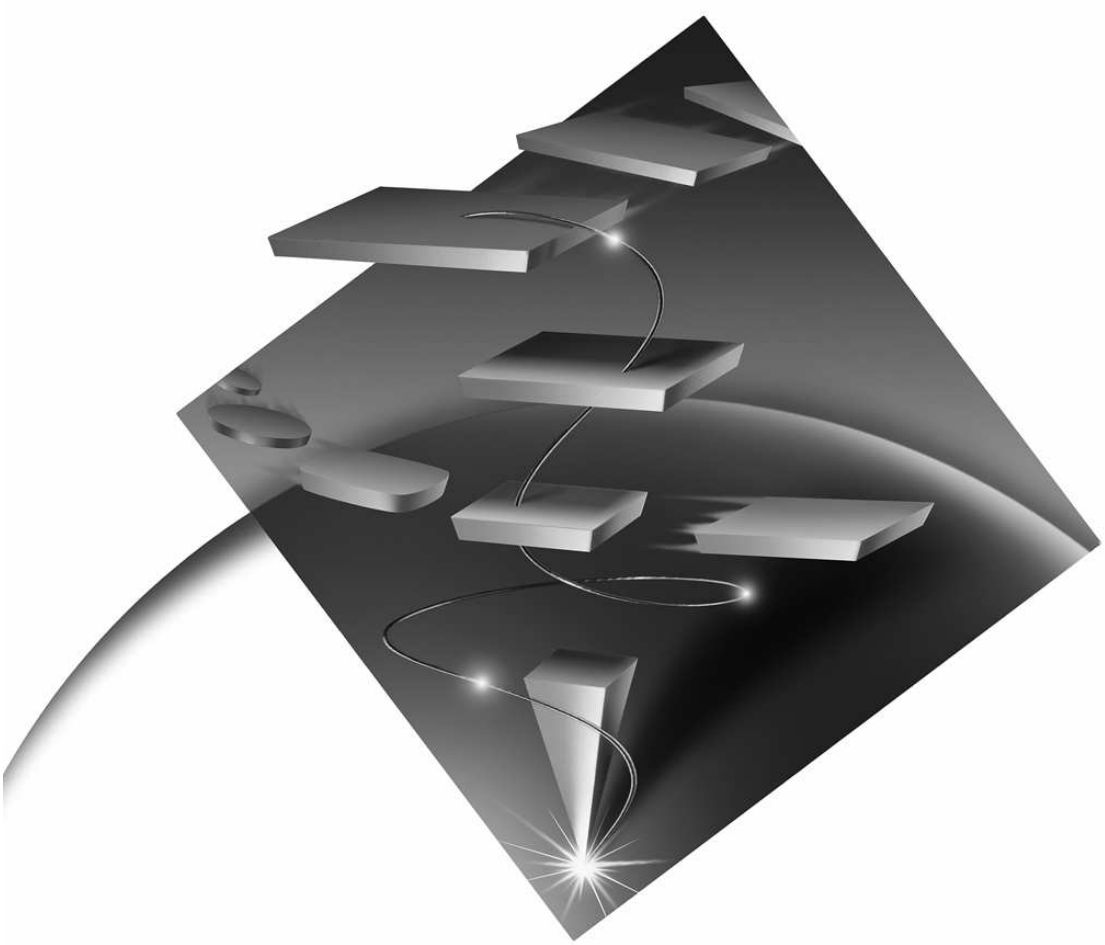
eNetwork Personal Communications
Version 4.2
for Windows 95 and Windows NT

Host Access Class Library



SC31-8527-00

eNetwork Personal Communications
Version 4.2
for Windows 95 and Windows NT

Host Access Class Library

SC31-8527-00

Note

Before using this information and the product it supports, be sure to read the general information under Appendix C, “Notices” on page 327.

First Edition (September 1997)

This edition applies to Version 4.2 of IBM eNetwork Personal Communications and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

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

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation
Department CGMD
P.O. Box 12195
Research Triangle Park, North Carolina
27709-2195

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

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

Contents

About This Book . . . xiii
Who Should Read This Book . . . xiii
How to Use This Book . . . xiii
Where to Find More Information . . . xiv

Chapter 1. Introduction . . . 1
C++ Objects . . . 1
Automation Objects . . . 2
LotusScript Extension . . . 2
ECL Concepts . . . 2
Connections, Handles and Names . . . 2
Sessions . . . 3
ECL Container Objects . . . 3
ECL List Objects . . . 4
Events . . . 4
Error Handling . . . 5
Addressing (Rows, Columns, Positions) . . . 5
Migrating from EHLLAPI . . . 6
Execution/Language Interface . . . 6
Features . . . 6
Session IDs . . . 7
Presentation Space Models . . . 8
SendKey Interface . . . 9
Events . . . 9
PS Connect/Disconnect and Multithreading . . . 9

Chapter 2. Host Access Class Library C++ . . . 11
Building C++ ECL Programs . . . 14
IBM Visual Age C++ . . . 14
Microsoft Visual C++ . . . 15
ECLBase Class . . . 17
Derivation . . . 17
ECLBase Methods . . . 18
GetVersion . . . 18
ConvertHandle2ShortName . . . 18
ConvertShortName2Handle . . . 19
ConvertTypeToString . . . 20
Convert Pos . . . 21
ECLConnection Class . . . 23
Derivation . . . 23
ECLConnection Methods . . . 24
ECLConnection Constructor . . . 24
ECLConnection Destructor . . . 25
GetHandle . . . 26
GetConnType . . . 27
GetName . . . 28
IsStarted . . . 29
IsCommStarted . . . 30
IsAPIEnabled . . . 31
IsReady . . . 32

© Copyright IBM Corp. 1997

GetCodePage . . . 32
StartCommunication . . . 33
StopCommunication . . . 34
RegisterCommEvent . . . 35
UnregisterCommEvent . . . 35
ECLConnList Class . . . 37
Derivation . . . 37
Usage Notes . . . 37
ECLConnList Methods . . . 38
ECLConnList Constructor . . . 38
ECLConnList Destructor . . . 39
GetFirstConnection . . . 39
GetNextConnection . . . 40
FindConnection . . . 41
GetCount . . . 42
Refresh . . . 43
ECLConnMgr Class . . . 45
Derivation . . . 45
ECLConnMgr Methods . . . 46
ECLConnMgr Constructor . . . 46
ECLConnMgr Deconstructor . . . 47
GetConnList . . . 47
Start Connection . . . 48
StopConnection . . . 49
RegisterStartEvent . . . 50
UnregisterStartEvent . . . 51
ECLCommNotify Class . . . 53
Derivation . . . 54
Usage Notes . . . 54
ECLCommNotify Methods . . . 57
NotifyEvent . . . 57
NotifyError . . . 57
NotifyStop . . . 58
ECLErr Class . . . 59
Derivation . . . 59
ECLErr Methods . . . 60
GetMsgNumber . . . 60
GetReasonCode . . . 61
GetMsgText . . . 61
ECLField Class . . . 63
Derivation . . . 63
ECLField Methods . . . 64
GetStart . . . 64
GetStartRow . . . 65
GetStartCol . . . 66
GetEnd . . . 67
GetEndRow . . . 68
GetEndCol . . . 69
GetLength . . . 70
GetText . . . 71
SetText . . . 72
IsModified, IsProtected, IsNumeric, IsHighIntensity, IsPenDetectable, IsDisplay . . . 74
GetAttribute . . . 75

ECLFieldList Class . . . 77
Derivation . . . 77
Properties . . . 77
ECLFieldList Methods . . . 78
Refresh . . . 78
GetFieldCount . . . 79
GetFirstField . . . 80
GetNextField . . . 81
FindField . . . 82
ECLKeyNotify Class . . . 84
Derivation . . . 85
Usage Notes . . . 85
ECLKeyNotify Methods . . . 88
NotifyEvent . . . 88
NotifyError . . . 89
NotifyStop . . . 89
ECLOIA Class . . . 90
Derivation . . . 90
Usage Notes . . . 90
ECLOIA Methods . . . 91
ECLOIA Constructor . . . 91
IsAlphanumeric . . . 92
IsAPL . . . 93
IsKatakana . . . 93
IsHiragana . . . 94
IsDBCS . . . 95
IsUpperShift . . . 95
IsNumeric . . . 96
IsCapsLock . . . 97
IsInsertMode . . . 97
IsCommErrorReminder . . . 98
IsMessageWaiting . . . 99
InputInhibited . . . 99
ECLPS Class . . . 101
Derivation . . . 101
Properties . . . 101
Usage Notes . . . 101
ECLPS Methods . . . 102
ECLPS Constructor . . . 102
ECLPS Destructor . . . 103
GetSize . . . 104
GetSizeRows . . . 105
GetSizeCols . . . 106
GetCursorPos . . . 106
GetCursorPosRow . . . 107
GetCursorPosCol . . . 108
SetCursorPos . . . 109
SendKeys . . . 110
SearchText . . . 111
GetText . . . 113
GetTextRect . . . 115
SetText . . . 117
ConvertPosToRowCol . . . 118
ConvertRowColToPos . . . 119

ConvertPosToRow . . . 120
ConvertPosToCol . . . 120
RegisterKeyEvent . . . 121
UnregisterKeyEvent . . . 123
GetFieldList . . . 123
ECLSession Class . . . 125
Derivation . . . 125
Properties . . . 125
Usage Notes . . . 125
ECLSession Methods . . . 126
ECLSession Constructor . . . 126
ECLSession Destructor . . . 127
GetPS . . . 127
GetOIA . . . 128
GetXfer . . . 129
GetWinMetrics . . . 130
RegisterUpdateEvent . . . 131
UnregisterUpdateEvent . . . 132
ECLStartNotify Class . . . 133
Derivation . . . 134
Usage Notes . . . 134
ECLStartNotify Methods . . . 136
NotifyEvent . . . 136
NotifyError . . . 136
NotifyStop . . . 137
ECLUpdateNotify Class . . . 138
Derivation . . . 139
Usage Notes . . . 139
ECLUpdateNotify Methods . . . 141
NotifyEvent . . . 141
NotifyError . . . 141
NotifyStop . . . 142
ECLWinMetrics Class . . . 143
Derivation . . . 143
Properties . . . 143
Usage Notes . . . 143
ECLWinMetrics Methods . . . 144
ECLWinMetrics Constructor . . . 144
ECLWinMetrics Destructor . . . 145
GetWindowTitle . . . 146
SetWindowTitle . . . 147
GetXpos . . . 148
SetXpos . . . 148
GetYpos . . . 149
SetYpos . . . 150
GetWidth . . . 151
SetWidth . . . 152
GetHeight . . . 153
SetHeight . . . 154
GetWindowRect . . . 155
SetWindowRect . . . 156
IsVisible . . . 157
SetVisible . . . 157
Active . . . 158

SetActive . . . 158
IsMinimized . . . 159
SetMinimized . . . 160
IsMaximized . . . 160
SetMaximized . . . 161
IsRestored . . . 161
SetRestored . . . 162
ECLXfer Class . . . 164
Derivation . . . 164
Properties . . . 164
Usage Notes . . . 164
ECLXfer Methods . . . 165
ECLXfer Constructor . . . 165
ECLXfer Destructor . . . 166
SendFile . . . 166
ReceiveFile . . . 168

Chapter 3. Host Access Class Library Automation Objects . . . 171
autECLConnList Class . . . 173
Properties . . . 173
autECLConnList Methods . . . 177
Collection Element Methods . . . 177
Refresh . . . 177
FindConnectionByHandle . . . 177
FindConnectionByName . . . 178
StartCommunication . . . 179
StopCommunication . . . 179
autECLConnMgr Class . . . 181
Properties . . . 181
autECLConnMgr Methods . . . 182
StartConnection . . . 182
StopConnection . . . 183
autECLFieldList Class . . . 184
Properties . . . 184
autECLFieldList Methods . . . 191
Collection Element Methods . . . 191
Refresh . . . 191
FindFieldByRowCol . . . 192
FindFieldByText . . . 192
GetText . . . 193
SetText . . . 194
autECLOIA Class . . . 195
Properties . . . 195
autECLOIA Methods . . . 204
SetConnectionByName . . . 204
SetConnectionByHandle . . . 204
StartCommunication . . . 205
StopCommunication . . . 206
autECLPS Class . . . 207
Properties . . . 207
autECLPS Methods . . . 213
SetConnectionByName . . . 213
SetConnectionByHandle . . . 213
SetCursorPos . . . 214

