries to establish a connection. For the OS/2, OS/2 Optimized, Windows 95, and Windows NT clients, the connection is made using the user ID and password of the client's primary connection. When running natively on the AS/400, the connection is made using the user ID and password under which the calling application is signed on. To specify a different user when running on the OS/2, OS/2 Optimized, Windows 95, or Windows NT client, you can optionally add *userid* and *password* parameters to the `connect()` member function.

2. **myDatabase.functionSupported(SQL_API_SQLTABLES, &supported);**

   This database member function returns a parameter to indicate whether the function designated by the constant value is supported by the call-level interface used by the database access classes.

3. **myDatabase.tables**

   This is a catalog function, meaning that it returns a result set with information from the catalog of the AS/400 database. The result set is associated with the object `myResults`.

   Note: All catalog functions would be coded in a similar fashion.

In the normal case where your application needs to access only one database source, the example above is all you need. If your application needs to switch among databases of different AS/400 systems, you can end the connection to one AS/400 system and connect to another AS/400 system as follows:

```
myDatabase.disconnect();
myDatabase.connect("AS400B", "MYUSERID", "MYPASSWORD");
```

Alternatively, if you are using the Optimized OS/2 client, Windows 95 client, or Windows NT client, you can create and connect with another IC4SQLDatabase object. If you are using the OS/2 client, you must first disconnect the original database connection, as in the example, because the OS/2 client supports only one connection at a time.

Note: If a transaction is open, the disconnect operation fails. To end transactions on this database, you must commit or rollback the transaction before you disconnect.

Using the Statement Class

The SQL Statement class, IC4SQLStatement, provides the ability to set parameters for SQL statements, prepare and execute the statements, and set or retrieve attributes.

SELECT Statement

If the text of the statement is a SELECT statement, then, when executing the statement, you must use the version of execute() or executeDirect() with a parameter for specifying a result set object, IC4SQLResultSet. Otherwise, an exception is thrown.

If the text of the SQL statement is not a SELECT statement, then the version of execute() or executeDirect() without a result set parameter should be used, although the version with a result set parameter is allowed. In that case, after the statement is executed, the result set is closed.

Executing a Statement Directly

Statements can be executed directly by passing an SQL text string as a parameter on the executeDirect() member function. When a statement is executed directly, the statement object is not left in a prepared state. You cannot subsequently do an execute() without first doing a prepare(). Following is an example of executing a statement directly:

```
IC4SQLStatement      myStatement(myDatabase);
IC4SQLResultSet      myResults;
myStatement.executeDirect(myResults, "select * from mylib.myfile");
              // Note: Must provide a result set object as parameter
              //       because a SELECT statement is being executed.
```

Binding Parameter Markers to User Variables

You can bind parameter markers in a statement text to user variables. Parameter markers in the statement text are designated by question marks. At the time the statement is executed, the current value in the bound user variable is taken as the parameter value. This applies both to doing an executeDirect() and to doing an execute() after doing a prepare(). Parameters can be bound before or after the statement is prepared. Parameter markers can be bound again after executing the statement, or the same variables can remain bound and have different values put into them for the next execute(). Binding a parameter marker again overrides the original parameter binding. Following is an example of how you could set a parameter marker:

```
IC4SQLStatement  myStatement(myDatabase);
IString  address = "1313 Mockingbird Lane";
myStatement.bindParameter(1,                  // Parameter number
                          SQL_C_CHAR,         // Type of user variable
                          SQL_CHAR,           // SQL type to convert to
                          address.size(),     // Max size of parameter
                          0,                  // Scale (ignored)
                          (char *)address,    // User variable
                          NULL);              // Null-terminated
myStatement.executeDirect("insert into paylib.list values('Your Name',?,5000)");
```

For each SQL C datatype, an overloaded `bindParameter()` member function is provided with default parameters. To make the above function call a little simpler, you could use the following alternative to set a parameter marker:

```
myStatement.bindParameter(1, address.size(), address);
```

When using the Optimized OS/2 client, Windows 95 client, Windows NT client, or AS/400 server, you can set the *valueLength* parameter (the last parameter) on `bindParameter()` to point to special values that have the following meanings:

SQL_NULL_DATA	A null parameter is provided.
SQL_DATA_AT_EXEC	The `execute` command prompts for the data.

Preparing and Executing a Statement

A statement is prepared by using `prepare()` and is then executed by using `execute()`. The advantage of preparing a statement object is that it can then be executed multiple times (usually with different parameter values) without having to pay the cost of parsing the statement text and setting up the AS/400 structures each time. This is a performance advantage. Following is an example of preparing and executing a statement with a parameter:

```
IC4SQLStatement myStatement(myDatabase);
myStatement.prepare("insert into payroll.list values('My Name', ?, 5000)");
char  address[40] = "123 Mystreet, Rochester, MN";
      // Note:  We do not use an IString variable here because
      // we will be changing the value of address for multiple executions
      // of the statement.  Because the variable is bound by its address,
      // an IString cannot be used because its internal address for
      // storing its contents may change when the contents are changed.
myStatement.bindParameter(1,sizeof(address), address); // Bind variable
                                                       // "address" to first
                                                       // parameter marker.
myStatement.execute();
strcpy(address, "456 Yourstreet, Chicago IL");         // Change parameter.
myStatement.execute();                                 // Execute with new
                                                       // parameter value.
```

You could also prepare the statement on the constructor as follows:

```
IC4SQLStatement myStatement(myDatabase, "insert into payroll.list values
('My Name',?,5000)");
```

Freeing Resources

When you are done with a prepared statement object and do not need it again, it is best to free the AS/400 resources. This happens automatically if the statement object is deleted or goes out of scope. However, if your application wants to explicitly free the resources until some later time when the statement is used again, you can free the resources by using `unprepare()`. The resources of open result set objects can be freed by doing a `close()` on the result set object. Following is an example of freeing resources:

```
myStatement.prepare("select * from mylib.myfile");
myStatement.execute(myResults);
              .
              .       (process results)
              .
myResults.close();           // Free result set resources
myStatement.unprepare();     // Free statement resources
```

Using the Result Set Class

The Result Set class, `IC4SQLResultSet`, manages and provides access to the result sets generated by SQL SELECT statements and catalog functions. After a result set is used on a successful catalog function, `execute()`, or `executeDirect()`, the result set can be used to do the following:

- Bind columns of the result set to user variables
- Fetch rows
- Get data from unbound columns of a fetched row
- Format a row of data into a string
- Retrieve attributes of the result set

The result set object can be closed at any time by calling `close()`. Any attempt to use a result set object that has been closed, or that has not been successfully used to return the results of an operation, causes an exception to be thrown.

The results of an `execute()`, `executeDirect()`, or catalog function are available through the specified result set object only when the function returns `SQL_SUCCESS` or `SQL_SUCCESS_WITH INFO`. If `SQL_NEED_DATA` is returned, the application must use `parameterData()` and `putData()` to supply parameter data until a successful return code from `parameterData()` is received.

Performance Consideration

After the result set is returned, the statement object and result set object are independent. This means that you could do another `select()` using the statement object to another result set object, even while the first result set object is still open. You could then proceed to fetch rows from either of the result set objects.

However, there is one performance consideration: If you do another `execute()` against the statement object, it is processed faster if the first result set object into which the results were put is closed before the next `execute()` is done. If the result set is closed before doing the next `execute()`, the resources of the originally prepared statement can be used again. If the `execute()` is done before closing the result set object, the cost of

preparing the statement is paid again, although in either case the `execute()` produces the same results.

Binding Columns

User variables can be bound to particular columns in the result set. When a `fetch()` is done, the values from the corresponding columns are set into your variables. If the conversion is supported, the values are converted to the type of variable you provide. The order in which columns are bound is irrelevant, as shown in the following example:

```
IC4SQLStatement myStatement(myDatabase);
IC4SQLResultSet myResultSet;
myStatement.executeDirect(myResultSet, "select * from mylib.names");
char empName[100];
short age;
long  shoesize;
float battingavg;
myResultSet.bindColumn(1, age);                     // Bind column 1
myResultSet.bindColumn(3, battingavg);              // Bind column 3
myResultSet.bindColumn(2, shoesize);                // Bind column 2
myResultSet.bindColumn(7, empName, sizeof(empName)); // Bind column 7
myResultSet.fetch();         // Fetch first row into bound variables
```

Bytes Available

The default on `bindColumns()` is to pass NULL as the *bytesAvailable* parameter. This causes an exception to be thrown when the `fetch()` is done if the value in that column is NULL. To prevent this, you can pass in a variable for *bytesAvailable*, as shown in the following example:

```
short age;
long  shoesize;
unsigned long rz1;
unsigned long rz2;
myResultSet.bindColumn(1, age, &rz1);
myResultSet.bindColumn(2, shoesize, &rz2);
```

Column Attributes

You can get the column attributes by using either of the following methods:

`columnAttribute()`	Gets one attribute for a column, and can also be used to determine the number of columns.
`describeColumn()`	Returns the result descriptor information for a column (a default set of the most commonly used attributes of a column).

Information is retrieved for the column index you provide. If the index is out of range, an exception is thrown.

Fetch and Get Data

`fetch()` causes the values for bound columns in a row of data to be retrieved and set into the application variables that were specified on the `bindColumn()`. The next `fetch()` retrieves the next row of data.

Data from columns that were not bound can be retrieved using `getData()`, as shown in the following example:

```
myStatement.executeDirect(myResultSet, "select * from mylib.names");
short weight;
myResultSet.bindColumn(4, weight);
myResultSet.fetch();       // Fetch data from column 4 into bound variable
if (weight > 300)
{
   double height;
   myResultSet.getData(7, height);     // Get data from column 7
   if (height < 5.5)
     cout << "Time for a diet \n";
}
myResultSet.close();
```

Closing Result Sets

When using the OS/2 client, the sum of all prepared statement objects and open result set objects cannot exceed 10. Therefore, it is advisable to close result set objects when you are done using them.

Once the last row of the result set has been retrieved, the result set should be closed.

If a statement object executes a SELECT statement and specifies an open result set object for the parameter, the open result set object is first closed and then reopened with the results of the new SELECT statement. If the implicit close of the result set object fails for any reason, the `execute()` of the SELECT statement returns an exception.

Using the Variable and Variable List Classes

The Variable class, `IC4SQLVariable`, provides a structure for storing the information pertaining to a user variable being used as a parameter in a statement or for receiving the data of a bound column. Both `bindColumn()` and `bindParameter()` can take a variable object as a parameter.

The Variable List class, `IC4SQLVariableList`, is a collection of variable objects. Both `bindColumns()` and `bindParameters()` can take a variable list object as a parameter. If a variable list object is provided, all variables in the list are bound to columns or parameters. Which column or parameter a variable is bound to is determined by the position number data member of the variable object.

Convenience operators are provided to allow “streaming” syntax. A variable list object of bound column information can be created and provided to the `bindColumns()` member function, as shown in the following example:

```
char name[20];
short age;
long  shoesize;
float battingavg;
myResultSet.bindColumns(IC4SQLVariableList()
                                    << IC4SQLVariable(name, sizeof(name))
                                    << age
                                    << shoesize
                                    << battingavg);
// The order of the variables determines which column is bound.
```

In the above example, most of the user variables being bound are simply provided as input to the << operator. The C++ compiler automatically converts them to `IC4SQLVariable` types, and then they are added to the `IC4SQLVariableList` object being created, which in turn is the parameter to `bindColumns()`.

The exception to this is the character variable `name`. Because `bindColumns()` requires the maximum length of the bound variable as input, creating an `IC4SQLVariable` object from `name` must be done explicitly so that the size can be provided. For the other (numeric) variables, the constructor for creating the variable objects defaults the size to the (known) size of the variable type.

Because `bindParameter()` can accept null-terminated strings without a specified length as parameters, the above requirement with character variables does not apply when doing `bindParameter()` with a variable or variable list object as parameter.

Using the Row Class

The Row class, `IC4SQLRow`, provides an easy way to fetch an entire row and save the results. The benefits of using row objects are that you do not have to create application variables to contain the fetched data or to explicitly bind each column. Using the `IC4SQLRow` class, you can:

- Print a row
- Format a row as a string
- Get the data of a specified column number, either in the format in which it was stored or converted to string format

Following is an example of fetching into a row object:

```
IC4SQLRow row;
myResultSet.bindToRow(row);
while(myResultSet.fetch() == SQL_SUCCESS)
{
  cout << row.columnDataAsString(3) << endl;
}
myResultSet.close();
```

A Simple Query Program

The following code segment illustrates how the environment, database, statement, and result set objects work together. The code processes a SELECT statement and displays the results, one row per line, to standard output.

```
IC4SQLEnvironment    myEnvironment;             // Create Environment
IC4SQLDatabase       myDB(myEnvironment);       // Create Database
IC4SQLResultSet      myResults;                 // Create Result Set

try
{
   myDB.connect("SystemA");                     // Connect to AS/400

   IC4SQLStatement myStatement(myDB);
   myStatement.executeDirect(myResults, "select * from mylib.myfile");
   RETCODE rc = myResults.fetch();              // Fetch first row
   while (rc == SQL_SUCCESS ||
          rc == SQL_SUCCESS_WITH_INFO)
   {
      cout << myResults << endl;                // Print the data
      rc = myResults.fetch();                   // Fetch next row
   }
}
catch (IC4SQLException)
{
   cout << "Something failed" << endl;
}
myResults.close();
```

Note the following points:

1. `myDB.connect("SystemA");`

   The database object tries to establish a connection. For the OS/2, OS/2 Optimized, Windows 95, and Windows NT clients, the connection is made using the user ID and password of the client's primary connection. When running natively on the AS/400, the connection is made using the user ID and password under which the calling application is signed on. To specify a different user when running on the OS/2, OS/2 Optimized, Windows 95, or Windows NT client, you can optionally add *userid* and *password* parameters to the `connect()` member function.

2. `IC4SQLStatement myStatement(myDB);`

   This constructs a statement object. When a statement object is constructed, it must be associated with a connected database object. The parameter specifies the `IC4SQLDatabase` object with which the statement will be associated.

3. `myStatement.executeDirect(myResults, "select * from mylib.myfile");`

   This directly executes the supplied statement text and associates the result set object `myResults` with the generated results.

4. `while (rc == SQL_SUCCESS...)`

   Keep fetching data until there is no more.

5. `cout << myResults << endl;`

   This outputs all the columns of data from the last `fetch()` and writes them to an output stream. Any column data that is not a character type is converted to string format prior to output.

6. `try { ... } catch (IC4SQLException) { ... }`

Using the Database Access Classes

If an error occurs in one of the member functions inside the brackets of the `try` clause, an `IC4SQLException` exception is thrown. In that case, control passes to the code in the `catch` clause, where you can take appropriate action. The exception object that was thrown contains more information about which member function failed and why it failed.

Chapter 5. Data Queue Access Classes

A **data queue** is an AS/400 object used for sharing data between programs. The object-type identifier for data queue is *DTAQ. The Access Class Library data queue classes provide a VisualAge for C++ class interface to help build applications that use these AS/400 objects.

The interface supports two types of data queues. Data queues that are accessed using a key are referred to as keyed data queues. Data queues that are accessed sequentially are referred to as nonkeyed data queues. These two types of data queues have important differences. Keyed data queues require the specification of a key when performing operations against the data queue. Entries are retrieved from a keyed data queue by key. If more than one entry has the same key, the entries are retrieved last-in first-out (LIFO). Data is retrieved from a nonkeyed data queue either first-in first-out (FIFO) or LIFO. However, both types also have many features in common, such as destroying and opening a data queue.

Data queue access classes are supported on the following:

- OS/2
- OS/2 Optimized
- Windows 95
- Windows NT
- AS/400

When compiling for the OS/2 client, you must define the preprocessor variable `__IC4PCS__`. When compiling for the OS/400 server, you must define the preprocessor variable `__IC40S400__`.

Introduction to the Data Queue Access Classes

Following are the three classes that provide the client or OS/400 interfaces:

Base Data Queue	Provides a common abstraction for both types of data queues. The Base Data Queue class, `IC4BaseDataQueue`, provides a common interface to both keyed and nonkeyed data queues. For all data queues, you can do operations such as the following: • Write an entry to the queue • Read an entry from the queue • Clear entries from the queue • Retrieve the name of the queue • Retrieve the text description for the queue
Data Queue	Provides the extended interface to a nonkeyed data queue, such as creating a nonkeyed data queue. The Data Queue class, `IC4DataQueue`, provides

	extensions for nonkeyed data queues. A member function to create a nonkeyed data queue is defined.
Keyed Data Queue	Provides the extended interfaces to a keyed data queue, such as creating a keyed data queue, writing to the data queue with a key, and reading from the data queue with a key. The Keyed Data Queue class, `IC4KeyedDataQueue`, provides extensions for keyed data queues. Member functions to do the following are defined: • Create a keyed data queue • Set and retrieve the key value • Set and retrieve the key comparison value • Read entries using key and key comparison values • Write entries using key values • Clear entries using key values
Data Queue Exception Class	The Data Queue Exception class, `IC4DataQueueException`, is derived from IC4Exception. All exceptions that are thrown by the data queue classes are instances of `IC4DataQueueException`.

Using the Data Queue Access Classes

Following are examples of how the data queue classes can be used.

Writing Data to a Data Queue

Writing (sending) data to a data queue is quite simple.

```
IC4DataQueue mydq("/QSYS.LIB/myLibrary.LIB/myQueue.DTAQ", "SystemA");
mydq.open();
mydq.write("test data");
mydq.close();
```

When writing to keyed data queues, you need to specify a key. You can do so in several ways:

- Specifying the *key* parameter on the constructor.
- Using the `setKey()` member function.
- Specifying the *key* parameter on the `write()` member function.

  Note: Unlike the `setKey()` member function, the *key* parameter on the `write()` member function is not persistent. The *key* parameter on the `write()` member function is used for only a single function call. Using the *key* parameter on the constructor is equivalent to using the `setKey()` member function.

Following is an example of writing data to a keyed data queue by using the *key* parameter on the constructor:

```
IC4KeyedDataQueue mydq("/QSYS.LIB/myLibrary.LIB/myKeyQ.DTAQ", "SystemA",
                       "KeyValue");
mydq.open();
mydq.write("keyed test data");
mydq.close();
```

Following is an example of writing data to a keyed data queue by using the `setKey()` member function:

```
IC4KeyedDataQueue mydq("/QSYS.LIB/myLibrary.LIB/myKeyQ.DTAQ", "SystemA");
mydq.open();
mydq.setKey("KeyValue1");
mydq.write("keyed test data 1");
mydq.setKey("KeyValue2");
mydq.write("keyed test data 2");
mydq.close();
```

Following is an example of writing data to a keyed data queue by specifying the *key* parameter on the `write()` member function:

```
IC4KeyedDataQueue mydq("/QSYS.LIB/myLibrary.LIB/myKeyQ.DTAQ", "SystemA");
mydq.open();
mydq.write("KeyValue1", "keyed test data 1");
mydq.write("KeyValue2", "keyed test data 2");
mydq.close();
```

Reading Data from a Data Queue

Reading (receiving) data from a data queue is also quite simple. Following is an example of reading data from a nonkeyed data queue:

```
IC4DataQueue mydq("/QSYS.LIB/myLibrary.LIB/myQueue.DTAQ", "SystemA");
IString data;

mydq.open();
mydq.read(data);
cout << data << endl;
mydq.close();
```

When reading from keyed data queues, you need to specify a key and search type. You can do so in several ways:

- Using the *key* parameter on the constructor.
- Using the `setKey()` and `setSearchType()` member functions.
- Specifying the *key* and *searchType* parameters on the `read()` member function.
- A combination of the above (for example, using the `setSearchType()` member function and specifying the *key* parameter on the `read()` member function).

  Note: Unlike the `setKey()` and `setSearchType()` member functions, the *key* and *searchType* parameters on the `read()` member function are not persistent. The *key* and *searchType* parameters on the `read()` member function are used for only a single function call.