SendKeys . . . 215
SearchText . . . 215
GetText . . . 216
SetText . . . 217
GetTextRect . . . 218
StartCommunication . . . 219
StopCommunication . . . 219
autECLSession Class . . . 221
Properties . . . 221
autECLSession Methods . . . 227
SetConnectionByName . . . 227
SetConnectionByHandle . . . 227
StartCommunication . . . 228
StopCommunication . . . 229
autECLWinMetrics Class . . . 230
Properties . . . 230
autECLWinMetrics Methods . . . 239
SetConnectionByName . . . 239
SetConnectionByHandle . . . 239
GetWindowRect . . . 240
SetWindowRect . . . 241
StartCommunication . . . 241
StopCommunication . . . 242
autECLXfer Class . . . 243
Properties . . . 243
autECLXfer Methods . . . 247
SetConnectionByName . . . 247
SetConnectionByHandle . . . 247
SendFile . . . 248
ReceiveFile . . . 249
StartCommunication . . . 250
StopCommunication . . . 251

Chapter 4. Host Access Class Library LotusScript Extension . . . 253
lsxECLConnection Class . . . 255
Properties . . . 255
lsxECLConnection Methods . . . 259
StartCommunication . . . 259
StopCommunication . . . 259
lsxECLConnList Class . . . 261
Properties . . . 261
lsxECLConnList Methods . . . 263
Refresh . . . 263
FindConnectionByHandle . . . 263
FindConnectionByName . . . 264
lsxECLConnMgr Class . . . 265
Properties . . . 265
lsxECLConnMgr Methods . . . 266
StartConnection . . . 266
StopConnection . . . 267
lsxECLField Class . . . 268
Properties . . . 268
lsxECLField Methods . . . 273
GetText . . . 273

SetText . . . 273
lsxECLFieldList Class . . . 275
Properties . . . 275
lsxECLFieldList Methods . . . 276
Refresh . . . 276
FindFieldByRowCol . . . 276
FindFieldByText . . . 277
lsxECLOIA Class . . . 279
Properties . . . 279
lsxECLOIA Methods . . . 287
lsxECLPS Class . . . 288
Properties . . . 289
lsxECLPS Methods . . . 293
SetCursorPos . . . 293
SendKeys . . . 293
SearchText . . . 294
GetText . . . 295
SetText . . . 296
GetTextRect . . . 297
lsxECLSession Class . . . 298
Properties . . . 298
lsxECLSession Methods . . . 303
lsxECLWinMetrics Class . . . 304
Properties . . . 305
lsxECLWinMetrics Methods . . . 311
GetWindowRect . . . 311
SetWindowRect . . . 311
lsxECLXfer Class . . . 313
Properties . . . 314
lsxECLXfer Methods . . . 317
SendFile . . . 317
ReceiveFile . . . 318

Appendix A. Sendkeys Mnemonic Keywords . . . 319

Appendix B. ECL Planes — Format and Content . . . 323
TextPlane . . . 323
FieldPlane . . . 323
ColorPlane . . . 325
ExtendedFieldPlane . . . 326

Appendix C. Notices . . . 327

Appendix D. Trademarks . . . 329

Index . . . 331

Figures

1. ECL Layers 1
2. Host Access Class Objects 12
3. Host Access Class Library Automation Objects 172

Tables

1. Mnemonic Keywords for the Sendkey Method 319
2. 3270 Field Attributes 324
3. 5250 Field Attributes 324
4. Color Plane Information 325
5. 3270 Extended Character Attributes 326
6. 5250 Extended Character Attributes 326

© Copyright IBM Corp. 1997

About This Book

This book provides necessary programming information for you to use the IBM eNetwork Personal Communications for Windows 95 and Windows NT Host Access Class Library (ECL). In this book, *Windows* refers to Windows 95 and Windows NT. Throughout this book, *workstation* refers to all supported personal computers. When only one model or architecture of the personal computer is referred to, only that type is specified.

Who Should Read This Book

This book is intended for programmers and developers who write application programs that use the Host Access Class Library (ECL) functions.

A working knowledge of Windows is assumed. For information about Windows, refer to the list of publications under “Where to Find More Information” on page xiv.

This book assumes you are familiar with the language and compiler that you are using. For information on how to write, compile, or link-edit programs, refer to “Where to Find More Information” on page xiv for the appropriate references for the specific language you are using.

How to Use This Book

This book is organized as follows:

Chapter 1, “Introduction” on page 1, gives an overview of the Host Access Class Library.

Chapter 2, “Host Access Class Library C++” on page 11, describes the Host Access Class Library C++ methods and properties.

Chapter 3, “Host Access Class Library Automation Objects” on page 171, describes the methods and properties of the Host Access Class Library Automation Objects.

Chapter 4, “Host Access Class Library LotusScript Extension” on page 253, describes the Host Access Class Library methods and properties of the Host Access Class Library LotusScript Extension.

Appendix A, “Sendkeys Mnemonic Keywords” on page 319, contains the mnemonic keywords for the Sendkeys method.

Appendix B, “ECL Planes — Format and Content” on page 323, describes the format and contents of the different data planes in the ECL presentation space model.

Conventions Used in This Book

The following conventions are used throughout the eNetwork Personal Communications or Communications Server library. Some of the conventions listed might not be used in this particular book.

© Copyright IBM Corp. 1997

Text Conventions

Bold	Bold type indicates verbs, functions, and parameters that you can use in a program or at a command prompt.
Italics	Italic type indicates the following things: A variable that you supply a value for. The names of window controls, such as lists, check boxes, entry fields, push buttons, and menu choices. They appear in the text as they appear in the window. Book titles. A letter is being used as a letter or a word is being used as a word. **Example:** When you see an *a*, make sure it is not supposed to be an *an*.
Bold italics	Bold italic type is used to emphasize a word.
UPPERCASE	Uppercase indicates constants, file names, keywords, and options that you can use in a program or at a command prompt. You can enter these values in uppercase or lowercase.
Example type	Example type indicates information that you are instructed to type at a command prompt or in a window.

Where to Find More Information

Following is a list of related publications:

IBM eNetwork Personal Communications Version 4.2 Quick Beginnings for Windows 95 and Windows NT, SC31-8476

IBM eNetwork Personal Communications Version 4.2 Reference for Windows 95 and Windows NT, SC31-8477

IBM eNetwork Personal Communications Version 4.2 System Management Programming for Windows 95 and Windows NT, SC31-8480

IBM eNetwork Personal Communications Version 4.2 Client/Server Programming for Windows 95 and Windows NT, SC32-8479

IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming, SC31-8316

See also:

IBM 3270 Information Display System Data Stream Programmer's Reference, GA23-0059

IBM 5250 Information Display System Functions Reference Manual, SA21-9247

Chapter 1. Introduction

The Host Access Class Library (ECL) is a set of objects that allow application programmers to access host applications easily and quickly. IBM eNetwork Personal Communications provides support for a wide variety of programming languages and environments by supporting three different ECL layers: C++ objects, Microsoft's COM-based automation technology (OLE), and LotusScript Extension (LSX). Each layer provides the same basic functionality, but each layer has some differences due to the different syntax and capabilities of each environment. The most functional and flexible layer is the C++ layer, which provides the basis for all others.

This layering concept allows the basic ECL functions to be used with a wide variety of programming environments including Microsoft Visual Basic, Visual Basic for Applications, Lotus Notes, Lotus WordPro and Visual C++. The following figure shows the ECL layers.

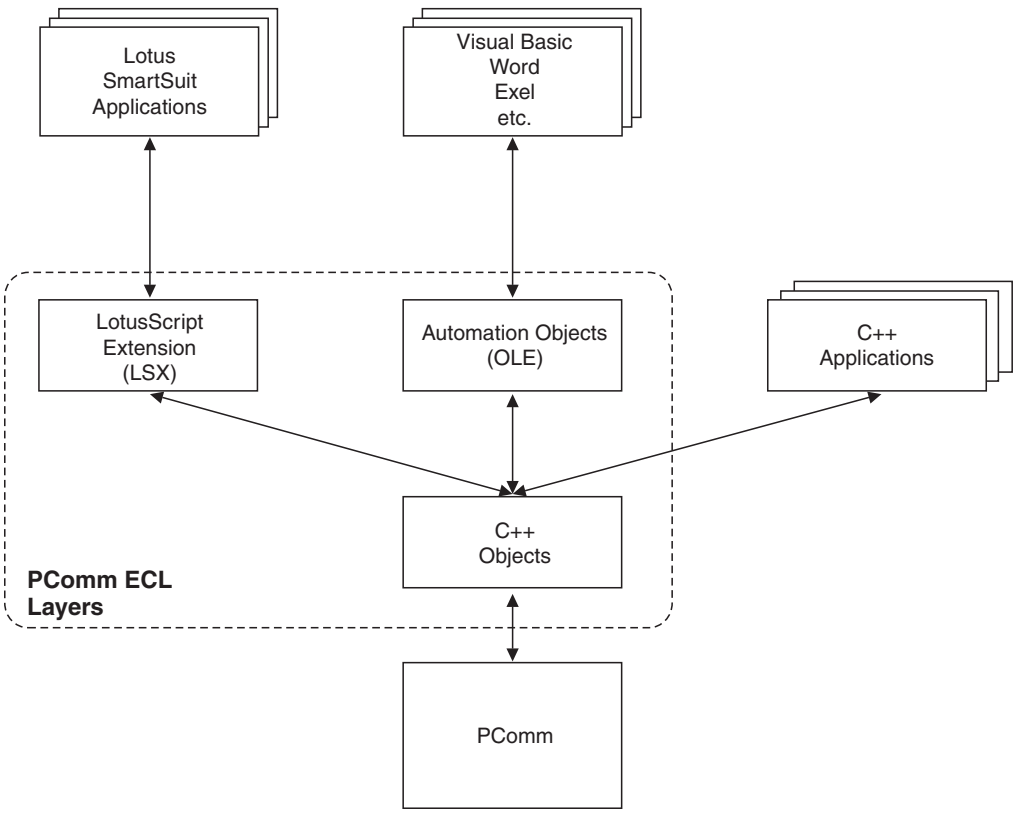


Figure 1. ECL Layers

C++ Objects

This C++ class library presents a complete object-oriented abstraction of a host connection that includes: reading and writing the host presentation space (screen), enumerating the fields on the screen, reading the Operator Indicator Area (OIA) for status information, accessing and updating information about the visual emulator

© Copyright IBM Corp. 1997

window, transferring files, and performing asynchronous notification of significant events. The class libraries support Microsoft Visual C++ compilers.

See Chapter 2, “Host Access Class Library C++” on page 11 for details on C++ objects.

Automation Objects

The Host Access Class Library Automation Objects allow Personal Communications to support Microsoft’s COM-based automation technology (formerly known as OLE automation). The ECL Automation Objects are a series of automation servers that allow automation controllers, for example, Microsoft’s Visual Basic, to programmatically access Personal Communications’ data and functionality. In other words, applications that are enabled for controlling the automation protocol (automation controller) can control some of Personal Communications’ operations (automation server).

See Chapter 3, “Host Access Class Library Automation Objects” on page 171 for details on the Automation Objects layer.

LotusScript Extension

The Host Access Class Library LotusScript Extension (LSX) is a language extension module for LotusScript (the scripting and macro language of Lotus Notes and all the Lotus SmartSuite products). This LSX gives users of Lotus products access to the ECL functions through easy-to-use scripting functions.

See Chapter 4, “Host Access Class Library LotusScript Extension” on page 253 for details on the LotusScript layer.

ECL Concepts

The following sections describe several essential concepts of the ECL. Understanding these concepts will aid you in making effective use of the library.

Connections, Handles and Names

In the context of the ECL, a connection is a single, unique Personal Communications emulator window. The emulator window may or may not be actually connected to a host and may or may not be visible on the screen. For instance, a Personal Communications window can be in a “disconnected” state. Connections are distinguished by their connection handle or by their connection name. Most ECL objects are associated with a specific connection. Typically, the object takes a connection handle or connection name as a parameter on the constructor of the object. For languages like Visual Basic that do not support parameters on constructors, a member function is supplied for making the association. Once constructed, the object cannot be associated with any other connection. For example, to create an ECLPS (Presentation Space) object associated with connection “B”, the following code would be used:

C++

```
ECLPS *PSObject;
PSObject = new ECLPS('B');
```

Visual Basic

```
Dim PSObject as Object
Set PSObject = CreateObject("PCOMM.autECLPS")
PSObject.SetConnectionByName("B")
```

LotusScript Extension

```
dim myPSObj as new lsxECLPS("B")
```

An ECL connection name is a single character from A to Z using upper case characters. There are a maximum of 26 connection names, and Personal Communications is currently limited to 26 concurrent connections. A connection's name is the same as its EHLLAPI short session ID, and the session ID shown on the Personal Communications window title and OIA.

An ECL handle is a unique 32-bit number that represents a single connection. Unlike a connection name, a connection handle is not limited to 26 values, and the value itself has no significance to the application. You can use a connection handle across threads and processes to refer to the same connection.

For future expandability, applications should use the connection handle whenever possible. Most ECL objects accept a handle or a name when a connection needs to be identified. There are functions available in the base ECL class to convert a handle to a name, and a name to a handle. These functions are available from any ECL object.

Note: Connection properties are dynamic. For example, the connection type returned by GetConnType may change if you reconfigure the connection to a different host. In general, the application should not assume that connection properties remain fixed.

Sessions

In the context of the ECL, a session object (ECLSession) is only a container for all the other connection-specific objects. It provides a shortcut for an application to create a complete set of ECL objects for a particular connection. The term *session* should not be confused with the Personal Communications session concept. A Personal Communications session refers to a physical emulation window on the screen.

Creating or destroying ECLSession objects does not effect Personal Communications sessions (windows). An application can create any number of ECLSession objects that refer to the same or different connections.

ECL Container Objects

Several of the ECL classes act as containers of other objects. For example, the ECLSession object contains an instance of the ECLPS, ECLOIA, ECLWinMetrics, and ECLXfer objects. Containers provide methods to return a pointer to the contained object. For example, the ECLSesison object has a GetOIA method, which returns a pointer to an OIA object. Contained objects are not implemented as public members of the container's class, but rather are accessed only through methods.

For performance or other reasons, the contained objects may or may not be created when the container object is created. The class implementation may choose to defer construction of the contained objects until the first time the application requests a pointer to them. The application should not assume that contained objects are created at the same time as the container. For example, an instance of the ECLPS object may not be constructed when an ECLSession object is constructed. Instead, the ECLSession class may delay the construction of the ECLPS object until the first time the GetPS method is called.

When a container class is destroyed, all the contained instances are also destroyed. Any pointers that have been returned to the application become invalid and must not be used.

Note: Some ECL layers (such as the Automation Objects) may hide the containment scheme or recast it into a naming scheme that does not use explicit pointers

ECL List Objects

Several ECL classes provide list iteration capabilities. For example, the ECLConnList class manages the list of connections. ECL list classes are not asynchronously updated to reflect changes in the list content. The application must explicitly call the Refresh method to update the contents of a list. This allows an application to iterate a list without concern that the list may change during the iteration.

Events

The ECL provides the capability of asynchronous notification of certain events. An application can choose to be notified when specific events occur. For example, the application can be notified when a new Personal Communications connection starts. Currently the ECL supports notification for the following events:

Connection start/stop

Communications connect/disconnect

Operator keystrokes

Presentation space or OIA updates

Notification of events is implemented by the ECLNotify abstract base classes. A separate class exists for each event type. To be notified of an event, the application must define and create an object derived from one of the ECLNotify abstract base classes. That object must then be registered by calling the appropriate ECL registration function. Once an application object is registered, it's NotifyEvent method is called whenever the event of interest occurs.

Notes:

1. The application's NotifyEvent method is called asynchronously on a separate thread of execution. Therefore, the NotifyEvent method should be re-entrant, and if it accesses application resources, appropriate locking or synchronization should be used.
2. Some ECL layers (such as the Automation Objects) may not fully support or implement ECL events.

Error Handling

At the C++ layer, ECL uses C++ structured exception handling. In general, errors are indicated to the application by the throwing of a C++ exception with an ECLErr object. To catch errors, the application should enclose calls to the ECL objects in a try/catch block such as:

```
try {
    PSObj = new ECLPS('A');
    x = PSObj->GetSize();

    //...more references to ECL objects...

} catch (ECLErr ErrObj) {
    ErrNumber = ErrObj.GetMsgNumber();
    MessageBox(NULL, ErrObj.GetMsgText(), "ECL Error");
}
```

When an ECL error is caught, the application can call methods of the ECLErr object to determine the exact cause of the error. The ECLErr object can also be called to construct a complete language-sensitive error message.

In both the Automation Objects layer and the LotusScript Extension layer, runtime errors cause an appropriate scripting error to be created. An application can use an On Error handler to capture the error, query additional information about the error and take appropriate action.

Addressing (Rows, Columns, Positions)

The ECL provides two ways of addressing points (character positions) in the host presentation space. The application can address characters by row/column numbers, or by a single linear position value. Presentation space addressing is always 1-based (not zero-based) no matter what addressing scheme is used.

The row/column addressing scheme is useful for applications that relate directly to the physical screen presentation of the host data. The rectangular coordinate system (with row 1 column 1 in the upper left corner) is a natural way to address points on the screen. The linear positional addressing method (with position 1 in the upper left corner, progressing from left to right, top to bottom) is useful for applications that view the entire presentation space as a single array of data elements, or for applications ported from the EHLLAPI interface which uses this addressing scheme.

At the C++ layer, the different addressing schemes are chosen by calling different signatures for the same methods. For example, to move the host cursor to a given screen coordinate, the application can call the ECLPS::SetCursorPos method in one of two signatures:

```
PSObj->SetCusorPos(81);
PSObj->SetCursorPos(2, 1);
```

These statements have the same effect if the host screen is configured for 80 columns per row. This example also points out a subtle difference in the addressing schemes — the linear position method can yield unexpected results if the application makes assumptions about the number of characters per row of the presentation space. For example, the first line of code in the example would put the cursor at column 81 of row 1 in a presentation space configured for 132 columns.

The second line of code would put the cursor at row 2 column 1 no matter what the configuration of the presentation space.

Note: Some ECL layers may expose only a single addressing scheme.

Migrating from EHLLAPI

Applications currently written to the Emulator High Level Language API (EHLLAPI) can be modified to use the Host Access Class Library. In general it requires significant source code changes or application restructuring to migrate from EHLLAPI to ECL. ECL presents a different programming model than EHLLAPI and in general requires a different application structure to be effective.

The following sections will help a programmer familiar with EHLLAPI understand how ECL is similar and how ECL is different than EHLLAPI. Using this information you can understand how a particular application can be modified to use the ECL.

Note: EHLLAPI uses the term *session* to mean the same thing as an ECL *connection*. The terms are used interchangeably in this section.

Execution/Language Interface

At the most fundamental level, EHLLAPI and ECL differ in the mechanics of how the API is called by an application program.

EHLLAPI is implemented as a single call-point interface with multiple-use parameters. A single entry point (hllapi) in a DLL provides all the functions based on a fixed set of four parameters. Three of the parameters take on different meanings depending on the value of the forth command parameter. This simple interface makes is easier to call the API from a variety of programming environments and languages. The disadvantage is a lot of complexity packed into one function and four parameters.

ECL is an object-oriented interface that provides a set of programming objects instead of explicit entry points or functions. The objects have properties and methods that can be used to manipulate a host connection. You do not have to be concerned with details of structure packing and parameter command codes, but can focus on the application functions. ECL objects can only be used from one of the supported ECL layer environments (C++, Automation Objects, or LotusScript). These three layers are accessible to most modern programming environments such as Microsoft Visual C++, Visual Basic and Lotus SmartSuit applications.

Features

At a high level, ECL provides a number of features not available at the EHLLAPI level. There are also a few features of EHLLAPI not currently implemented in any ECL class.

ECL unique features include:

Connection (session) start/stop functions

Event notification for host communications link connect/disconnect

Event notification for connection (session) start/stop

Comprehensive error trapping

Generation of language-specific error message text

No architectural limit to the number of connections (sessions). Currently, Personal Communications is limited to 26.

Support for multiple concurrent connections (sessions) and multithreaded applications

Row/column addressing for host presentation space

Simplified model for presentation space

Automatic generation of list of fields and attributes

Keyword-based function key strings

EHLLAPI features not currently implemented in the ECL include:

Structured field support

OIA character images

Lock/unlock presentation space

Session IDs

The ECL architecture is not limited to 26 sessions. Therefore, a single character session ID such as that used in EHLLAPI is not appropriate. The ECL uses the concept of a connection handle, which is a simple 32-bit value that has no particular meaning to the application. A connection handle uniquely identifies a specific connection (session). You can use a connection handle across threads and processes to refer to the same connection.

All ECL objects and methods that need to reference a particular connection accept a connection handle. In addition, for backward compatibility and to allow a reference from the emulator user interface (which does not display the handle), some objects and methods also accept the traditional session ID. The application can obtain a connection handle by enumerating the connections with the ECLConnList object. Each connection is represented by an ECLConnection object. The ECLConnection::GetHandle method can be used to retrieve the handle associated with that specific connection.

It is highly recommended that applications use connection handles instead of connection names (EHLLAPI short session ID). Future implementations of the ECL may prevent applications that use connection names from accessing more than 26 sessions. In some cases it may be necessary to use the name, such as when the user is required to input the name of a specific session the application is to utilize. In the following C+ + example, you supply the name of a session. The application then finds the connection in the connection list and creates PS and OIA objects for that session:

```
ECLConnList                    ConnList;            // Connection list
ECLConnection            *ConnFound;            // Ptr to found connection
ECLPS                            *PS;                    // Ptr to PS object
ECLOIA                           *OIA;                   // Ptr to OIA object
char                           UserRequestedID;

//... user inputs a session name (A-Z) and it is put
//... into the UserRequesteID variable.   Then...

ConnList.Refresh();          // Update list of connections
ConnFound = ConnList.FindConnection(UserRequestedID);
if (ConnFound == NULL) {
   // Session name given by user does not exist...
}
else {
   // Create PS and OIA objects using handle of the
   // connection just found:
   PS = new ECLPS(ConnFound.GetHandle());
  OIA= new  ECLOIA(ConnFound.GetHandle());

   // The following would also work, but is not the
   // preferred method:
   PS = new ECLPS(UserRequestedID);
   OIA= new ECLOIA(UserRequestedID);
}
```

The second way of creating the PS and OIA objects shown in the example is not preferred because is uses the session name instead of the handle. This creates an implicit 26-session limit in this section of the code. Using the first example shown allows that section of code to work for any number of sessions.

Presentation Space Models

The ECL presentation space model is easier to use than that of EHLLAPI. The ECL presentation space consists of a number of *planes*, each of which contains one type of data. The planes are:

Text

Field attributes

Color

Extended attributes

The planes are all the same size and contain one byte for each character position in the host presentation space. An application can obtain any plane of interest using the ECLPS::GetScreen method.

This model is different from the EHLLAPI, in which text and non-text presentation space data is often interleaved in a buffer. An application must set the EHLLAPI session parameter to specify what type of data to retrieve, then make another call to copy the data to a buffer. The ECL model allows the application to get the data of interest in a single call and different data types are never mixed in a single buffer.

SendKey Interface

The ECL method for sending keystrokes to the host (ECLPS::Sendkeys) is similar to the EHLLAPI SendKey function. However, EHLLAPI uses cryptic escape codes to represent non-text keys such as ENTER, PF1 and BACKTAB. The ECLPS object uses bracketed keywords to represent these keystrokes. For example, the following C++ sample would type the characters “ABC” at the current cursor position, followed by an ENTER key:

```
ECLPS           *PS;

PS = new ECLPS('A');                // Get PS object for "A"
PS->SendKeys("ABC[enter]");         // Send keystrokes
```

Events

EHLLAPI provides some means for an application to receive asynchronous notification of certain events. However, the event models are not consistent (some events use semaphores, others use window system messages), and the application is responsible for setting up and managing the event threads. The ECL simplifies all the event handling and makes it consistent for all event types. The application does not have to explicitly create multiple threads of execution, the ECL takes care of the threading internally.

However, you must be aware that the event procedures are called on a separate thread of execution. Access to dynamic application data must be synchronized when accessed from an event procedure. The event thread is spawned when the application registers for the event, and is terminated when the event is unregistered.

PS Connect/Disconnect and Multithreading

An EHLLAPI application must manage a connection to different sessions by calling ConnectPS and DisconnectPS EHLLAPI functions. The application must be carefully coded to avoid being connected to a session indefinitely because sessions have to be shared by all EHLLAPI applications. You must also ensure that an application is connected to a session before using certain other EHLLAPI functions.

The ECL does not require any explicit session connect or disconnect by the application. Each ECL object is associated with a particular connection (session) when it is constructed. To access different connections, the application only needs to create objects for each one. For example, the following example sends the keystrokes “ABC” to session A, then “DEF” to session B, and then the ENTER key to session A. In an EHLLAPI program, the application would have to connect/disconnect each of the sessions since it can interact with only one at a time. An ECL application can just use the objects in any order needed:

```
ECLPS           *PSA,  *PSB;

PSA = new ECLPS('A');
PSB = new ECLPS('B');

PSA->Sendkeys("ABC");
PSB->Sendkeys("DEF");
PSA->Sendkeys("[enter]");
```

For applications that interact with multiple connections (sessions), this can greatly simplify the code needed to manage the multiple connections.

In addition to the single working session, EHLLAPI also places constraints on the multithreaded nature of the application. Connecting to the presentation space and disconnecting from the presentation space has to be managed carefully when the application has more than one thread calling the EHLLAPI interface, and even with multiple threads the application can interact with only one session at a time.

The ECLPS does not impose any particular multithreading restrictions on applications. An application can interact with any number of sessions on any number of threads concurrently.