Following is an example of reading data from a keyed data queue by using `set` member functions:

```
IC4KeyedDataQueue mydq("/QSYS.LIB/myLibrary.LIB/myKeyQ.DTAQ", "SystemA");
IString data;

mydq.open();
mydq.setKey("KeyValue1");
mydq.setSearchType(IC4KeyedDataQueue::Search_LE);

mydq.read(data);
cout << data << endl;

mydq.setKey("KeyValue2");
mydq.read(data);
cout << data << endl;

mydq.close();
```

Following is an example of reading data from a keyed data queue by using the *key* and *searchType* parameters on the `read()` member function:

```
IC4KeyedDataQueue mydq("/QSYS.LIB/myLibrary.LIB/myKeyQ.DTAQ", "SystemA");
IString data;

mydq.open();

mydq.read("KeyValue1", data, IC4KeyedDataQueue::Search_LE);
cout << data << endl;

mydq.read("KeyValue2", data, IC4KeyedDataQueue::Search_EQ);
cout << data << endl;

mydq.close();
```

Asynchronous Read

You can request that data be read without waiting for the data to be returned. The data can be accessed at a later time. This is useful when the operating system that you are using does not support threads, and you want the application to continue without waiting for the read operation to complete. Following is an example of an asynchronous read:

```
IC4DataQueue mydq("/QSYS.LIB/myLibrary.LIB/myQueue.DTAQ", "SystemA");
char myChar[100];
IC4BaseDataQueue::asyncHandle readHandle;
unsigned long length;

mydq.open();

readHandle = mydq.asyncRead(myChar, 100, &length);
...
mydq.checkData(readHandle);
if (length == 0)
  cout << "No data available" << endl;
else
  cout << "Data = " << myChar << endl;

mydq.close();
```

Note: Asynchronous read is not available for native AS/400 applications using Access Class Library.

Creating and Deleting Data Queues

To create and delete data queues on an AS/400 system, member functions `create()` and `destroy()` are used. The `create()` member function does an implicit `open()` function call, and the `destroy()` member function does an implicit `close()` function call. Following is an example of creating and deleting a nonkeyed data queue:

```
IC4DataQueue mydq("/QSYS.LIB/myLibrary.LIB/newQueue.DTAQ", "SystemA");
mydq.create(50, IC4BaseDataQueue::Order_FIFO,
       IC4BaseDataQueue::Force_No, "*LIBCRTAUT",
       False, "test data queue");
...
mydq.destroy();
```

The above example specifies all parameters. The following example uses the minimum number of parameters to create a keyed data queue:

```
IC4KeyedDataQueue mydq("/QSYS.LIB/myLibrary.LIB/newKeyedQ.DTAQ", "SystemA");
mydq.create();
...
mydq.destroy();
```

Using the Base Data Queue Class

Using an abstract base class, behavior can be encapsulated and common interfaces provided for nonkeyed and keyed data queues, such as `write()` and `read()`, although the underlying programming interfaces are different.

A base level of interfaces is provided in the class `IC4BaseDataQueue` that encompasses both nonkeyed and keyed data queues. This base level includes simple write and read interfaces for both types of objects. Normally on a keyed data queue you would have to specify a key on a `write()` and a key and search type on a `read()`. However, you can specify these values (key and search type) on the keyed data queue object and then use the abstract interface of `IC4BaseDataQueue` for more general processing. You do not have to specify the key for each request.

Using the Data Queue Access Classes

Following is an example of reading information from both a nonkeyed and a keyed data queue:

```
// Common routine for reading from both keyed and nonkeyed data queues
void readFromDataQueue(IC4BaseDataQueue &mydq)
{
  IString data;

  cout << "...Checking for data..." << endl;
  mydq.open()
  mydq.read(data);
  if (mydq.dataLength())
    cout << "  Data = " << data << endl;
  else
    cout << "  No data on queue" << endl;
}

main()
{
  IString type;
  cout << "Read from keyed (K) or non-keyed?(N)" << endl;
  cin >> type;
  if (type.upperCase() == "K")
  {
    IC4KeyedDataQueue mydq("/QSYS.LIB/MYLIB.LIB/MYKEYDQ.DTAQ");
    mydq.setKey("12345678");
    mydq.setSearchType(IC4KeyedDataQueue::Search_EQ);
    readFromDataQueue(mydq);
  }
  else
  {
  IC4DataQueue mydq("/QSYS.LIB/MYLIB.LIB/MYDQ.DTAQ");
  readFromDataQueue(mydq);
  }
}
```

In the above example, `readFromDataQueue()` does a generic read. `main()` does the extra processing and set up for keyed data queues to work properly, but leaves generic processing to `readFromDataQueue()`.

Chapter 6. Data Translation Access Class

The data translation class provides several member functions to transform data between the AS/400 and PC formats. For example, these data translation functions convert:

- ASCII character string representations of numbers into EBCDIC machine representations of numbers and vice versa
- ASCII character strings into EBCDIC characters strings and vice versa
- ASCII representations of hexadecimal numbers into their binary form and vice versa

Data translation access classes are supported on the following:

- OS/2
- OS/2 Optimized
- Windows 95
- Windows NT

When compiling for the OS/2 client, you must define the preprocessor variable `__IC4PCS__`.

Introduction to the Data Translation Access Class

Data Translation
: The member functions of the data translation class, `IC4Translate`, are static so that a program can use the member functions without an object of the class being constructed.

Data Translation Exception Class
: The Data Translation Exception class, `IC4TranslateException`, is derived from IC4Exception. All exceptions thrown by the data translation classes are instances of `IC4TranslateException`.

Using the Data Translation Access Class

Following are examples of how the data translation access classes can be used.

Converting an ASCII Numeric to an EBCDIC Binary

The following example uses the member function `a2bin2()` to convert an ASCII numeric string to a two-byte EBCDIC binary number, and then uses the member function `hex2a()` to print out the result.

```
#include <ic4xlt.hpp>
#include <iostream.h>

void main ()
{
```

```
char *ASCIINumeric="20";         // The number to convert.
char binary2[2];                 // The receiver for the translated
                                 // value.

cout << "Converting an ASCII numeric string to binary(2):" << endl;
cout << "ASCIINumeric = " << ASCIINumeric << endl;

IC4Translate::a2bin2(binary2,ASCIINumeric);

cout << "binary2      = 0x"
     << IC4Translate::hex2a(binary2,2)
     << endl;
}
```

The output for the above program looks like the following:

```
Converting an ASCII numeric string to binary(2):
ASCIINumeric = 20
binary2      = 0x0014
```

Converting an ASCII Numeric to an EBCDIC Packed Decimal

The following program converts an ASCII numeric string to an EBCDIC packed decimal. It uses the member function `a2pak()` to do the conversion, and then uses the member function `hex2a()` to print out the result.

```
#include <ic4xlt.hpp>
#include <iostream.h>

void main ()
{
char *ASCIINumeric="-20.002";      // The number to convert.
const int DecimalDigits=3;         // Number of decimal digits in
                                   // the ASCIINumeric.
const int PackedLength=3;          // Number of bytes of packed
                                   // decimal data to return.

char packedDecimal[PackedLength];  // Receiver for translated value.

cout << "Converting an ASCII numeric string to packed decimal:"
     << endl;
cout << "ASCIINumeric   = " << ASCIINumeric << endl;

IC4Translate::a2pak(packedDecimal,ASCIINumeric,
                    PackedLength,DecimalDigits);

cout << "Packed Decimal = 0x"
     << IC4Translate::hex2a(packedDecimal,PackedLength)
     << endl;
}
```

The output for the above program looks like the following:

```
Converting an ASCII numeric string to packed decimal:
ASCIINumeric   = -20.002
Packed Decimal = 0x20002D
```

Converting an ASCII Character to an EBCDIC Character

For OS/2 Optimized, Windows 95, and Windows NT, a system name may be specified when calling the `a2e()` or `e2a()` functions. A connection to the specified system must be started before the function is called. If a system name is not specified, a connection to the Client Access default system must be started before the function is called. The connection is used to determine the AS/400 EBCDIC code page for the conversion.

Translating from ASCII to EBCDIC can be easily done. The following example shows how you would translate from ASCII to EBCDIC using the code page of a specific AS/400 system. If you want to use the code page of the Client Access default system, use one of the other `a2e()` methods that do not specify a system name.

```
#include <ic4xlt.hpp>
#include <iostream.h>

void main()
{
  char *system = "myas400";                 // System whose code page will be used
                                            // when translating from ASCII to
                                            // EBCDIC
                                            // NOTE: A connection to the system
                                            // must exist when a2e() is called
  IString theSource = "A string to convert"; //String to convert
  char theTarget[20];                       //Receiver to hold the translated string

  IC4Translate::a2e(theTarget, (char *)theSource, system,
                    sizeof(theTarget), theSource.length());
}
```

Using the Data Translation Access Class

Chapter 7. Data Type Access Classes

The Access Class Library data type access class group provides a VisualAge for C++ class interface for applications that need to manipulate data in AS/400 system formats. Data type access classes are supported on the following:

- OS/2 Optimized
- Windows 95
- Windows NT

Introduction to the Data Type Access Classes

The data type class group provides a set of classes that make it easier for a client application to work with data from an AS/400 system. This class group supports the transformation of both character and numeric data to a format that can be used by a VisualAge C++ program running on a PC. Because programs have widely varying requirements for access to the data that is managed by the data type class group, the classes have been designed to provide a very flexible interface.

The data types classes are designed to manage both an AS/400 view and a PC view of the data. An instance of one of the data type classes, called a data type object, can store two representations of the data:

External format	The data in a form expected by the AS/400 system
Local format	The data in a form that can be easily manipulated by a C++ program on the PC

Conversion member functions, `translateFromExternal()` and `translateToExternal()`, are provided to convert between the formats. Your program can control when the data conversion is done.

The storage that contains the data representations can be assigned and managed by a data type object for ease of programming, or the storage can be assigned and managed by the program for greatest control over where the data resides. A program can choose one of these options or any combination of the two options. Storage that is managed by the data type object is called **object-managed** storage. Storage that is managed independently by the program is called **user-managed** storage.

Character data from an AS/400 can be complicated to work with. When character data is converted between AS/400 and PC formats, the correct character set code pages must be used during the conversion. The data types classes that work with character data allow the specification of code page values so that the data conversion can be done correctly.

The following sections provide a more detailed look at the classes that make up the data type class group.

Introduction to the Data Type Access Classes

Abstract Base Classes

The following abstract base classes provide common abstractions for the derived data type classes. In the following sections, the term data type refers to an instance of a class that is derived from `IC4DataType`.

- Data type

  `IC4DataType` is the base class for all data type classes. It declares the member functions that are common to all data type classes.

- Code-page-dependent data type

  `IC4CPDDataType` is a base class for those data type classes that need national language code page conversion capabilities. It is derived from `IC4DataType`. `IC4Char`, `IC4Array`, and `IC4Struct` are derived from this class.

- Base array data type

  `IC4ArrayBase` is a base class supporting the `IC4Array` template class. It is derived from `IC4CPDDataType`. It implements the type-independent member functions for the `IC4Array` template class.