Chapter 2. Host Access Class Library C++

This C++ class library presents a complete object-oriented abstraction of a host connection that includes: reading and writing the host presentation space (screen), enumerating the fields on the screen, reading the Operator Indicator Area (OIA) for status information, accessing and updating information about the visual emulator window, transferring files, and performing asynchronous notification of significant events. The class libraries support IBM VisualAge C++ and Microsoft Visual C++ compilers.

The Host Access Class Library C++ layer consists of a number of C++ classes arranged in a class hierarchy. See Figure 2 that illustrates the C++ inheritence hierarchy of the Host Access Class Library C++ layer. Each object inherits from the class immediately above it in the diagram.

All the examples shown in this section of the document are supplied in the ECLSAMPS.CPP file. This file can be used to compile and execute any of the examples using any supported compiler.

© Copyright IBM Corp. 1997

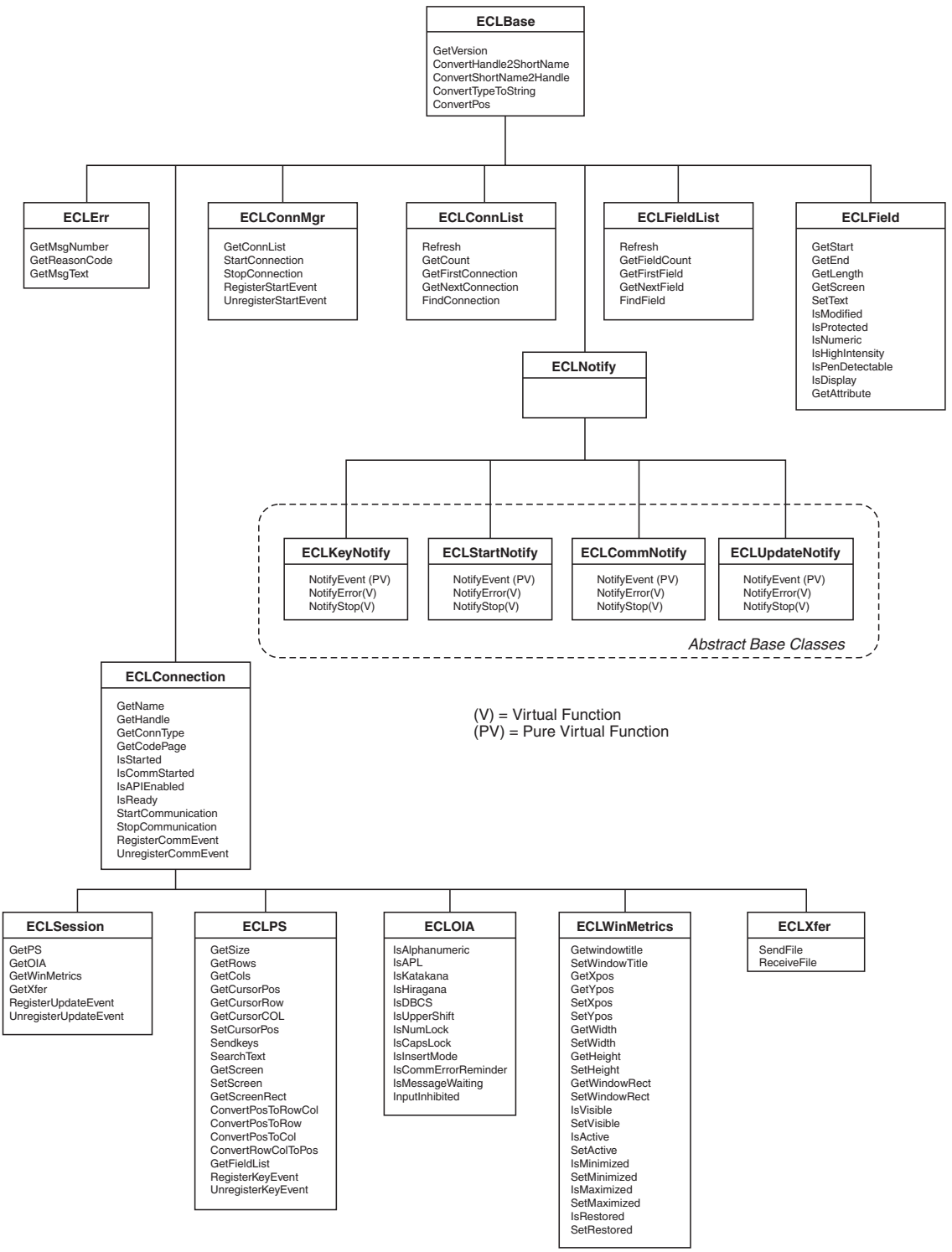


Figure 2. Host Access Class Objects

Figure 2 also shows all the member functions of each class. Note that in addition to the functions shown for each class, classes inherit all the functions of the parent class. For example, the function **IsReady**() is available on ECLSession, ECLPS, ECLOIA, ECLWinMetrics, and ECLXfer classes.

Each class is described briefly in the following sections. See the individual class descriptions in the C++ section of this document for more details.

The following is a brief overview of the Host Access Class Library C++ classes. Each class name begins with ECL, for Host Access Class Library.

ECLBase, on page 17 is the base class for all ECL objects. It provides some basic utility methods such as the conversion of connection names and handles. Because all ECL objects inherit from this class, these methods can be used on any ECL object.

ECLConnection, on page 23 represents a single Personal Communications connection and contains connection information such as the connection status, the type of connection (for example, 3270 or 5250), and the name and handle of the connection. This class is also the base class for all the connection-specific ECL objects such as ECLPS and ECLOIA.

ECLConnList, on page 37 contains a list of all the Personal Communications connections that were in existence at the time the object was created or the last time the Refresh method was called. Each connection is represented by an ECLConnection object.

ECLConnMgr, on page 45 enumerates all the currently running Personal Communications connections (windows) using the ECLConnList object. Is also provides methods for starting new connections and stopping connections.

ECLConnNotify, on page 53 is a notification class that an application can use to be notified whenever a connection is disconnected from or connected to a host. It can be used to monitor the status of a connection and take action when a connection is disconnected unexpectedly.

ECLErr, on page 59 provides a method for returning run-time error information from Host Access Class Library classes.

ECLField, on page 63 contains information about a single field on the screen, such as the field attributes, field color, position on the screen or length. A method is also supplied to update input fields.

ECLFieldList, on page 77 contains a collection of ECLField objects. When the Refresh method is called, the current host screen is examined, and the list of fields is extracted and used to build the list of ECLField objects. An application can use this collection to manage fields without having to build the list itself.

ECLKeyNotify, on page 84 is a notification class that an application can use to be notified of keystroke events. The application can filter (remove) keystrokes, replace them with other keystrokes or discard them.

ECLOIA, on page 90 provides access to operator status information such as shift indicators, input inhibited conditions and communications errors.

ECLPS, on page 101 represents the presentation space (screen) of a single connection. It contains methods for obtaining a copy of the screen contents in the form of data planes. Each plane represents a specific aspect of the presentation space, such as the text, field attributes and color attributes. Methods are provided for searching for strings in the presentation space, sending keystrokes to the host, getting and setting the host cursor position, and many other functions. Also provided is an ECLFieldList object that can be used to enumerate the list of fields on the screen.

ECLSession, on page 125 contains a collection of all the connection-specific objects. ECLSession can be used to easily create a complete set of objects for a particular connection.

ECLStartNotify, on page 133 is a notification class that an application can use to be notified whenever a connection is started or stopped. It can be used to monitor the status of the system and take action when a connection is closed unexpectedly.

ECLUpdateNotify, on page 138 is a notification class that an application can use to be notified whenever the host screen or OIA is updated.

ECLWinMetrics, on page 143 represents the physical window in which the emulation is running. Methods are provided for getting and setting the window state (min, max, restored), window size and visibility.

ECLXfer, on page 164 initiates file transfers to or from the host over the connection.

Building C++ ECL Programs

This section describes the mechanics of how to build a C++ program which uses the ECL. The source code preparation, compiling and linking requirements are described.

IBM Visual Age C++

The following sections describe how to prepare, compile, and link IBM VisualAge C++ applications that use the ECL. Personal Communications supports IBM VisualAge C++ Version 3.5 and later.

Source Code Preparattion

Programs that use ECL classes must include the ECL header files to obtain class definitions and other compile-time information. Although, it is possible to include only the subset of header files the application requires, for simplicity, it is recommended that applications include all ECL header files using the ECALL.HPP file.

Any C++ source file which contains references to ECL objects or definitions should have the following statement before the first reference:

```
#include "eclall.hpp"
```

Compiling

The compiler must be instructed to search the PCOMM subdirectory containing the ECL header files. This is done using the /I compiler option.

The application must be compiled for Multithreaded execution using the /Gm+ compiler option.

Linking

The linker must be instructed to include the ECL linkable library file (PCSECLVA.LIB). This is done by specifying the fully qualified name of the library file on the linker command line.

Executing

When an application that uses the ECL is executed, the PCOMM libraries must be found in the system path. By default, the PCOMM directory is added to the system path during PCOMM installation.

Example

The following MAKFILE is an example of how to build an IBM VisualAge C++ application using the ECL:

```
#------------------------------------------------------------
# Sample make file for IBM VisualAge C++
#------------------------------------------------------------

all:          sample.exe


pcomm = c:progra~1\person~1\samples

debug = /O- /Ti+
msgs   =  /Word+pro+ret+use+cmd
includes = -I $ (pcomm)

iccflags = /c /Gd- /Sm /Re /ss /Q /Gm+ $(msgs) $(debug) $(includes)


#------------------------------------------------------------
# General way to generate a ".obj" from a ".cpp
#------------------------------------------------------------
.cpp.obj:
   icc $(iccflags) $*.cpp


#------------------------------------------------------------
# Compile and link SAMPLE.CPP
#------------------------------------------------------------
sample.exe:        sample.obj
   ilink sample.obj \
         user32.lib kernel32.lib \
         $(pcomm) \pcseclva.lib \
        /DEBUG  /OUT:sample.exe

sample.obj:       sample.cpp
```

Microsoft Visual C++

The following sections describe how to prepare, compile, and link Microsoft Visual C++ applications that use the ECL. Personal Communications currently supports Microsoft Visual C++ Version 4.2.

Source Code Preparation

Programs that use ECL classes must include the ECL header files to obtain the class definitions and other compile-time information. Although it is possible to include only the subset of header files the application requires, for simplicity it is recommended that applications include all ECL header files using the ECLALL.HPP file.

Any C++ source file which contains references to ECL objects or definitions should have the following statement before the first reference:

```
#include  "eclall.hpp"
```

Compiling

The compiler must be instructed to search the PCOMM subdirectly containing the ECL header files. This is done using the /I compiler option, or the Developer Studio Project Setting dialog.

The application must be compiled for multithreaded execution by using the /MT (for executable files), or /MD (for DLLs) compiler options.

Linking

The linker must be instructed to include the ECL linkable library file (PCSECLVC.LIB). This is done by specifying the fully qualified name of the library file on the linker command line, or by using the Developer Studio Project Settings dialog.

Executing

When an application that uses the ECL is executed, the PCOMM libraries must be found in the system path. By default, the PCOMM directory is added to the system path during PCOMM installation.

ECLBase Class

ECLBase is the base class for all ECL objects. It provides some basic utility methods such as the conversion of connection names and handles. Because all ECL objects inherit from this class, these methods can be used on any ECL object.

An application should not create objects of this class directly.

Derivation

None

ECLBase Methods

The following shows the methods that are valid for ECLBase classes.

int GetVersion(void)
char ConvertHandle2ShortName(long ConnHandle)
long ConvertShortName2Handle(char ShortName)
void ConvertTypeToString(int ConnType,char *TypeString)
inline void ConvertPos(ULONG *Row, ULONG *Col, ULONG PSCols)

GetVersion

This method returns the version of the Host Access Class Library. The value returned is the decimal version number multiplied by 100. For example, version 1.02 would be returned as 102.

Prototype

int GetVersion(void)

Parameters

None

Return Value

int The ECL version number multiplied by 100.

Example

```
//-------------------------------------------------------------------
// ECLBase::GetVersion
//
// Display major version number of ECL library.
//-------------------------------------------------------------------
void Sample2() {

if (ECLBase::GetVersion() >= 200) {
   printf("Running version 2.0 or later.\n");
}
else {
   printf("Running version 1.XX\n");
}

} // end sample
```

ConvertHandle2ShortName

This method returns the name (A-Z) of the ECL connection handle specified. Note that this function may return a name even if the specified connection does not exist.

Prototype

char ConvertHandle2ShortName(long ConnHandle)

Parameters
long ConnHandle The handle of an ECL connection.

Return Value
char The name of the ECL connection in the range of 'A' to 'Z'.