Derived Data Types

The data type access class group includes several classes that are derived from one of the abstract base classes. The classes are discussed in "Basic Data Types" and "Aggregate Data Types" on page 49.

An instance of a data type class is called a **data item**. Every data item has **external format data**, and every data item (except `IC4Byte` in some cases) also has **local format data**.

The derived data type classes can be further defined as basic data types and aggregate data types.

Basic Data Types

The data type access class group includes several classes, called the basic data type classes, that represent the basic types of AS/400 system data.

Because the local format data is stored as a standard C++ data type, an instance of one of the basic data types can be used as a variable of its C++ data type. The C++ data type associated with each basic data type class is listed in the following table:

Table 7 (Page 1 of 2). Data Type Access Class Group: Basic Data Type Classes

AS/400 Data Type	Class	C++ Data Type
Binary 2	`IC4Bin2`	`short int`
Binary 4	`IC4Bin4`	`long int`
Unsigned binary 2	`IC4UnsignedBin2`	`unsigned short int`
Unsigned binary 4	`IC4UnsignedBin4`	`unsigned long int`
Short floating point	`IC4Float4`	`float`

Table 7 (Page 2 of 2). Data Type Access Class Group: Basic Data Type Classes		
AS/400 Data Type	**Class**	**C++ Data Type**
Long floating point	`IC4Float8`	`double`
Character string	`IC4Char`	`char*`
Byte string	`IC4Byte`	`void*`

Aggregate Data Types

The data type access class group includes two classes that represent aggregates of the basic types:

- Array

  `IC4Array` represents an array of data items all of the same data type. The data items in an array are called **elements** of the array.

- Structure

  `IC4Struct` represents a structure of data items that may be of various data types. The data items in a structure are called **members** of the structure.

A data item that is an element of an array or a member of a structure is said to be **contained** in the array or structure.

The conversion member functions, `translateFromExternal()` and `translateToExternal()`, of an aggregate data item call the conversion member functions of the data items that they contain.

Data Type Exception Class

The Data Type Exception class, `IC4DataTypeException`, is derived from `IC4Exception`. All exceptions that are thrown by the data type class group are instances of `IC4DataTypeException`.

Using the Data Type Access Classes

Following are examples of how the data type access classes can be used.

Constructing Data Items

All data type classes have a default constructor and a copy constructor.

The data type classes also have forms of the constructor that allow you to assign specific storage locations for the local values, external values, or both values of the data item. Each class also has member functions for assigning specific storage locations. For more information on assigning specific storage locations, see “Assigning Specific Storage Locations for Data Item Values” on page 51.

The basic data type classes have forms of the constructor that allow the constructed data item to be given an initial value.

Using the Data Type Access Classes

Assigning Values to Data Items

All data type classes (except `IC4Struct`) have an assignment operator for assigning an item the value of another data item of the same type. This assignment operator assigns the local and external format data the values of another data item's local and external format data. In general, operations or member functions that affect the value of a data item affect only the value of local format data. The exceptions to this general rule are the assignment operator and the `translateToExternal()` function, which is discussed in "Converting the Value Between Local and External Formats."

A data type may define other assignment operators too. Some data type classes define additional assignment operators. For example, `IC4Char` defines an assignment operator that assigns an `IC4Char` object the value of a null-terminated character array:

```
IC4Char& operator=(const char* value);
```

These assignment operators affect only the local format data and not the external format data.

Some data type classes have special member functions for setting the value of a data item. For example, instead of an overloaded assignment operator, `IC4Byte` defines the member function `setDataValue()` for assigning the data item the value of a block of storage.

Using a Data Item As a Variable of Its Local Format Data Type

The data type classes allow you to work with AS/400 data as if it were actually in the format that the PC expects. A data item can be used as if it were a variable of its local format data type. For example, `IC4Bin4` has a local format data type of `long int` and, therefore, can be used wherever a `long int` variable or value is called for. This includes using the data item in expressions, as the left-hand side of an assignment statement, and as an argument to a member function taking any long int form (including a `long int&`).

Using a data item as if it were a variable of its local format data type only affects the local format data. No automatic conversion occurs between the local and external data format values. This means that if you have set the local format data value and you want to get the associated external format data value, you must explicitly request conversion to the external format. See "Converting the Value Between Local and External Formats."

Converting the Value Between Local and External Formats

All data items have the conversion member functions `translateFromExternal()` and `translateToExternal()`. The `translateFromExternal()` member function converts from the external format to the local format. The `translateToExternal()` member function converts to the external format from the local format.

Accessing Local and External Format Data Values

The local data value is easily accessible by treating the data item as a variable of its local format data type. All of the basic data types provide conversion operators for that purpose. The local format data is also accessible using the `dataPointer()` member function. The `dataPointer()` function returns a `void*`. The number of bytes pointed to by the `dataPointer()` member function is returned by the `dataSize()` member function.

The external format data is accessible using the `externalDataPointer()` member function. The `externalDataPointer()` member function returns a `void*`. The number of bytes pointed to by the `externalDataPointer()` member function is returned by the `externalDataSize()` member function. No conversion member functions or operators exist for manipulating the external format data value directly. The intent of the data type classes is to provide an alternative format (that is, the local format) to manipulate and to provide a conversion member function to affect the external format data value. You can directly set the external format data value in either of the following ways:

- Directly manipulating the storage pointed to by `externalDataPointer()`;
- Assigning a specific storage location for external format data (see "Assigning Specific Storage Locations for Data Item Values").

Assigning Specific Storage Locations for Data Item Values

You can assign specific storage locations for the local and external format data values of a data item. For example, assigning specific storage can be useful when you receive a block of storage that contains the external format data for a data item. Instead of constructing a data item and copying the value into object-managed storage, you can use this block of storage as the external format data storage. You can assign specific storage locations in either of two ways:

- By assigning storage locations at construction
- By assigning storage using the `setDataPointer()` and `setExternalDataPointer()` member functions

If you do not assign storage locations, the data item implicitly assigns and manages its own storage locations.

When you assign storage for a data item, the data item does **not** assume ownership of the storage. Your program is responsible for managing the assigned storage. The storage must remain allocated for the life of the data item or until the storage is unassigned or assigned to a different location.

Note that all data type classes provide overloaded forms of the `setDataPointer()` and `setExternalDataPointer()` member functions. For example, `IC4Float8` has the following form of the `setDataPointer()` member function:

```
void setDataPointer(double * dataPointer);
```

whereas `IC4Char` has the following form:

```
void setDataPointer(char * dataPointer, unsigned long dataSize);
```

Using the Data Type Access Classes

See the *IBM Access Class Library for Windows Reference*, SC41-4622, for the overloaded forms of the member functions supported by each data type class.

When storage is assigned, the assigned storage location is assumed to hold the value for the data item. Any previous value the data item had is lost.

Defining Structures of Data Items

Structures of data items are supported by the `IC4Struct` class. A data item in a structure is called a member of the structure. You need to use the `IC4Struct` class instead of a C++ `struct` if you want the members of the structure to reside in contiguous storage. This is because data items manage their storage independently and each has two values (the local format data value and the external format data value). By using `IC4Struct`, you are guaranteeing that the local and external format data of all structure members are in contiguous storage.

The local and external format data of `IC4Struct` members are packed to byte boundaries. If you are mapping an `IC4Struct` over a C++ `struct`, you need to be aware of this padding and do explicit padding of the `IC4Struct` data item if necessary.

Unlike a C++ `struct`, structures of data items are built at run time.

The local format data and the external format data of a structure of data items can be thought of as a C++ `struct` of the local format data of the structure's members. From here on, a structure of data items will be called a structure.

Adding Members to an IC4Struct Data Item

Members are added to a structure using the `IC4Struct addMember()` member function or insertion operator (`operator<<`). Following is an example.

```
IC4Bin4 item1(1);                     // Construct member 1
IC4Bin4 item2(2);                     // Construct member 2
IC4Char item3("Item 3",6);            // Construct member 3
IC4Struct structure;                  // Construct structure
structure << item1 << item2 << item3; // Add the members
```

A data item must have an external data size greater than zero to be added to a structure. That is why an external data size is specified on the construction of the `IC4Char` data item added as the third member of the structure in the above example.

The local format data of the structure in the previous example has a form equivalent to the following C++ struct.

```
struct {
  long item1;
  long item2;
  char item3[6];
}
```

The external format data of the structure in the previous example has a form equivalent to the following C++ struct.

```
struct {
  char item1[4];
  char item2[4];
  char item3[6];
}
```

Once added to a structure, the data size and external data size of the data item cannot change. Therefore, data items of type `IC4Char` and `IC4Byte` must be given appropriate sizes before adding them as members to a structure.

Once added to a structure, items cannot be removed from the structure. If the structure is destructed, items that were members of the structure should not be used. The storage associated with the items is no longer valid and cannot be reassigned.

Storage Assignment for an IC4Struct Data Item

Structures that have been assigned user-managed storage for local/external format data will assign storage locations for the members' local/external format data as the members are added. For structures that have not been assigned user-managed storage for local/external format data, calling the `dataPointer()` or `externalDataPointer()` member function of any member of the structure will cause an exception if the structure has not yet assigned storage to its members. The values of the structure's members can be accessed, but the members' pointers are not accessible until the structure assigns storage. Following is an example that would result in an exception:

```
IC4Bin4 item1(1);
IC4Bin4 item2(2);
IC4Struct structure;
structure << item1 << item2;
long* x = (long*)item1.dataPointer();  // THROWS AN EXCEPTION!
```

A structure that has not had user-managed storage assigned for its local format data defers assigning storage for its members' local format data until one of the following occurs:

- You explicitly assign storage for the local format data of the structure by using the `IC4Struct::setDataPointer()` member function.
- You request a pointer to the local format data of the structure by using the `IC4Struct::dataPointer()` member function.
- You request that the structure's data be converted between the local and external formats using the `IC4Struct::translateFromExternal()` member function or the `IC4Struct::translateToExternal()` member function.

Due to this deferred storage assignment behavior, if a structure has not assigned storage for the local format data of its members, calling the `dataPointer()` member function of any one of its members results in an exception being thrown. In both cases, the `errorId()` member function of the exception returns 1 and the first text line contains "1815".

Likewise, a structure that has not had user-managed storage assigned for its external format data defers assigning storage for the external format data of its members until one of the following occurs:

- You explicitly assign storage for the external format data of the structure by using the `IC4Struct::setExternalDataPointer()` member function.
- You request a pointer to the external format data of the structure by using the `IC4Struct::externalDataPointer()` member function.
- You request that the structure's data be converted between the local and external formats using the `IC4Struct::translateToExternal()` member function or the `IC4Struct::translateFromExternal()` member function.

Due to this deferred storage assignment behavior, if a structure has not assigned storage for the external format data of its members, calling the `externalDataPointer()` member function of any one of its members results in an exception being thrown. The `errorId()` member function of the exception returns 1 and the first text line contains "1815".

In general, you should not assign storage to data items that are destined to be members of a structure. The structure reassigns storage for its members. When storage is assigned, however, the values of the members are preserved; therefore, assigning storage to a data item can be used as a means of giving the data item a value without moving data.

Lifetime of Structure Members

Data items that are added as members of a structure must outlive the structure. Otherwise, the structure will have member references that are not valid, and the behavior of the structure will be unpredictable. A simple way to ensure that the members of a structure live as long as the structure is to make the members of a structure members of a class. This is shown in the following example:

```
class MyStruct: public IC4Struct
{
  public:
    IC4Bin4 a;
    IC4Char b;

    MyStruct(): IC4Struct(), a(36), b(32,32)
    {
      *this << a << b; // Add members to the structure
    }
}
```

The members of the structure are in the scope of MyStruct and are destructed when the MyStruct object is destructed.

Assignment Operations Between IC4Struct Data Items

Another reason for deriving a class from `IC4Struct` is to define an assignment operator. Because an `IC4Struct` data item deals with its members at the abstract (`IC4DataType`)

level, the IC4Struct data item has no way of assigning its members the values of another structure's members. The lack of an assignment operator prevents defining an array of structures (IC4Array<IC4Struct>). You, however, can define a specific IC4Struct-based structure with a meaningful assignment operator. The following example expands on the previous example.

```
class MyStruct: public IC4Struct
{
  public:
    IC4Bin4 a;
    IC4Char b;

    MyStruct(): IC4Struct(), a(36), b(32,32)
    {
      *this << a << b; // Add members to the structure
    }

    MyStruct& operator=(const MyStruct& other)
    {
       a = other.a;
       b = other.b;
       return *this;
    }
}
```

The new MyStruct type can now be used in the definition of an array of structures (IC4Array<MyStruct>).

Defining Arrays of Data Items

Arrays of data items are supported by the IC4Array template class. A data item in an array is called an element of the array. It is important to use the IC4Array data type instead of a C++ array if you want the elements of the array to map over contiguous storage. This is because data items manage their storage independently, and each data item has two values (the local format data value and the external format data value). By using the IC4Array data type, you are guaranteeing that the local format data of all array elements is in contiguous storage, and that the same is true for the external format data. The elements are packed to byte boundaries.

From here on, an array of data items is called an array.

An array constructs its elements when it is given the element count. This occurs either when the array is constructed or when the setElementCount() member function is called. In the following example, an array of five elements of type IC4Bin4 is constructed:

```
IC4Array<IC4Bin4> array(5); // Construct 5-element array.
```

In the following example, an array with no elements is constructed first and then the five elements are constructed in response to setElementCount():

```
IC4Array<IC4Bin4> array;  // Construct array with no elements.
array.setElementCount(5); // Make it a 5-element array.
```

Using the Data Type Access Classes

The elements of an array are constructed using the default constructor for their data type. Some data items are not completely constructed using the default constructor. For example, the default constructor for `IC4Char` constructs a data item with a size of zero. In cases like this, the `initializeElements()` member function must be called to initialize the elements, so that they have a size greater than zero before the array can be used. The `initializeElements()` member function must be given a data item of the appropriate size to be used for all of the elements of the array. For example:

```
IC4Array<IC4Char> charArray(5); // Construct 5-element array.
// The elements of the array have local and external data
// sizes of zero because the default IC4Char constructor is
// used implicitly by IC4Array.

// Construct an IC4Char data item that will be used to
// initialize the elements of the array.  Give the
// local format data an initial value and specify an
// external data size of 15 bytes.
IC4Char initializer("The initializer",15);

charArray.initializeElements(initializer);
// All 5 elements now have the same local and external data
// sizes and the same local format data value as the
// initializer.
```

Once the elements of an array have been given a size greater than zero, the size of an element cannot be changed.

An array of `IC4Struct` data items cannot be defined directly because the `IC4Struct` class has no public assignment operator. To define an array of structures, you must define a subclass of `IC4Struct` that has a public assignment operator. See the "MyStruct" example in “Assignment Operations Between IC4Struct Data Items” on page 54.

Using the Character String Data Type

The character string class, `IC4Char`, represents a string of AS/400 text characters that can be converted between code pages. The conversion member functions `translateFromExternal()` and `translateToExternal()` convert the data between local and external formats based on the code pages specified for the data. See “Overriding Code Pages” on page 60 for a discussion of code pages.

The `IC4Char` class is complex because it may represent a character string that has a combination of several of the following characteristics.

- Variable or fixed length
- Padded or unpadded (when external format data is fixed length)
- Comprised of single-byte or double-byte characters or a mixture of both, or is comprised of unicode characters
- Pure graphic or bracketed by shift-out and shift-in code points (if comprised of double-byte characters). See “Converting to the External Format and Padding External Format Data” on page 57 for more information.
- Code page overrides expected or not

The local and external format data of an IC4Char data item are of fixed size if the data item is contained in a structure (IC4Struct) or an array (IC4Array). A data item that is not contained in a structure or an array can be given a fixed size in an indirect way. To give the external format data a fixed size, give the data item an external data size greater than zero before calling translateToExternal() for the first time. Similarly, to give the local format data a fixed size, give the data item a local data size greater than zero before calling translateFromExternal() for the first time. By giving the data a size greater than zero, you are indicating that the size is important to you.

When an IC4Char data item has a fixed external size, the external format data is padded by the translateToExternal() member function unless padding is disabled. The enabling and disabling of padding is discussed in "Converting to the External Format and Padding External Format Data."

Assigning an Arbitrary Character Array

IC4Char has a member function that allows a data item to be assigned the value of an arbitrary array of characters. The member function is as follows:

```
void setDataValue(const char * value, unsigned long dataSize=0)
```

The array of characters may include null bytes. This is useful for assigning the data item double-byte or unicode character string values.

Assigning a Null-Terminated Character Array

IC4Char has an assignment operator, operator=(const char* value), that allows a data item to be assigned the value of a null-terminated character array. This assignment operator is similar to calling setDataValue(value,strlen(value)).

Converting to the External Format and Padding External Format Data

When the translateToExternal() member function is called, if characters that cannot be converted between code pages are found in the local format data, an exception is thrown.

When the external format data is of fixed length, translateToExternal() either does or does not pad the data. If translateToExternal() is to pad the external format data, padding is said to be enabled. If translateToExternal() is not to pad the external format data, padding is said to be disabled or not enabled. The discussion that follows describes more about external format data padding.

If padding is enabled, the external format data of an IC4Char data item is, by default, padded with EBCDIC blanks (0x40). The local format data is not affected by padding. Padding is performed by the translateToExternal() member function.

Padding is enabled if the external data size is greater than zero the first time translateToExternal() is called. When padding is enabled, the external data size is fixed. The fact that the external size was explicitly set by the user indicates that the size is significant and should not change. If the number of bytes resulting from the conversion exceeds the external data size, an exception is thrown.

Padding is disabled if the `translateToExternal()` member function is called when the external data size of the data item is zero. The fact that the external size was not explicitly set by the user before calling the translate member function indicates that the size is not significant. In this case, the external size is set to the number of bytes resulting from the conversion. Once disabled, the padding remains disabled, regardless of the external data size, unless explicitly re-enabled by the user. Padding can be re-enabled by calling the `setExternalPad()` member function.

An `IC4Char` data item that is contained in a structure or array has an external data size greater than zero. Padding can be disabled by calling the `setExternalPad()` member function and defaulting the argument to '\0'. Though padding is disabled, the fact that the data item is contained in a structure or array means the external data size is fixed. In this case, if the number of bytes resulting from the conversion is less than the external data size, the data size and the remainder of the string are left unchanged.

If the code page of the external format data is either a double-byte code page or a mixed double-byte and single-byte code page, sequences of double-byte characters resulting from the conversion are bracketed by shift-out and shift-in characters, by default. If the `IC4Char` item is to represent pure graphic data (that is, double-byte characters with no bracketing shift-out and shift-in characters), the `setGraphic()` member function must be called.

Converting from the External Format

When the `translateFromExternal()` member function is called, if characters that cannot be converted between code pages are found in the external format data, an exception is thrown.

If an `IC4Char` data item is contained in a structure or array, its local data size is not changed by the `translateFromExternal()` member function. Instead, if the number of bytes resulting from the conversion is less than the local data size, the data size and the remainder of the string are left unchanged. If the number of bytes resulting from the conversion is greater than the local data size, an exception is thrown.

In the following example, assume that *item* is an `IC4Char` data item contained in a structure *s*. Also assume that *item* has a data size and an external data size of 10, and that the conversion from external results in only 8 bytes of local format data. This kind of change in size is likely when dealing with double-byte data.

```
s.translateFromExternal();  // Convert the structure from
                            // external format to local format.
                            // Note that this converts "item" too.
if (item.translatedBytes() < item.dataSize())
{
  // As expected, the number of bytes resulting from the
  // conversion is less than the number of bytes of local format
  // data.
  cout << "Expected." << endl;
}
else
{
  // Error -- unexpected number of translated bytes.
  cout << "Error!" << endl;
}
```

If an IC4Char data item is not contained in a structure or an array, the local data size is set to the number of bytes resulting from the conversion.

Adding to a Structure or Initializing an Array

The default constructor for IC4Char produces data items of arbitrary length. Before a data item of this type can be added to a structure or be used to initialize the elements of an array, the data item must be given an appropriate size for its local and external format data. IC4Char has a form of the constructor and a member function called initialize() that serves this purpose. Note that after a data item is added to a structure, it cannot change in size. Also, once an array has been initialized, its elements cannot change in size.

Changing the Size of a Character String Data Item: An IC4Char data item that is not a member of a structure or an element of an array can change in size. For example, assigning an IC4Char data item the value of a character array sets the data size of the data item to the length of the character array while the external data value and size are unaffected. In this case, the external data value and size are not meaningful with respect to the current local format data value until translateToExternal() is called.