Example
```
//-------------------------------------------------------------------
// ECLBase::ConvertHandle2ShortName
//
// Display name of first connection in the connection list.
//-------------------------------------------------------------------
void Sample3() {

ECLConnList   ConnList;
long Handle;
char Name;

if (ConnList.GetCount() > 0) {
   // Print connection name of first connection in the
   // connection list.
   Handle = ConnList.GetFirstConnection()->GetHandle();
   Name = ConnList.ConvertHandle2ShortName(Handle);
   printf("Name of first connection is: %c \n", Name);
}
else printf("There are no connections.\n");

} // end sample
```

ConvertShortName2Handle
This method returns the connection handle of the ECL connection with the specified name. The name must be in the range 'A' to 'Z'. Note that this function may return a handle even if the specified connection does not exist.

Prototype
char ConvertShortName2Handle(char Name)

Parameters
char Name The name of an ECL connection in the range of 'A' to 'Z'.

Return Value
long The handle of the ECL connection.

Example

```
//---------------------------------------------------------------------
// ECLBase::ConvertShortName2Handle
//
// Display handle of connection 'A'.
//---------------------------------------------------------------------
void Sample4() {

ECLConnList   ConnList;
long Handle;
char Name;

Name = 'A';
Handle = ConnList.ConvertShortName2Handle(Name);
printf("Handle of connection A is: 0x%lx \n", Handle);

} // end sample
```

ConvertTypeToString

This method converts a connection type returned by ECLConnection::GetConnType() into a null terminated string. The string returned is not language sensitive.

Prototype

void ConvertTypeToString(int ConnType,char *Buff)

Parameters

int ConnType The connection type and must be one of the HOSTTYPE_* constants defined in ECLBASE.HPP.

char *Buff The buffer of at least 21 bytes in which the string will be returned.

ConnType	Returned String
HOSTTYPE_3270DISPLAY	"3270 DISPLAY"
HOSTTYPE_3270PRINTER	"3270 PRINTER"
HOSTTYPE_5250 DISPLAY	"5250 PRINTER"
HOSTTYPE_5250PRINTER	"5250 PRINTER"
HOSTTYPE_VT	"ASCII TERMINAL"
HOSTTYPE_PC	"PC SESSION"
Any other value	"UNKNOWN"

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLBase::ConvertTypeToString
//
// Display type of connection 'A'.
//-------------------------------------------------------------------
void Sample5() {

ECLConnection   *pConn;
char                TypeString[21];

pConn = new ECLConnection('A');

pConn->ConvertTypeToString(pConn->GetConnType(), TypeString);
// Could also use:
// ECLBase::ConvertTypeToString(pConn->GetConnType(), TypeString);

printf("Session A is a %s \n", TypeString);

delete pConn;

} // end sample
```

Convert Pos

This method is an inline function (macro) to convert an ECL position coordinate into a row/column coordinate given a position and the width of the presentation space. This function is faster than using ECLPS::ConvertPosToRowCol() for applications that already know (or assume) the width of the presentation space.

Prototype

inline void ConvertPos(ULONG Pos,ULONG *Row,ULONG *Col,ULONG PSCols).

Parameters

ULONG Pos The linear positional coordinate to be converted (input).

ULONG *Row The pointer to the returned row number of the given position (output).

ULONG *Col The pointer to the returned column number of the given position (output).

ULONG *PSCols The number of columns in the host presentation space (input).

Return Value

None

Example

```
//---------------------------------------------------------------------
// ECLBase::ConvertPos
//
// Display row/column coordinate of a given point.
//---------------------------------------------------------------------
void Sample6() {

ECLPS        *pPS;
ULONG         NumRows, NumCols, Row, Col;

try {
   pPS = new ECLPS('A');

   pPS->GetSize(&NumRows, &NumCols);    // Get height and width of PS

   // Get row/column coordinate of position 81
   ECLBase::ConvertPos(81, &Row, &Col, NumCols);
   printf("Position 81 is row %lu, column %lu \n", Row, Col);

  delete  pPS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLConnection Class

ECLConnection contains connection-related information for a given connection. This object can be created directly by an application, and is also created indirectly by the ECLConnList object or when creating any object that inherits from ECLConnection (for example, ECLSession).

The information returned by the methods of this object are current as of the time the method is called.

ECLConnection is inherited by ECLSession, ECLPS, ECLOIA, ECLWinMetrics, and ECLXfer.

Derivation

ECLBase > ECLConnection

ECLConnection Methods

The following shows the methods that are valid for ECLConnection classes.

ECLConnection(char ConnName)
ECLConnection(long ConnHandle)
~ECLConnection()
long GetHandle()
int GetConnType()
char GetName()
BOOL IsStarted()
BOOL IsConnected()
BOOL IsAPIEnabled()
BOOL IsReady()
unsigned int GetCodePage()
void StartCommunication()
void StopCommunication()
void RegisterCommEvent
(ECLConnNotify *ConnNotifyClass)
void UnregisterCommEvent(ECLConnNotify *ConnNotifyClass)

ECLConnection Constructor

This method constructs an ECLConnection object from either a connection name or a handle.

Prototype

ECLConnection(long ConnHandle)

ECLConnection(char ConnName)

Parameters

long ConnHandle Handle of connection to create a connection object.

long ConnName Name (A-Z) of connection to create a connection object.

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLConnection::ECLConnection       (Constructor)
//
// Create two connection objects for connection 'A', one created
// by name, the other by handle.
//-------------------------------------------------------------------
void Sample7() {

ECLConnection          *pConn1, *pConn2;
long                   Hand;

try {
  pConn1  = new ECLConnection('A');
  Hand    = pConn1->GetHandle();
  pConn2  = new ECLConnection(Hand);  // Another ECLConnection for 'A'

   printf("Conn1 is for connection %c, Conn2 is for connection %c.\n",
            pConn1->GetName(),  pConn2->GetName());

   delete pConn1;    // Call destructors
  delete  pConn2;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLConnection Destructor

This method destroys an ECLConnection object.

Prototype

~ECLConnection()

Parameters

None

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLConnection:: ECLConnection       (Destructor)
//
// Create two connection objects, then delete both of them.
//-------------------------------------------------------------------
void Sample8() {

ECLConnection          *pConn1, *pConn2;
long                   Hand;

try {
  pConn1   = new  ECLConnection('A');
  Hand      = pConn1->GetHandle();
  pConn2   = new  ECLConnection(Hand);   // Another  ECLConnection  for  'A'

   printf("Conn1 is for connection %c, Conn2 is for connection %c.\n",
            pConn1->GetName(),  pConn2->GetName());

   delete pConn1;    // Call destructors
  delete  pConn2;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetHandle

This method returns the handle of the connection. This handle uniquely identifies the connection and may be used in other ECL functions that require a connection handle.

Prototype

long GetHandle()

Parameters

None

Return Value

long Connection handle of the ECLConnection object.

Example

The following example shows how to return the handle of the first connection in the connection list.

```
//---------------------------------------------------------------------
// ECLConnection::GetHandle
//
// Get the handle of connection 'A' and use it to create another
// connection object.
//---------------------------------------------------------------------
void Sample9() {

ECLConnection          *pConn1, *pConn2;
long                   Hand;

try {
  pConn1  = new ECLConnection('A');
  Hand    = pConn1->GetHandle();
  pConn2  = new ECLConnection(Hand);  // Another ECLConnection for 'A'

   printf("Conn1 is for connection %c, Conn2 is for connection %c.\n",
           pConn1->GetName(),  pConn2->GetName());

   delete pConn1;    // Call destructors
  delete  pConn2;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetConnType

This method returns the connection type. This connection type may change over time (for example, you may reconfigure the connection for a different host). The application should not assume the connection type is fixed. See below for connection types returned.

Note: The ECLBase::ConvertTypeToString function converts the connection type to a null terminated string.

Prototype

int GetConn Type()

Parameters

None

Return Value

int Connection type constant (HOSTTYPE_* from HOSTBASE.HPP). The following table shows the value returned and its meaning.

Value Returned	Meaning
HOSTTYPE_3270DISPLAY	3270 display
HOSTTYPE_3270PRINTER	3270 printer
HOSTTYPE_5250DISPLAY	5250 display
HOSTTYPE_5250PRINTER	5250 printer
HOSTTYPE_VT	ASCII VT display
HOSTTYPE_UNKNOWN	Unknown connection type

Example

The following example shows how use the GetConnType method to return the connection type.

```
//-------------------------------------------------------------------
// ECLConnection::GetConnType
//
// Find the first 3270 display connection in the current list of
// all connections.
//-------------------------------------------------------------------
void Sample10() {

ULONG           i;                      // Connection counter
ECLConnList ConnList;         // Connection list object
ECLConnection *Info=NULL; // Pointer to connection object

for (i=0; i<ConnList.GetCount(); i++) {

   Info = ConnList.GetNextConnection(Info);
   if (Info->GetConnType() == HOSTTYPE_3270DISPLAY) {
      // Found the first 3270 display connection
      printf("First 3270 display connection is '%c'.\n",
                  Info->GetName());
     return;
  }

} // for
printf("Found no 3270 display connections.\n");

} // end sample
```

GetName

This method returns the connection name (a single, alphabetic character from A-Z) of the connection. This name also corresponds to the EHLLAPI session ID.

Prototype

char GetName()

Parameters
None

Return Value
char Connection short name.

Example
The following example shows how to use the GetName method to return the connection name.

```
//---------------------------------------------------------------------
// ECLConnection::GetName
//
// Find the first 3270 display connection in the current list of
// all connections and display its name (PComm session ID).
//---------------------------------------------------------------------
void Sample11() {

ULONG          i;                    // Connection counter
ECLConnList ConnList;         // Connection list object
ECLConnection *Info=NULL; // Pointer to connection object

for (i=0; i<ConnList.GetCount(); i++) {

   Info = ConnList.GetNextConnection(Info);
   if (Info->GetConnType() == HOSTTYPE_3270DISPLAY) {
      // Found the first 3270 display connection, display the name
      printf("First 3270 display connection is '%c'.\n",
                  Info->GetName());
     return;
   }

} // for
printf("Found no 3270 display connections.\n");

} // end sample
```

IsStarted

This method indicates if the connection is started. A started connection may or may not be connected to a host. Use the IsCommStarted function to determine if the connection is currently connected to a host.

Prototype
BOOL IsStarted()

Parameters
None

Return Value

BOOL TRUE value if the connection is started; FALSE value if the connection is not started.

Example

```
//-------------------------------------------------------------------
// ECLConnection::IsStarted
//
// Display list of all started connections.    Note they may or may
// not be communications-connected to a host, and may or may not
// be visible on the screen.
//-------------------------------------------------------------------
void Sample12() {

ECLConnection *Info;          // Pointer to connection object
ECLConnList ConnList;              // Connection list object

// Print list of started connections

for (Info = ConnList.GetFirstConnection();
        Info != NULL;
        Info = ConnList.GetNextConnection(Info)) {

  if  (Info->IsStarted())
      printf("Connection %c is started.\n", Info->GetName());
}

} // end sample
```

IsCommStarted

This method indicates if the connection is currently connected to the host (for example, it indicates if host communications is active for the connection). This function returns a FALSE value if the connection is not started (see “IsStarted” on page 29).

Prototype

BOOL IsCommStarted()

Parameters

None

Return Value

BOOL TRUE value if the connection is connected to the host; FALSE value if the connection is not connected to the host.

Example

```
//-------------------------------------------------------------------
// ECLConnection::IsCommStarted
//
// Display list of all started connections which are currently
// in communications with a host.
//-------------------------------------------------------------------
void Sample13() {
```

```
ECLConnection *Info;        // Pointer to connection object
ECLConnList ConnList;           // Connection list object

for (Info = ConnList.GetFirstConnection();
       Info != NULL;
       Info = ConnList.GetNextConnection(Info)) {

  if  (Info->IsCommStarted())
      printf("Connection %c is connected to a host.\n", Info->GetName());
}

} // end sample
```

IsAPIEnabled

This method indicates if the connection is API-enabled. A connection that does not have API enabled cannot be used with the Host Access Class Library. This function returns a FALSE value if the connection is not started.

Prototype

BOOL IsAPIEnabled()

Parameters

None

Return Value

BOOL TRUE value if API is enabled; FALSE value if API is not enabled.

Example

```
//-------------------------------------------------------------------
// ECLConnection::IsAPIEnabled
//
// Display list of all started connections which have APIs enabled.
//-------------------------------------------------------------------
void Sample14() {

ECLConnection *Info;        // Pointer to connection object
ECLConnList ConnList;           // Connection list object

for (Info = ConnList.GetFirstConnection();
       Info != NULL;
       Info = ConnList.GetNextConnection(Info)) {

  if  (Info->IsAPIEnabled())
      printf("Connection %c has APIs enabled.\n", Info->GetName());
}

} // end sample
```

IsReady

This method indicates that the connection is ready, meaning the connection is started, connected, and API-enabled. This function is faster and easier than calling IsStarted, IsCommStarted, and IsAPIEnabled.

Prototype
BOOL IsReady()

Parameters
None

Return Value
BOOL TRUE if the connection is started, CommStarted, and API-enabled; FALSE if otherwise.

Example
```
//-------------------------------------------------------------------
// ECLConnection::IsReady
//
// Display list of all connections which are started, comm-connected
// to a host, and have APIs enabled.
//-------------------------------------------------------------------
void Sample15() {