For an IC4Char data item that is a structure or an array, the sizes reflected by dataSize() and externalDataSize() cannot change. For example, assigning to an IC4Char data item the value of a character array whose length N is smaller than the data size of the item affects only the first N bytes of the local format data and leaves the remaining bytes unchanged.

Assigning Code Pages

The conversion between local and external format data of an IC4Char data item is dependent on the code pages associated with the local and external format data of the data item. Every IC4Char data item has its own pair of code pages.

By default, the code pages for the IC4Char data item are 437 (ASCII English) for local format data and 37 (EBCDIC English) for external format data. You can assign

different code pages for local and external format data using the following member functions.

```
void setCodePage(unsigned long codePage)
void setExternalCodePage(unsigned long externalCodePage)
void setCodePages(unsigned long codePage,
                  unsigned long externalCodePage)
void setCodePages(const char * systemName)
```

Overriding Code Pages

An `IC4Char` data item has two versions of the `translateFromExternal()` and `translateToExternal()` conversion member function. One version takes no arguments and when used, the code pages for the data item are used for conversion between local and external format data. The second version takes arguments for local and external format code pages. When the second version is used, the specified code pages are used for the conversion and the code pages for the data item are ignored. This is called **overriding code pages**.

In general, you would use the code pages for the data item instead of overriding them using the special versions of the conversion functions. However, in some cases overriding is useful. Overriding the code pages is especially useful in the following cases:

- Several `IC4Char` data items are members of a structure or elements of an array, and you want to control the code pages of some of the data items through the `IC4Struct` or `IC4Array` conversion member functions.
- An `IC4Char` data item (or an `IC4Struct` or `IC4Array` data item that contains an `IC4Char` data item) is an argument for a program call, and you want to convert the data based on the code pages of the PC and the AS/400 system that is the target of the program call.

In the latter case, the overriding is done by the program access class.

About Overriding and the Program Access Class: Let us assume that you are creating a PC program that copies a portion of a document back and forth between a PC and an AS/400. Your program is to be independent of the national language of the a document. In other words, your program must handle documents written in English, Japanese, French, Spanish, or any other language supported by the combined PC and AS/400 system. The document is presented to the user for editing and then stored on the AS/400 system. Your program assumes that the default code pages for the PC and AS/400 systems are to be used for conversion and that the code pages are conversion compatible. Your PC program calls a program on the AS/400 to retrieve part of a document and calls another program on the AS/400 to store the document. Both AS/400 programs expect a single structure as a parameter. Following is the definition of the structure.

Decimal Offset	Hexadecimal Offset	AS/400 Type	Description
0	0	Char(10)	Library name
10	A	Char(10)	Document name

Decimal Offset	Hexadecimal Offset	AS/400 Type	Description
20	14	Char(128)	Section ID
148	94	Binary(4)	Option
152	98	Binary(4)	Text length
156	9C	Char(*)	Text

The first three character fields of the structure are not national language dependent fields. The AS/400 programs expect the fields to be of code page 37 (EBCDIC English). The last character field, called **Text**, is national language dependent. It is the document text, and it is expected to be of the AS/400 system default code page. Your program does not know what the AS/400 system default code page is but the `IC4Program` class does. Your program needs a way to tell the program access class that it should override the code pages of the **Text** field. It does this using the `setCodePageOverrideExpected()` member function of the `IC4Char` data item representing the **Text** field.

The following class defines a data type for the structure. The use of `setCodePageOverrideExpected()` is emphasized.

```
class TextDescription: public IC4Struct
{
  public:
    // Define the fields of the structure...
    IC4Char libraryName;
    IC4Char documentName;
    IC4Char sectionId;
    IC4Bin4 option;
    IC4Bin4 textLength;
    IC4Char text;

    // Define the constructor for this data type...
    TextDescription() : libraryName("DOCLIB    "),
                        documentName("DOC       "),
                        sectionId(128),
                        option(),
                        textLength(1024),
                        text(1024)
    {
      // Initialize the local format data to blanks...
      memset(sectionId.dataPointer(),' ',128);
      memset(text.dataPointer(),' ',1024);

      // Indicate that the text field's code pages should be
      // overridden by the program access class...
      text.setCodePageOverrideExpected();
      // We must do this BEFORE adding the item as a member
      // of the structure.

      // Add the fields to the structure...
      *this << libraryName << documentName << sectionId
            << option << textLength << text;
```

```
    }
}
```

The `TextDescription` data type defined in the example is used to define an argument for the AS/400 programs that retrieve and store the document information.

Once the argument members have been set, the AS/400 program can be called. The code pages for the first three `IC4Char` members are not overridden. The code pages for the last text member are overridden. The program access class assumes that the code page of the local format data is the default code page of the PC system, and that the code page of the external format data is the default code page of the AS/400 system. The program access class gets both of these code pages by making API calls to AS/400 Client Access. By using this approach, you can ensure that information is maintained in your national language on both the PC and the AS/400.

About Overriding and Structures: The code pages for a structure may be overridden, but the override has no effect unless the structure has members that expect their code pages to be overridden (as specified by the `setCodePageOverrideExpected()` function). When a structure member does not expect its code pages to be overridden, the conversion is done using the member functions that do not take code page values as arguments. This allows creating a structure that has members that have code pages that are overridden as well as members that have code pages that are not overridden.

A structure does not expect its code pages to be overridden unless:

- It has a member that, at the time it was added to the structure, expected its code pages to be overridden or
- Its `setCodePageOverrideExpected()` function is called.

It is possible to call `setCodePageOverrideExpected()` for a member *after* it has been added to a structure. This call does not affect whether the code pages for the structure are overridden. If the structure did not expect its code pages to be overridden before the call, it still would not after the call. Therefore, the code pages for the structure are not implicitly overridden by the program access class or by any containing structure or array. However, the structure does override the code pages of its members that expect their code pages to be overridden. The structure uses its own code pages for the overrides.

The following is an example of a structure contained in another structure. The contained structure does not expect its code pages to be overridden. However, it does contain an `IC4Char` member that expects its code pages to be overridden.

```
// Define constants used in the example...

const unsigned int ASCIIJapaneseCodePage  =  932;
const unsigned int EBCDICJapaneseCodePage = 5026;

// The following is an ASCII Japanese string defined in code page 932.
const char ASCIIJapanese[9] =
    { 0xB2, 0x96,0xF2, 0x96,0xD1, 0x89,0x93, 0x61, 0x00 };


// Define the structure that is to be contained in another...

class StructB: public IC4Struct
{
  public:
    IC4Char c1;  // Code pages will not be overridden.
    IC4Char c2;  // Code pages will be overridden (see below).
    IC4Char c3;  // Code pages will not be overridden.

    StructB(): c1("c1"),
               c2((char *)ASCIIJapanese,10),
               c3("c3")
    {
      *this << c1 << c2 << c3;

      // We make the call to setCodePageOverrideExpected()
      // after adding the item to the structure so that
      // the structure will not expect its code pages to
      // be overridden...
      c2.setCodePageOverrideExpected();
      this->setCodePages(ASCIIJapaneseCodePage,
                         EBCDICJapaneseCodePage);
    }
}


// Define the structure that contains another structure...

class StructA: public IC4Struct
{
  public:
    StructB structB; // Code pages will NOT be overridden.
    IC4Char c; // Code pages will be overridden (see below).

    StructA(): structB(), c("c")
    {
      c.setCodePageOverrideExpected();

      *this << structB << c;
    }
}
```

Using the Data Type Access Classes

When a StructA data item is constructed and the translateToExternal() member function is called as follows:

```
StructA structA;
structA.translateToExternal(437,37);
```

The following results occur:

- The data item structA.structB.c1 is converted to external format based on its own code pages (which, by default, are 437 and 37).
- The data item structA.structB.c2 is converted to external format based on the code pages of structB (932 and 5026).
- The data item structA.structB.c3 is converted to external format based on its own code pages (which, by default, are 437 and 37).
- The data item structA.c is converted to external format based on the explicitly requested code pages 437 and 37.

When the setCodePageOverrideExpected() member function of the data item structA.structB is called, and then the translateToExternal() member function of the containing structure is called again as follows:

```
structA.structB.setCodePageOverrideExpected();
structA.translateToExternal(437,37);
```

The expected results are the same except for the conversion of data item structA.structB.c2. An attempt is made to convert structA.structB.c2 to external format based on the explicitly requested code pages 437 and 37. The attempt may or may not fail, but the results are probably not what is wanted. The results may not be what is wanted because the local data is ASCII Japanese, but the local data was converted as if it were ASCII English into EBCDIC English.

About Overriding and Arrays: The behavior of arrays relative to code-page dependent elements is similar to the behavior of structures.

If the elements of an array are initialized with an item that expects code pages to be overridden, the array will expect its code pages to be overridden. If the conversion member function for the array is called without specifying code pages, the elements are converted using the code pages of the array. If the conversion member function for the array is called with specific code pages, the elements are converted using the specified code pages.

If the setCodePageOverrideExpected() member function for an array element is called after the array elements have been initialized with an item that does not expect code pages to be overridden, the array does not expect code pages to be overridden. If the conversion member functions for the array are called, the elements that expect code pages to be overridden are converted using the code pages of the array.

If the setCodePageOverrideExpected() member function for an array is called but the array contains no elements that expect code pages to be overridden, the overriding of

the code pages for the array does not affect the code pages used for converting its elements. Each element is converted using its own code pages.

Using the Byte String Data Type

The byte string class, `IC4Byte`, represents an arbitrary string of AS/400 bytes. The string may be variable length or fixed length. It can be manipulated as an ordinary C++ void pointer (`void*`) under most circumstances. An `IC4Byte` data item has no special assignment operator.

An `IC4Byte` data item may have both local and external format data or it may have only external format data. The usual case is to have only external format data because `IC4Byte` data items represent data that you would not want to be converted. In some circumstances, an `IC4Byte` data item can be used as a placeholder for other types whose sizes and offsets cannot be determined until run-time. In such cases, the `IC4Byte` data item must be given both a local and an external size to accommodate the sizes of those data items that will eventually take its place. An `IC4Byte` data item does not have local format data unless a data size is specified on construction or on the `initialize()` member function.

An `IC4Byte` has a special member function, `setDataValue()`, that provides a convenient way of assigning a value to the data item.

The `translateFromExternal()` and `translateToExternal()` member functions of a data `IC4Byte` data item have no effect.