ECLConnection *Info;         // Pointer to connection object
ECLConnList ConnList;             // Connection list object

for (Info = ConnList.GetFirstConnection();
        Info != NULL;
        Info = ConnList.GetNextConnection(Info)) {

>   if  (Info->IsReady())
       printf("Connection %c is ready (started, comm-connected, API
               enabled).\n",  Info->GetName());
}

} // end sample
```

GetCodePage

This method returns the host code page for which the connection is configured.

Prototype
unsigned int GetCodePage()

Parameters
None

Return Value

unsigned int Host code page of the connection.

Example

```
//-------------------------------------------------------------------
// ECLConnection::GetCodePage
//
// Display host code page for each ready connection.
//-------------------------------------------------------------------
void Sample16() {

ECLConnection *Info;        // Pointer to connection object
ECLConnList ConnList;           // Connection list object

for (Info = ConnList.GetFirstConnection();
       Info != NULL;
       Info = ConnList.GetNextConnection(Info)) {

  if  (Info->IsReady())
     printf("Connection %c is configured for host code page %u.\n",
                Info->GetName(),  Info->GetCodePage());
}

} // end sample
```

StartCommunication

This method connects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator communication menu and choosing Connect.

Prototype

void StartCommunication()

Parameters

None

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLConnection::StartCommunication
//
// Start communications link for any connection which is currently
// not comm-connected to a host.
//-------------------------------------------------------------------
void Sample17() {

ECLConnection *Info;        // Pointer to connection object
ECLConnList ConnList;           // Connection list object

for (Info = ConnList.GetFirstConnection();
```

```
        Info != NULL;
        Info = ConnList.GetNextConnection(Info)) {

   if (!(Info->IsCommStarted())) {
      printf("Starting comm-link for connection %c...\n", Info->GetName());
     Info->StartCommunication();
  }
}

} // end sample
```

StopCommunication

This methods disconnects the PCOMM emulator from the host data stream. This has the same effect as going to the PCOMM emulator communication menu and choosing Disconnect.

Prototype

void StopCommunication()

Parameters

None

Return Value

None

Example

```
//---------------------------------------------------------------------
// ECLConnection::StopCommunication
//
// Stop comm-link for any connection which is currently connected
// to a host.
//---------------------------------------------------------------------
void Sample18() {

ECLConnection *Info;         // Pointer to connection object
ECLConnList ConnList;            // Connection list object

for (Info = ConnList.GetFirstConnection();
        Info != NULL;
        Info = ConnList.GetNextConnection(Info)) {

   if (Info->IsCommStarted()) {
      printf("Stopping comm-link for connection %c...\n", Info->GetName());
     Info->StopCommunication();
  }
}

} // end sample
```

RegisterCommEvent

This member function registers an application object to receive notification of all communication link connect/disconnect events. To use this function, the application must create an object derivied from the ECLCommNotify class. A pointer to that object is then passed to this registration function.*Implementation Restriction:* An application can register only one object for communication event notification.

After a notify object has been registered with this function, it will be called whenever the connections communications link with the host connects or disconnects. The object will receive notification for all communication events whether they are caused by the StartCommunications() function or explicitly by the user. This event should not be confused with the connection start/stop event which is triggered when a new PCOMM connection starts or stops.

The optional InitEvent parameter causes an initial event to be generated when the object is registered. This can be useful to syncronize an event object with the current state of the communications link. If InitEvent is specified as FALSE, no initial event is generated when the object is registered. The default for this parameter is TRUE.

The application must call UnregisterCommEvent() before destroying the notification object. The object is automatically unregistered if the ECLConnection object where it is registered is destroyed.

See the description of “ECLCommNotify Class” on page 53 for more information.

Prototype

void RegisterCommEvent(ECLCommNotify *NotifyObject, BOOL InitEvent = TRUE)

Parameters

ECLCommNotify *AppObject Pointer to an object derived from ECLCommNotify class.

BOOL InitEvent Generate an initial event with the current state.

Return Value

None

Example

See “ECLCommNotify Class” on page 53 for an example of ECLConnection::RegisterCommEvent.

UnregisterCommEvent

This member function unregisters an application object previously registered for communication events with the RegisterCommEvent() function. A registered application notify object should not be destroyed without first calling this function to unregister it. If there is no notify object currently registered, or the registered object is not the *NotifyObject* passed in, this function does nothing (no error is thrown).

When a notify object is unregistered, its NotifyStop() member function will be called.

See the description of “ECLCommNotify Class” on page 53 for more information.

Prototype
void UnregisterCommEvent(ECLCommNotify *NotifyObject)

Parameters
ECLCommNotify *AppObject This is a currently registered application notification object.

Return Value
None

Example
See “ECLCommNotify Class” on page 53 for an example of ECLConnection::UnregisterCommEvent.

ECLConnList Class

ECLConnList obtains information about all host connections on a given machine. An ECLConnList object contains a collection of all the connections that are currently known in the system.

The ECLConnList object contains a collection of ECLConnection objects. Each element of the collection contains information about a single connection. A connection in this list may be in any state (for example, stopped or disconnected). All started connections appear in this list. The ECLConnection object contains the state of the connection.

The list is a snapshot of the set of connections at the time this object is created, or the last time the Refresh method was called. The list is not dynamically updated as connections are started and stopped. An application can use the RegisterStartEvent member of the ECLConnMgr object to be notified of connection start and stop events.

An ELCConnList object may be created directly by the application or indirectly by the creation of an ECLConnMgr object.

Derivation

ECLBase>ECLConnList

Usage Notes

An ECLConnList object provides a static snapshot of current connections. The Refresh method is automatically called upon construction of the ECLConnList object. If you use the ECLConnList object right after construction it contains an accurate representation of the list of connections at that moment. However, you should call the Refresh method in the ECLConnList object before you start accessing it if some time has passed since its construction.

The application can iterate over the collection by using the GetFirstConnection and GetNextConnection methods. The object pointers returned by GetFirstConnection and GetNextConnection are valid only until the Refresh member is called, or the ECLConnList object is destroyed. The application can locate a specific connection of interest in the list using the FindConnection function. Like GetNextConnection, the returned pointer is valid only until the next Refresh or the ECLConnList object is destroyed.

The order of connections in the connection list is undefined. An application should not make any assumptions about the list order. The order of connections in the list does not change until the Refresh function is called.

An ECLConnList object is automatically created when an ECLConnMgr object is created. However, the ECLConnList object can be created without an ECLConnMgr object.

ECLConnList Methods

The following section describes the methods that are valid for the ECLConnList class.

ECLConnection * GetFirstConnection()
ECLConnection * GetNextConnection(ECLConnection *Prev)
ECLConnection * FindConnection(Long ConnHandle)
ECLConnection * FindConnection(char ConnName)
long GetCount()
void Refresh()

ECLConnList Constructor

This method creates an ECLConnList object and initializes it with the current list of connections.

Prototype

ECLConnList();

Parameters

None

Return Value

None

Example

```
//---------------------------------------------------------------------
// ECLConnList::ECLConnList              (Constructor)
//
// Dynamically construct a connection list object, display number
// of connections in the list, then delete the list.
//---------------------------------------------------------------------
void Sample19() {

ECLConnList *pConnList;    // Pointer to connection list object

try {
   pConnList = new ECLConnList();
   printf("There are %lu connections in the connection list.\n",
            pConnList->GetCount());

   delete pConnList;    // Call destructor
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLConnList Destructor

This method destroys an ECLConnList object.

Prototype

~ECLConnList()

Parameters

None

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLConnList:: ECLConnList              (Destructor)
//
// Dynamically construct a connection list object, display number
// of connections in the list, then delete the list.
//-------------------------------------------------------------------
void Sample20() {

ECLConnList *pConnList;    // Pointer to connection list object

try {
   pConnList = new ECLConnList();
   printf("There are %lu connections in the connection list.\n",
            pConnList->GetCount());

   delete pConnList;    // Call destructor
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetFirstConnection

The GetFirstConnection method returns a pointer to the first connection information object in the ECLConnList collection. See “ECLConnection Class” on page 23 for details on its contents. The returned pointer becomes invalid when the ECLConnList Refresh member is called or the ECLConnList object is destroyed. The application should not delete the returned object. If there are no connections in the list, NULL is returned.

Prototype

ECLConnection *GetFirstConnection()

Parameters
None

Return Value
ECLConnection * Pointer to the first ECLConnection object in the list. If there are no connections in the list, null is returned.

Example
```
//-------------------------------------------------------------------
// ECLConnection::GetFirstConnection
//
// Iterate over list of connections and display information about
// each one.
//-------------------------------------------------------------------
void Sample21() {

ECLConnection *Info;         // Pointer to connection object
ECLConnList ConnList;             // Connection list object
char TypeString[21];         // Type of connection

for (Info = ConnList.GetFirstConnection();             // Get first one
        Info != NULL;                                          // While there is one
        Info = ConnList.GetNextConnection(Info)) {    // Get next one

  ECLBase::ConvertTypeToString(Info->GetConnType(),  TypeString);
   printf("Connection %c is a %s type connection.\n",
             Info->GetName(),  TypeString);
}

} // end sample
```

GetNextConnection
This method returns a pointer to the next connection information object in the ECLConnList collection given a connection in the list. The application supplies a pointer to a connection previously returned by this function or GetFirstConnection. See “ECLConnection Class” on page 23 for details on its contents. The returned pointer is not valid after the next ECLConnList Refresh() call, or the ECLConnList object is destroyed. A NULL pointer is returned if there is an attempt to read past the end of the list. Successive calls to this method (supplying the prior pointer on each call) iterates over the list of connections. After the last connection is returned, subsequent calls return a NULL pointer. The first connection in the list can be obtained by supplying NULL for the previous connection.

Prototype
ECLConnection *GetNext Connection (ECLConnection *Prev)

Parameters

ECLConnection *Prev Pointer returned by prior call to this function, GetFirstConnection(), or NULL.

Return Value

ECLConnInfo * This is the pointer to the next ECLConnection object, or NULL if end of list.

Example

```
//-------------------------------------------------------------------
// ECLConnection::GetNextConnection
//
// Iterate over list of connections and display information about
// each one.
//-------------------------------------------------------------------
void Sample22() {

ECLConnection *Info;         // Pointer to connection object
ECLConnList ConnList;             // Connection list object
char TypeString[21];         // Type of connection

for (Info = ConnList.GetFirstConnection();             // Get first one
       Info != NULL;                                          // While there is one
       Info = ConnList.GetNextConnection(Info)) {    // Get next one

  ECLBase::ConvertTypeToString(Info->GetConnType(),  TypeString);
   printf("Connection %c is a %s type connection.\n",
            Info->GetName(),  TypeString);
}

} // end sample
```

FindConnection

This method searches the current connection list for the connection specified. The desired connection can be specified by handle or by name. There are two signatures for the FindConnection method. If the specified connection is found, a pointer to the ECLConnection object is returned. If the specified connection is not in the list, NULL is returned. The list is not automatically refreshed by this function; if a new connection has started since the list was constructed or refreshed it is not found.The returned pointer is to an object in the connection list maintained by the ECLConnList object. The returned pointer is invalid after the next ECLConnList::Refresh call or the ECLConnList object is destroyed.

Prototype

```
ECLConnection *FindConnection(Long ConnHandle),

ECLConnection *FindConnection(char ConnName)
```

Parameters

Long ConnHandle Handle of the connection to find in the list.

char ConnName Name of the connection to find in the list.

Return Value

ECLConnection * Pointer to the requested ECLConnection object. If the specified connection is not in the list, NULL is returned.

Example

```
//-------------------------------------------------------------------
// ECLConnection::FindConnection
//
// Find connection 'B' in the list of connections.    If found, display
// its type.
//-------------------------------------------------------------------
void Sample23() {

ECLConnection *Info;          // Pointer to connection object
ECLConnList ConnList;              // Connection list object
char TypeString[21];          // Type of connection

Info = ConnList.FindConnection('B');    // Find connection by name
if (Info != NULL) {

  ECLBase::ConvertTypeToString(Info->GetConnType(),  TypeString);
   printf("Connection 'B' is a %s type connection.\n",
              TypeString);
}
else printf("Connection 'B' not found.\n");

} // end sample
```

GetCount

This method returns the number of connections currently in the ECLConnList collection.

Prototype

ULONG GetCount()

Parameters

None

Return Value

ULONG Number of connections in the collection.

Example

```
//-------------------------------------------------------------------
// ECLConnList::GetCount
//
// Dynamically construct a connection list object, display number
// of connections in the list, then delete the list.
//-------------------------------------------------------------------
void Sample24() {

ECLConnList *pConnList;    // Pointer to connection list object

try {
   pConnList = new ECLConnList();
   printf("There are %lu connections in the connection list.\n",
            pConnList->GetCount());

   delete pConnList;    // Call destructor
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

Refresh

This method updates the ECLConnList collection with a list of all currently known connections in the system. All pointers previously returned by GetNextConnection, GetFirstConnection and FindConnection become invalid.