The default constructor for `IC4Byte` produces data items with a length of zero. Before a data item of this type can be added to a structure or be used to initialize the elements of an array, the data item must be given an appropriate size for its local and external format data. `IC4Byte` has a form of the constructor and a member function called `initialize()` that serve this purpose.

An `IC4Byte` data item that is not a member of a structure or an element of an array can change both the local and external data sizes.

For an `IC4Byte` data item that is a member of a structure or an element of an array, the sizes reflected by `dataSize()` and `externalDataSize()` cannot change. In such cases, an `IC4Byte` data item can only be assigned the value of another `IC4Byte` data item if the corresponding data sizes and external data sizes of the data items match.

A Simple Program

The following program uses many of the data types to define the layout of information contained in hypothetical data queues. The information in the data queues is in AS/400 data formats.

For this example, assume that you have an AS/400-based application comprised of one or more **requestor** programs that are dependent on another program, called the **request server**, for asynchronous service. Requests for service are sent to a data queue (called the **request queue**) and the request server handles the requests one at

a time. Responses to requests are sent to another data queue (called the **response queue**). The response queue is a keyed data queue where the key is the name of the requesting program.

In this example, you have decided to move one of the requestor programs out to a workstation to take advantage of workstation presentation facilities and to integrate with other applications on the workstation. However, the requester is still dependent on the request server running on the AS/400. You know the format of the requests (for example, the request queue entries), and you need some way to build requests in that format on the workstation and to send the requests to the request queue (for example, the data queue).

The format of a request is the following:

Offset	Type	Field Description
0	Char(10)	An identifier that uniquely identifies the requesting program.
10	Binary(2)	An identifier that indicates the kind of task to be done by the task handler.
12	Char(*)	Special information dependent on the kind of task to be done.

The requester sends several kinds of requests. Following is the format of the special information included with request 31, retrieve statistics:

Offset	Type	Field Description
0	Char(2)	Each bit indicates a different statistic to be retrieved. There are 16 statistics.

The format of a response to request 31 is the following:

Offset	Type	Field Description
0	Char(10)	An identifier that uniquely identifies the requesting program
10	Binary(2)	An identifier that indicates the request responded to (request 31)
12	Binary(2)	The number of statistics returned
14	Char(2)	Reserved
16	Binary(2)	Statistic ID (0 to 15)
18	Char(6)	Reserved
24	Float(8)	Statistic value

The last three fields repeat for each statistic requested.

The following program sends a retrieve statistics request to the request queue and retrieves the response:

```
#include <iostream.h>
#include "ic4dt400.hpp"
#include "ic4array.hpp"
#include "ic4struc.hpp"
#include "ic4dq.hpp"
#include "ic4keydq.hpp"

//---------------------------------------------------------------------
// Define a class to represent the request structure...
class Request: public IC4Struct
{
public:
  IC4Char               requester;
  IC4UnsignedBin2       reqId;
  IC4Byte               statSelection;

  Request(): IC4Struct(),
             requester("REQUESTER1"),
             reqId(31),                   // Retrieve statistics.
             statSelection("\xFF\xFF",2)  // Get ALL statistics.
  { *this << requester << reqId << statSelection; }

  Request& operator=(const Request& other)
  {
    IC4Struct::operator=(other);
    requester = other.requester;
    reqId = other.reqId;
    statSelection = other.statSelection;
    return *this;
  }
};

//---------------------------------------------------------------------
// Define a class to represent the variable portion of the response
// structure...
class StatStruct: public IC4Struct
{
public:
  IC4Bin2   statistic;
  IC4Byte   reserved;
  IC4Float8 value;

  StatStruct(): statistic(), reserved(6), value()
  { *this << statistic << reserved << value; }

  StatStruct& operator=(const StatStruct& other)
  {
    IC4Struct::operator=(other);
    statistic = other.statistic;
    reserved  = other.reserved;
    value = other.value;
    return *this;
  }
};
```

```
//---------------------------------------------------------------------
// Define a class to represent the response structure...
class Response: public IC4Struct
{
public:
  IC4Char               requester;
  IC4UnsignedBin2       reqId;
  IC4UnsignedBin2       statCount;
  IC4Byte               reserved;
  IC4Byte               tempStats;

  IC4Array<StatStruct>  stat;

  Response(): IC4Struct(),
              requester(10), reqId(), statCount(), reserved(2),
              tempStats(256,256)
  { *this << requester << reqId << statCount << reserved << tempStats; }

  Response& operator=(const Response& other)
  {
    IC4Struct::operator=(other);
    requester = other.requester;
    reqId = other.reqId;
    statCount = other.statCount;
    reserved = other.reserved;
    tempStats = other.tempStats;
    stat = other.stat;
    return *this;
  }

  void translateFromExternal()
  {
    IC4Struct::translateFromExternal();
    stat.setElementCount(statCount);
    stat.overlay(tempStats);
    stat.translateFromExternal();
  }
};

int main (int argc, char *argv[])
{
    //-----------------------------------------------------------------
    // Construct a request structure...
    Request request;

    //-----------------------------------------------------------------
    // The request needs to be converted for the AS/400...
    request.translateToExternal();

    //-----------------------------------------------------------------
    // Construct a request queue object and open a connection...
    IC4DataQueue reqQ("/QSYS.LIB/QUSRSYS.LIB/reqQ.DTAQ");
    reqQ.open();

    //-----------------------------------------------------------------
    // Send the request to the request queue...
    reqQ.write((char*)request.externalDataPointer(),
               request.externalDataSize());
```

```
    //-------------------------------------------------------------------
    // Construct a response structure...
    Response response;

    //-------------------------------------------------------------------
    // Construct a response queue object and open a connection...
    IC4KeyedDataQueue rspQ("/QSYS.LIB/QUSRSYS.LIB/rspQ.DTAQ");
    rspQ.open();

    //-------------------------------------------------------------------
    // We want the key converted to EBCDIC...
    rspQ.setConvertKey(True);

    //-------------------------------------------------------------------
    // Retrieve the response from the response queue...
    rspQ.read("REQUESTER1",(char*)response.externalDataPointer(),
              response.externalDataSize());

    //-------------------------------------------------------------------
    // The response needs to be converted for the PC...
    response.translateFromExternal();

    //-------------------------------------------------------------------
    // Print out the statistics...
    for (int i = 0; i < response.statCount; ++i)
    {
      cout << "Statistic " << response.stat[i].statistic << "  "
           << response.stat[i].value << endl;
    }

    return 0;
}
```

The `Response` class overrides `translateFromExternal()` so that it can overlay the returned data with an array of the appropriate size. The appropriate size is not known until the response data is retrieved from the response queue. In the mean time, the `Response` class uses an `IC4Byte` data item to get the retrieved statistics.

Using the Data Type Access Classes

Chapter 8. User Space Access Class

A **user space** is an AS/400 system object that is used for sharing data between programs. The object-type identifier for a user space is *USRSPC. A user space has no enforced structure and can be as large as 16MB.

The Access Class Library user space class provides a VisualAge for C++ class interface to help build applications that use these AS/400 objects.

User space access classes are supported on the following:

- OS/2 Optimized
- Windows 95
- Windows NT
- AS/400

Introduction to the User Space Access Class

User Space	The User Space class, `IC4UserSpace`, provides an interface to a user space. Member functions enable you to do such things as create a user space, read data from a user space, and write data into a user space.
User Space Exception Class	The User Space Exception class, `IC4UserSpaceException`, is derived from IC4Exception. All exceptions that are thrown by the user space classes are instances of `IC4UserSpaceException`.

Using the User Space Access Class

Following are examples of how the user space class can be used.

Creating and Deleting a User Space

To create and delete user spaces on an AS/400 system, member functions `create()` and `destroy()` are used. The `create()` member function does an implicit `open()` function call, and the `destroy()` member function does an implicit `close()` function call. Following is an example of creating a user space:

```
IC4UserSpace myUS("/QSYS.LIB/mylib.LIB/myus.USRSPC", "SystemA");
myUS.create("*DEFAULT", 1024, "*NO", " ", '\x00', "My User Space", "*ALL");
...
myUS.destroy();
```

The above example specifies all parameters. The following example uses the minimum number of parameters:

```
IC4UserSpace myUS("/QSYS.LIB/mylib.LIB/myus.USRSPC", "SystemA");
myUS.create(1024);
...
myUS.destroy();
```

Writing Data to a User Space

Writing data to a user space is quite simple.

```
IC4UserSpace myUS("/QSYS.LIB/mylib.LIB/myus.USRSPC", "SystemA");
IString data = "User space test data";
myUS.open();
...
myUS.write(data);
...
myUS.close();
```

Reading Data from a User Space

Reading data from a user space is also quite simple.

```
IC4UserSpace myUS("/QSYS.LIB/mylib.LIB/myus.USRSPC", "SystemA");
IString readData;
myUS.open();
...
myUS.read(readData, 1, 1000);
...
myUS.close();
```

Changing the Automatic Extendibility of a User Space

When the user space object is created, it is not automatically extendible. To make it extendible, use the `changeAutomaticExtendibility()` member function. Following is an example of changing the automatic extendibility attribute of a user space:

```
IC4UserSpace myUS("/QSYS.LIB/mylib.LIB/myus.USRSPC", "SystemA");
myUS.open();
myUS.changeAutomaticExtendibility(IC4UserSpace::Extendible);
...
myUS.close();
```

Changing the Initial Value of a User Space

Following is an example of changing the initial value of a user space:

```
IC4UserSpace myUS("/QSYS.LIB/mylib.LIB/myus.USRSPC", "SystemA");
myUS.open();
myUS.changeInitialValue('A');
...
myUS.close();
```

Changing the Size of a User Space

Following is an example of changing the size of a user space:

Using the User Space Access Class

```
IC4UserSpace myUS("/QSYS.LIB/mylib.LIB/myus.USRSPC", "SystemA");
myUS.open();
myUS.changeSize(10000);
...
myUS.close();
```

Using the User Space Access Class

Appendix A. Error Codes

Exception objects thrown by the Access Class Library consist of an error ID and one or more lines of text that give more detail about the error that occurred. The error ID can be accessed using the `IException::errorId()` member function. The text of the exception object can be accessed using `IException::text()`, which returns the last line of the text first.

The error ID indicates where the error was initiated. Use the error ID with the following table to determine what additional information is contained in the exception object, and where you can find more information about the error that occurred.

The following table lists the kinds of information contained in exception objects. In some cases, the second line of text or third line of text is not included.