Prototype

void Refresh()

Parameters

None

Return Value

None

Example

```
//--------------------------------------------------------------------
// ECLConnection::Refresh
//
// Loop-and-wait until connection 'B' is started.
//--------------------------------------------------------------------
void Sample25() {

ECLConnection *Info;          // Pointer to connection object
ECLConnList ConnList;              // Connection list object
int i;

printf("Waiting up to 60 seconds for connection B to start...\n");
for (i=0; i<60; i++) {       // Limit wait to 60 seconds
   ConnList.Refresh();        // Refresh the connection list
   Info = ConnList.FindConnection('B');
   if ((Info != NULL) && (Info->IsStarted())) {
      printf("Connection B is now started.\n");
     return;
  }
   Sleep(1000L);                  // Wait 1 second and try again
}

printf("Connection 'B' not started after 60 seconds.\n");

} // end sample
```

ECLConnMgr Class

ECLConnMgr manages all Personal Communications connections on a given machine. It provides methods relating to the management of connections such as starting and stopping connections. It also creates an ECLConnList object to enumerate the list of all known connections on the system (see “ECLConnList Class” on page 37).

Derivation

ECLBase>ECLConnMgr

ECLConnMgr Methods

The following shows the methods that are valid with the ECLConnMgr class.

ECLConnMgr()
~ECLConnMgr()
ECLConnList * GetConnList()
void StartConnection(char *ConfigParms)
void StopConnection(Long ConnHandle, char *StopParms)
void RegisterStartEvent(ECLStartNotify *NotifyObject)
void UnregisterStartEvent(ECLStartNotify *NotifyObject)

ECLConnMgr Constructor

This method constructs an ECLConnMgr object.

Prototype

ECLConnMgr()

Parameters

None

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLConnMgr::ECLConnMgr                (Constructor)
//
// Create a connection mangager object, start a new connection,
// then delete the manager.
//-------------------------------------------------------------------
void Sample26() {

ECLConnMgr                    *pCM; // Pointer to connection manager object

try {
   pCM = new ECLConnMgr();    // Create connection manager
  pCM->StartConnection("profile=coax  connname=e");
   printf("Connection 'E' started with COAX profile.\n");
   delete pCM;                    // Delete connection manager
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLConnMgr Deconstructor

This method destroys an ECLConnMgr object.

Prototype
~ECLConnMgr()

Parameters
None

Return Value
None

Example
```
//-------------------------------------------------------------------
// ECLConnMgr:: ECLConnMgr                (Destructor)
//
// Create a connection mangager object, start a new connection,
// then delete the manager.
//-------------------------------------------------------------------
void Sample27() {

ECLConnMgr                  *pCM; // Pointer to connection manager object

try {
   pCM = new ECLConnMgr();    // Create connection manager
  pCM->StartConnection("profile=coax  connname=e");
   printf("Connection 'E' started with COAX profile.\n");
   delete pCM;                  // Delete connection manager
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetConnList

This method returns a pointer to an ECLConnList object. See “ECLConnList Class” on page 37 for more information. The ECLConnList object is destroyed when the ECLConnMgr object is destroyed.

Prototype
ECLConnList * GetConnList()

Parameters
None

Return Value
ECLConnList * Pointer to an ECLConnList object

Example
```
//-------------------------------------------------------------------
// ECLConnMgr::GetConnList
//
// Use connection manager's connection list object to display
// number of connections (see also ECLConnList::GetCount).
//-------------------------------------------------------------------
void Sample28() {

ECLConnMgr                    CM;  // Connection  manager  object

printf("There are %lu connections in the connection list.\n",
        CM.GetConnList()->GetCount());

} // end sample
```

Start Connection

This method starts a new Personal Communications emulator connection. The ConfigParms string contains connection configuration information as explained under “Usage Notes.”

Prototype
void StartConnection(char *ConfigParms)

Parameters
char *ConfigParms Null terminated connection configuration string.

Return Value
None

Usage Notes
The connection configuration string is implementation-specific. Different implementations of the Host Access Class Library may require different formats or information in the configuration string. This call is asynchronous in nature; the new connection may not yet be started when this call returns. An application can use the RegisterStartEvent function to be notified when a connection starts.

For Personal Communications, the configuration string has the following format:

PROFILE=["]<filename>["] [CONNNAME=<c>] [WINSTATE=<MAX|MIN|RESTORE|HIDE>]

Optional parameters are enclosed in square brackets []. The parameters are separated by at least one blank. Parameters may be in upper, lower, or mixed case and may appear in any order. The meaning of each parameter is as follows:

PROFILE=<filename>: Names the Personal Communications workstation profile (.WS file) that contains the connection configuration information. This parameter is not optional; a profile name must be supplied. If the file name contains blanks, the name must be enclosed in double quotation marks. The <filename>

value may be either the profile name with no extension, the profile name with the .WS extension, or the fully-qualified profile name path.

CONNNAME=<c> specifies the connection name (EHLLAPI short session ID) of the new connection. This value must be a single, alphabetic character (A-Z). If this value is not specified, the next available connection name is assigned automatically. If a connection already exists with the specified name an error is thrown (ERRMAJ_INVALID_SESSION).

WINSTATE=<MAX|MIN|RESTORE|HIDE> specifies the initial state of the emulator window. The default if this parameter is not specified is RESTORE.

Note: Due to the asynchronous nature of this call, it is possible for this function to return without error, but the connection fails to start. For example, if two connections are started in a short period of time with the same connection name the second StartConnection does not fail because the first connection has not yet started. However, when the second connection finally attempts to register its name it does fail to start because the name is already in use by the first connection. To minimize this possibility, connections should be started without specifying the CONNNAME parameter if possible.

Example

The following is an example of the StartConnection method.

```
ECLConnMgr      Manager();              // Connection  manager  object

// Start a host connection "E" and check for errors

try {
  Manager.StartConnection("profile=coax  connname=e");
}
catch (ECLErr Error) {
   MessageBox(NULL, Error.GetMsgText(), "Session start error!", MB_OK);
}
```

StopConnection

This method stops (terminates) the emulator connection identified by the connection handle. See “Usage Notes” on page 50 for contents of the StopParms string.

Prototype

void StopConnection(Long ConnHandle, char *StopParms)

Parameters

Long ConnHandle Handle of the connection to be stopped.

char * StopParms Null terminated connection stop parameter string.

Return Value

None

Usage Notes

The connection stop parameter string is implementation-specific. Different implementations of the Host Access Class Library may require a different format and contents of the parameter string. For Personal Communications the string has the following format:

```
[SAVEPROFILE=<YES|NO|DEFAULT>]
```

Optional parameters are enclosed in square brackets []. The parameters are separated by at least one blank. Parameters may be in upper, lower, or mixed case and may appear in any order. The meaning of each parameter is as follows:

SAVEPROFILE=<YES|NO|DEFAULT> controls the saving of the current connection configuration back to the workstation profile (.WS file). This causes the profile to be updated with any configuration changes you may have made during the connection. If NO is specified, the connection is stopped and the profile is not updated. If YES is specified, the connection is stopped and the profile is updated with the current (possibly changed) configuration. If DEFAULT is specified, the update option is controlled by the File->Save On Exit emulator menu option. If this parameter is not specified, DEFAULT is used.

Example

```
//-------------------------------------------------------------------
// ECLConnMgr::StopConnection
//
// Stop the first connection in the connection list.
//-------------------------------------------------------------------
void Sample29() {

ECLConnMgr                    CM;  // Connection  manager  object

if (CM.GetConnList()->GetCount() > 0) {

   printf("Stopping connection %c.\n",
            CM.GetConnList()->GetFirstConnection()->GetName());

  CM.StopConnection(
     CM.GetConnList()->GetFirstConnection()->GetHandle(),
     "saveprofile=no");
}
else printf("No connections to stop.\n");


} // end sample
```

RegisterStartEvent

This method registers an application object to receive notification of all connection start and stop events. To use this function, the application must create an object derived from the ECLStartNotify class. A pointer to that object is then passed to this registration function. *Implementation Restriction:* An application can register only one object for connection start or stop notification.

After a notify object has been registered with this function, it is called whenever a Personal Communications connection is started or stopped. The object receives notification for all connections whether they are started by the StartConnection function or explicitly by you. This event should not be confused with the start/stop Communication event, which is triggered when a connection connects or disconnects from a host system.

See “ECLStartNotify Class” on page 133 for more information.

Prototype
void RegisterStartEvent(ECLStartNotify *NotifyObject)

Parameters
ECLStartNotify *NotifyObject Pointer to object derived from the ECLStartNotify class.

Return Value
None

Example
```
//------------------------------------------------------------------
// ECLConnMgr::RegisterStartEvent
//
// See “ECLStartNotify Class” on page 133 for example of this method.
//------------------------------------------------------------
```

UnregisterStartEvent
This method unregisters an application object previously registered for connection start or stop events with the RegisterStartEvent function. A registered application notify object should not be destroyed without first calling this function to unregister it. If there is no notify object currently registered, or the registered object is not the NotifyObject passed in, this function does nothing (no error is thrown).

When a notify object is unregistered, its NotifyStop method is called.

See “ECLStartNotify Class” on page 133 for more information.

Prototype
void UnregisterStartEvent(ECLStartNotify *NotifyObject)

Parameters
None

Return Value
None

Example

```
//-------------------------------------------------------------------
// ECLConnMgr::UnregisterStartEvent
//
// See “ECLStartNotify Class” on page 133 for example of this method.
//-------------------------------------------------------------------
```

ECLCommNotify Class

ECLCommNotify is an abstract base class. An application cannot create an instance of this class directly. To use this class, the application must define its own class which is derived from ECLCommNotify. The application must implement the NotifyEvent() member function in its derived class. It may also optionally implement NotifyError() and NotifyStop() member functions.

The ECLCommNotify class is used to allow an application to be notified of communications connect/disconnect events on a PComm connection. Connect/disconnect events are generated whenever a PComm connection (window) is connected or disconnected from a host system.

To be notified of communications connect/disconnect events, the application must perform the following steps:

1. Define a class derived from ECLCommNotify.
2. Implement the derived class and implement the NotifyEvent() member function.
3. Optionally implement the NotifyError() and/or NotifyStop() functions.
4. Create an instance of the derived class.
5. Register the instance with the ECLConnection::RegisterCommEvent() function.

The example shown demonstrates how this may be done. When the above steps are complete, each time a connection's communications link is connected or disconnected from a host, the applications NotifyEvent() member function will be called.

If an error is detected during event generation, the NotifyError() member function is called with an ECLErr object. Events may or may not continue to be generated after an error, depending on the nature of the error. When event generation terminates (either due to an error, by calling the ECLConnection::UnregisterCommEvent, or by destruction of the ECLConnection object) the NotifyStop() member function is called. However event notification is terminated, the NotifyStop() member function is always called, and the application object is unregistered.

If the application does not provide an implementation of the NotifyError() member function, the default implementation is used (a simple message box is displayed to the user). The application can override the default behavior by implementing the NotifyError() function in the applications derived class. Likewise, the default NotifyStop() function is used if the application does not provide this function (the default behavior is to do nothing).

Note that the application can also choose to provide its own constructor and destructor for the derived class. This can be useful if the application wants to store some instance-specific data in the class and pass that information as a parameter on the constructor. For example, the application may want to post a message to an application window when a communications event occurs. Rather than define the window handle as a global variable (so it would be visible to the NotifyEvent() function), the application can define a constructor for the class which takes the window handle and stores it in the class member data area.

The application must not destroy the notification object while it is registered to recieve events.

Implementation Restriction: Currently the ECLConnection object allows only one notification object to be registered for communications event notification. The ECLConnection::RegisterCommEvent will throw an error if a notify object is already registered for that ECLConnection object.

Derivation

ECLBase>ECLNotify>ECLCommNotify

Usage Notes

```
//-------------------------------------------------------------------
// ECLCommNotify class
//
// This sample demonstrates the use of:
//
// ECLCommNotify::NotifyEvent
// ECLCommNotify::NotifyError
// ECLCommNotify::NotifyStop
// ECLConnection::RegisterCommEvent
// ECLConnection::UnregisterCommEvent
//-------------------------------------------------------------------


 //.....................................................................
// Define a class derived from ECLCommNotify
//.....................................................................
class MyCommNotify: public ECLCommNotify
{
public:
   // Define my own constructor to store instance data
  MyCommNotify(HANDLE  DataHandle);

   // We have to implement this function
   void       NotifyEvent(ECLConnection *ConnObj, BOOL Connected);