Table 8 (Page 1 of 2). Information in Exception Objects

Error ID	Error Initiated By	Line 1 of Text	Line 2 of Text	Line 3 of Text
1	Access Class Library	Access classes' error code. For specific error codes, see Table 9 on page 76.		
2	OS/400	AS/400 message ID	Message text	
3	Client Access Optimized for OS/2	Client Access return code	Text associated with the return code	
4	AS/400 Client Access for OS/2	Client Access return code	Text associated with the return code	
5	ODBC Call-Level Interface	SQL state	Native error value	Message text
6	DB2 for OS/400 Call-Level Interface	SQL state	Native error value	Message text

Error Codes

Table 8 (Page 2 of 2). Information in Exception Objects

Error ID	Error Initiated By	Line 1 of Text	Line 2 of Text	Line 3 of Text
8	AS/400 Client Access for Windows 95/NT	Client Access return code	Text associated with the return code	

Additional information about an error ID:

Error ID	Information Source
1	This manual
2	Message text
3	See the appropriate header file from the following list: • CWB.H • CWBRC.H • CWBSV.H • CWBCO.H • CWBDQ.H • CWBNLCNV.H • CWBDT.H
4	*Client Access/400 for DOS and OS/2 API Reference*, SC41-3562
5	*Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*, ISBN 1-55615-658-8
6	*DB2 for OS/400 SQL Call Level Interface*, SC41-4806
8	See the appropriate header file from the following list: • CWB.H • CWBRC.H • CWBSV.H • CWBCO.H • CWBDQ.H • CWBNLCNV.H • CWBDT.H

The following table contains error codes that are contained in the text of the exception objects when the error is initiated by the Access Class Library (error Id 1). Line 1 of the text contains one of the error codes listed below:

Table 9 (Page 1 of 3). Exception Text for Error ID 1 Exceptions

Line 1 of Text	Meaning	Class Group
0001	IFS path name not valid	Common
0002	Invalid object type	Common
0003	Function not supported	Common
0004	Parameter out of range	Common
0005	Length parameter not valid	Common
0006	An `open()` was not performed but is required	Common
0007	NULL parameter not allowed	Common
0008	System name not correct	Common

Table 9 (Page 2 of 3). Exception Text for Error ID 1 Exceptions

Line 1 of Text	Meaning	Class Group
0009	Internal error in the Access Class Library	Common
0011	Incompatible arguments specified on member function call.	Common
1100	The value specified for decimal positions is not valid.	Data Area
1320	The keyword specified has not been added to this command object.	Command
1321	The keyword specified has already been added to this command object.	Command
1400	Object resources not allocated.	Database
1401	Retrieval of SQL status information failed.	Database
1402	Attempted to execute a SELECT statement without providing a *results* parameter.	Database
1403	The option specified requires a *value* parameter with a different type than the parameter provided.	Database
1404	Attempted to use a result set object which was closed or which never had result set data put into it.	Database
1405	Data mapping error occurred in requested column.	Database
1500	Target pointer is NULL.	Data Translation
1501	Source pointer is NULL.	Data Translation
1704	A code page value for external format data is not available because it was not set and the program has not been run.	Program
1705	An argument has an external data size of zero.	Program
1800	The size of an array can be set only once.	Data Types
1801	The size specified for the user-managed storage for local or external format data is too small.	Data Types
1802	The size of a structure member has changed. The structure cannot be used.	Data Types
1803	Attempting to overlay a data item, but offsets exceed the bounds of the allocated storage.	Data Types
1804	The operation cannot be done because the array has no elements.	Data Types
1805	The arrays must have the same number of elements when doing this operation.	Data Types
1806	Array elements must be initialized before operator[] can be used.	Data Types
1807	The size of an array element has changed. The array cannot be used.	Data Types
1808	The size of user-managed storage for local or external format data is not large enough for the array.	Data Types
1810	Assignment cannot be done because data items are not assignment compatible.	Data Types

Error Codes

Table 9 (Page 3 of 3). Exception Text for Error ID 1 Exceptions

Line 1 of Text	Meaning	Class Group
1812	Either the structure is contained by another structure or array, or the structure has assigned storage to its members and no additional members can be added.	Data Types
1813	The data item is contained so storage cannot be assigned.	Data Types
1814	The data item is contained and this operation would cause a change in size, so it cannot be done.	Data Types
1815	Local or external format data storage has not been assigned yet by the containing structure or array.	Data Types
1816	The data item is already contained by another structure or array.	Data Types
1817	User-managed storage is not large enough to complete this operation.	Data Types
1818	Trying to initialize a data item that is contained by another data item, or trying to initialize a data item with storage that is already owned by the data item.	Data Types
1819	The pad string is more than one byte and the number of bytes to be filled is not a multiple of the pad length.	Data Types
1822	The external format size must be greater than zero to do this operation.	Data Types
1823	The local format data must be greater than zero to do this operation.	Data Types

The data queue access classes and the user space access class do not generate any unique error codes. However, they do throw some general errors, which are listed in Table 9 on page 76.

Bibliography

This bibliography lists related publications that you may find helpful:

- The *Client Access/400 for DOS and OS/2 API Reference*, SC41-3562, provides detailed information about the application program interfaces (APIs) provided with Client Access for DOS and OS/2.
- The *Client Access for Windows 95 API and Technical Reference*, SC41-3513, contains the application programming interface information needed to write cooperative processing applications for Client Access/400.
- The *Client Access/400 Optimized for OS/2 API and Technical Reference*, SC41-3511, provides information about using the Client Access/400 Optimized for OS/2 API interface to create applications. Includes a description of the structure of API calls and examples of calls in certain programming languages.
- The *CL Reference*, SC41-4722, provides a description of the AS/400 control language (CL) and its OS/400 commands. (Non-OS/400 commands are described in the respective licensed program publications.) Also provides an overview of *all* the CL commands for the AS/400 system, and it describes the syntax rules needed to code them.
- The *DB2 for OS/400 SQL Call Level Interface*, SC41-4806, provides the information necessary for application programmers to write applications using the DB2 call level interface.
- The *IBM Access Class Library for Windows Reference*, SC41-4622, provides reference information on the access class library for the Windows 95 and Windows NT platforms. This book helps you create VisualAge for C++ applications that run on Windows and access AS/400 data and services.
- The *IBM Access Class Library for OS/400 Reference*, SC41-4620, provides reference information on the access class library for the OS/400 server. This book helps you create VisualAge for C++ for AS/400 applications that run on the AS/400 server and access AS/400 data and services.
- The *ILE Concepts*, SC41-4606, explains concepts and terminology pertaining to the Integrated Language Environment (ILE) architecture of the OS/400 licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- The *Integrated File System Introduction*, SC41-4711, provides an overview of the integrated file system, which includes:
  - What it is
  - Why you might want to use it
  - Concepts and terminology
  - The interfaces you can use to interact with it
  - The APIs and techniques you can use in programs that interact with it
  - Characteristics of individual file systems
- The *System API Reference*, SC41-4801, provides information for the experienced programmer on how to use the application programming interfaces (APIs) to such OS/400 functions as:
  - Dynamic Screen Manager
  - Files (database, spooled, hierarchical)
  - Message handling
  - National language support
  - Network management
  - Objects
  - Problem management
  - Registration facility
  - Security
  - Software products
  - Source debug
  - UNIX**-type
  - User-defined communications
  - User interface
  - Work management

  Includes original program model (OPM), Integrated Language Environment (ILE), and UNIX-type APIs.
- The *VisualAge for C++ for AS/400 C Library Reference*, SC09-2441, describes the run-time library functions available to C++ programs developed with VisualAge for C++ for AS/400.
- The *VisualAge for C++ for AS/400 C++ Language Reference*, SC09-2442, describes the C++ language as implemented by VisualAge for C++ for AS/400, including keywords, preprocessor directives, and constructs.
- The *VisualAge for C++ for AS/400 C++ Programming Guide*, SC41-2417, describes programming techniques to write ILE C++ applications that run on AS/400. Topics include performing I/O operations;

Bibliography

working with AS/400 files, devices, pointers, and locales; implementing interlanguage calls; using templates; handling exceptions; and porting code.

- The *VisualAge for C++ for AS/400 C++ User's Guide*, SC41-2416, explains how to compile, bind, and debug C++ applications that run in the Integrated Language Environment on the AS/400.
- The *VisualAge for C++ for AS/400 IBM Open Class Library Reference*, SC09-2440, describes the classes from the IBM Open Class Library: I/O Streams, Complex Mathematics, Collections, Data Types and Exceptions. Also describes the Binary Coded Decimal Class Library for OS/400.
- The *VisualAge for C++ for AS/400 IBM Open Class Library User's Guide*, SC09-2443, shows how to create C++ programs with the many different classes of the IBM Open Class Library: I/O Streams, Complex Mathematics, Collections, Data Types and Exceptions, and Database Access. Also describes how to create C++ programs using the IBM Binary Coded Decimal Class Library for OS/400.

The following book contains additional information that you may find helpful:

- *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*, ISBN 1-55615-658-8

Index

E

F

H

I

L

M

N

O

P

R

S

U

V

W

Reader Comments—We'd Like to Hear from You!

AS/400 Advanced Series
IBM Access Class Library
User's Guide
Version 3

Publication No. SC41-4623-00

Overall, how would you rate this manual?

	Very Satisfied	Satisfied	Dissatisfied	Very Dissatisfied
Overall satisfaction				

How satisfied are you that the information in this manual is:

Accurate				
Complete				
Easy to find				
Easy to understand				
Well organized				
Applicable to your tasks				
THANK YOU!				

Please tell us how we can improve this manual:

_____

_____

_____

_____

_____

May we contact you to discuss your responses? __ Yes __ No
Phone: (____) _____ Fax: (____) _____ Internet: _____

To return this form:

- Mail it
- Fax it
  United States and Canada: **800+937-3430**
  Other countries: **(+1)+507+253-5192**
- Hand it to your IBM representative.

Note that IBM may use or distribute the responses to this form without obligation.

_____ _____
Name Address

_____ _____
Company or Organization

_____ _____
Phone No.

Reader Comments—We'd Like to Hear from You!
SC41-4623-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN DEPT 542 IDCLERK
IBM CORPORATION
3605 HWY 52 N
ROCHESTER MN 55901-9986

Fold and Tape **Please do not staple** Fold and Tape

SC41-4623-00

Cut or Fold
Along Line

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

SC41-4623-00