   // We choose to implement this function
   void NotifyStop (ECLConnection *ConnObj, int Reason);

   // We will take the default behaviour for this so we
   // don't implement it in our class:
   // void NotifyError (ECLConnection *ConnObj, ECLErr ErrObject);

private:
   // We will store our application data handle here
  HANDLE  MyDataH;
};

//.....................................................................
void MyCommNotify::NotifyEvent(ECLConnection *ConnObj,
                               BOOL  Connected)
//
// This function is called whenever the communications link
// with the host connects or disconnects.
//
```

```
// For this example, we will just write a message.    Note that we
// have access the the MyDataH handle which could have application
// instance data if we needed it here.
//
// The ConnObj pointer is to the ECLConnection object upon which
// this event was registered.
//.....................................................................
{
  if  (Connected)
      printf("Connection %c is now connected.\n", ConnObj->GetName());
  else
      printf("Connection %c is now disconnected.\n", ConnObj->GetName());

  return;
}

 //.....................................................................
MyCommNotify::MyCommNotify(HANDLE DataHandle)      // Constructor
//.....................................................................
{
   MyDataH = DataHandle;    // Save data handle for later use
}

//.....................................................................
void MyCommNotify::NotifyStop(ECLConnection *ConnObj,
                              int  Reason)
//.....................................................................
{
   // When notification ends, display message
   printf("Comm link monitoring for %c stopped.\n", ConnObj->GetName());
}

 //.....................................................................
// Create the class and start notification on connection 'A'.
//.....................................................................
void Sample30() {

ECLConnection *Conn;       // Ptr to connection object
MyCommNotify *Event;       // Ptr to my event handling object
HANDLE  InstData;          // Handle to application data block (for example)

try {
   Conn = new ECLConnection('A');              // Create connection obj
   Event = new MyCommNotify(InstData);       // Create event handler

   Conn->RegisterCommEvent(Event);             // Register for comm events

   // At this point, any comm link event will cause the
   // MyCommEvent::NotifyEvent() function to execute.    For
   // this sample, we put this thread to sleep during this
  //  time.

   printf("Monitoring comm link on 'A' for 60 seconds...\n");
  Sleep(60000);

   // Now stop event generation.    This will cause the NotifyStop
   // member to be called.
  Conn->UnregisterCommEvent(Event);
```

```
    delete Event;    // Don't delete until after unregister!
   delete  Conn;
}
catch (ECLErr Err) {
    printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLCommNotify Methods

The following section describes the methods that are valid for the ECLCommNotify class.

```
                ECLCommNotify()
               ~ECLCommNotify()
virtual void NotifyEvent (
                ECLConnection *ConnObj,
                BOOL Connected) = 0
virtual void NotifyError (
                ECLConnection *ConnObj,
                ECLErr ErrObject)
virtual void NotifyStop  (
                ECLConnection *ConnObj,
                int Reason)
```

NotifyEvent

This method is a “pure virtual” member function (the application *must* implement this function in classes derived from ECLCommNotify). This function is called whenever a connection starts or stops and the object is registered for start/stop events. The Connected BOOL is TRUE if the communications link is connected, or FALSE if it is not connected to the host.

Parameters

ECLConnection *ConnObj This is the pointer to ECLConnection object where the event occurred.

BOOL Connected This is TRUE if comm link is connected and FALSE if disconnected.

Return Value

None

NotifyError

This method is called whenever the ECLConnection object detects an error during event generation. The error object contains information about the error (see the ECLErr class description). Events may continue to be generated after the error, depending on the nature of the error. If the event generation stops due to an error, the NotifyStop() function is called. An application can choose to implement this function or allow the ECLCommNotify base class to handle the error. The base class will display the error in a message box using the text supplied by the ECLErr::GetMsgText() function. If the application implements this function in its derived class, it will override the base class function.

Parameters

ECLConnection *ConnObj This is the pointer to ECLConnection object in which the error occurred.

ECLErr ErrObject This is the ECLErr object describing the error.

Return Value
None

NotifyStop

This method is called when event generation is stopped for any reason (for example, due to an error condition or a call to ECLConnection::UnregisterCommEvent, etc.).

Implementation Note: the reason code is currently unused and will be zero.

Parameters
ECLConnection *ConnObj This is the ptr to ECLConnection object that is stopping notification.

int Reason This is unused (zero).

Return Value
None

ECLErr Class

The ECLErr class provides a method of returning run-time error information from Host Access Class Library classes. In error situations, ECLErr objects are created and populated with error and diagnostic information. The ECLErr objects are then thrown as C++ exceptions. The error and diagnostic information can then be queried from the caught ECLErr object.

Applications should not create or throw ECLErr objects directly.

Derivation

ECLBase>ECLErr

ECLErr Methods

The following section describes the methods that are valid for the ECLErr class.

const int GetMessageNumber()
const int GetReasonCode()
const char *GetMsgText()

GetMsgNumber

This method returns the message number that was set when this ECLErr object was created. Error message numbers are described in ERRORIDS.HPP.

Prototype

const int GetMsgNumber()

Parameters

None

Return Value

const int The error message number.

Example

```
//---------------------------------------------------------------------
// ECLErr::GetMsgNumber
//
// Cause an 'invalid parameters' error and tryp the ECL exception.
// The extract the error number and language-sensative text.
//---------------------------------------------------------------------
void Sample31() {

ECLPS      *PS = NULL;

try {
    PS = new ECLPS('A');
  PS->SetCursorPos(999,999);   //  Invalid  parameters
}
catch (ECLErr ErrObj) {
    printf("The following ECL error was trapped:\n");
    printf("%s \nError number: %lu\nReason code: %lu\n",
       ErrObj.GetMsgText(),
       ErrObj.GetMsgNumber(),
       ErrObj.GetReasonCode());
}

if (PS != NULL)
   delete  PS;

} // end sample
```

GetReasonCode

This method gets the reason code (sometimes referred to as the secondary or minor return code) from the ECLErr object. This code is generally used for debugging and diagnostic purposes. It is subject to change in future versions of the Host Access Class Library and should not be used programmatically. Descriptions of the reason codes can be found in ERRORIDS.HPP.

Prototype

const int GetReasonCode()

Parameters

None

Return Value

const int The ECLErr reason code.

Example

```
//---------------------------------------------------------------------
// ECLErr::GetReasonCode
//
// Cause an 'invalid parameters' error and tryp the ECL exception.
// The extract the error number and language-sensative text.
//---------------------------------------------------------------------
void Sample32() {

ECLPS      *PS = NULL;

try {
    PS = new ECLPS('A');
  PS->SetCursorPos(999,999);   //  Invalid  parameters
}
catch (ECLErr ErrObj) {
    printf("The following ECL error was trapped:\n");
    printf("%s \nError number: %lu\nReason code: %lu\n",
      ErrObj.GetMsgText(),
      ErrObj.GetMsgNumber(),
      ErrObj.GetReasonCode());
}

if (PS != NULL)
  delete  PS;

} // end sample
```

GetMsgText

This method returns the message text associated with the error code used to create this ECLErr object. The message text is returned in the language for which Personal Communications is currently installed.

Note: The returned pointer is invalid after the ECLErr object is deleted.

Prototype
const char *GetMsgText()

Parameters
None

Return Value
char * The message text associated with the error code that is part of this ECLErr object.

Example
```
//---------------------------------------------------------------------
// ECLErr::GetMsgText
//
// Cause an 'invalid parameters' error and tryp the ECL exception.
// The extract the error number and language-sensative text.
//---------------------------------------------------------------------
void Sample33() {

ECLPS      *PS = NULL;

try {
   PS = new ECLPS('A');
  PS->SetCursorPos(999,999);   //  Invalid  parameters
}
catch (ECLErr ErrObj) {
   printf("The following ECL error was trapped:\n");
   printf("%s \nError number: %lu\nReason code: %lu\n",
      ErrObj.GetMsgText(),
      ErrObj.GetMsgNumber(),
      ErrObj.GetReasonCode());
}

if (PS != NULL)
  delete  PS;

} // end sample
```

Usage Notes
The message text is retrieved from the Personal Communications message facility.

ECLField Class

ECLField contains information for a given field in an ECLFieldList object contained by an ECLPS object. An application should not create an object of this type directly. ECLField objects are created indirectly by the ECLFieldList object.

An ECLField object describes a single field of the host presentation space. It has methods for querying various attributes of the field and for updating the text of the field (for example, modifying the field text). Field attributes cannot be modified.

Derivation

ECLBase>ECLField

ECLField Methods

The following section describes the methods that are valid for the ECLField class.

ULONG GetStart()
void GetStart(ULONG *RowULONG *Col)
ULONG GetStartRow()
ULONG GetStartCol()
ULONG GetEnd()
void GetEnd(ULONG *RowULONG *Col)
ULONG GetEndRow()
ULONG GetEndCol()
ULONG GetLength()
ULONG GetText(char *Buff, ULONG BuffLen, PS_PLANE Plane = TextPlane)
void SetText(char *text)
BOOL IsModified()
BOOL IsProtected()
BOOL IsNumeric()
BOOL IsHighIntensity()
BOOL IsPenDetectable()
BOOL IsDisplay()
unsigned charGetAttribute()

GetStart

This method returns the position in the presentation space of the first character of the field. There are two signatures for the GetStart method. ULONG GetStart returns the position as a linear value with the upper left corner of the presentation space being “1”. void GetStart(ULONG *Row, ULONG *Col) returns the position as a row and column coordinate.

Prototype

ULONG GetStart(),

void GetStart(ULONG *Row, ULONG *Col)

Parameters

ULONG *Row This output parameter is a pointer to the row value to be updated.

ULONG *Col This output parameter is a pointer to the column value to be updated.

Return Value

ULONG Position in the presentation space represented as a linear array.

Example

The following example shows how to return the position in the presentation space of the first character of the field.

```
/--------------------------------------------------------------------
// ECLField::GetStart
//
// Iterate over list of fields and print each field
// starting pos, row, col, and ending pos, row, col.
```

```
//-------------------------------------------------------------------
void Sample34() {

ECLPS              *pPS;                  // Pointer to PS object
ECLFieldList *pFieldList;       // Pointer to field list object
ECLField         *pField;              // Pointer to field object

try {
   pPS = new ECLPS('A');                        // Create PS object for 'A'

   pFieldList = pPS->GetFieldList();       // Get pointer to field list
   pFieldList->Refresh();                        // Build the field list

   printf("Start(Pos ,Row ,Col )    End(Pos ,Row ,Col ) Length(Len )\n");
   for (pField = pFieldList->GetFirstField();           // First field
      pField != NULL;                                                  // While more
      pField = pFieldList->GetNextField(pField)) {    // Next field

     printf("Start(%04lu,%04lu,%04lu)    End(%04lu,%03lu,%04lu)
      Length(%04lu)\n",
        pField->GetStart(),  pField->GetStartRow(),
        pField->GetStartCol(),
        pField->GetEnd(),  pField->GetEndRow(),
        pField->GetEndCol(),  pField->GetLength());
  }
  delete  pPS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetStartRow

This method returns the starting row position of a given field in the ECLFieldList collection for the conection associated with the ECLPS object.

Prototype

ULONG GetStartRow()

Parameters

None

Return Value

ULONG This is the starting row of a given field.

Example

```
/-------------------------------------------------------------------
// ECLField::GetStartRow
//
// Iterate over list of fields and print each field
// starting pos, row, col, and ending pos, row, col.
//-------------------------------------------------------------------
void Sample34() {
```

```
ECLPS              *pPS;                  // Pointer to PS object
ECLFieldList *pFieldList;       // Pointer to field list object
ECLField         *pField;              // Pointer to field object

try {
   pPS = new ECLPS('A');                        // Create PS object for 'A'

   pFieldList = pPS->GetFieldList();       // Get pointer to field list
   pFieldList->Refresh();                     // Build the field list

   printf("Start(Pos ,Row ,Col )    End(Pos ,Row ,Col ) Length(Len )\n");
   for (pField = pFieldList->GetFirstField();          // First field
      pField != NULL;                                          // While more
      pField = pFieldList->GetNextField(pField)) {    // Next field

      printf("Start(%04lu,%04lu,%04lu)   End(%04lu,%03lu,%04lu)   Length(%04lu)\n",
         pField->GetStart(), pField->GetStartRow(), pField->GetStartCol(),
        pField->GetEnd(),  pField->GetEndRow(),
        pField->GetEndCol(),  pField->GetLength());
  }
  delete  pPS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetStartCol

This method return the starting column position of a given field in the ECLFieldList collection for the connection associated witht the ECLPS object.

Prototype

ULONG GetStartCol()

Parameters

None

Return Value

ULONG This is the starting column of a given field.

Example

```
/-------------------------------------------------------------------
// ECLField::GetStartCol
//
// Iterate over list of fields and print each field
// starting pos, row, col, and ending pos, row, col.
//-------------------------------------------------------------------
void Sample34() {

ECLPS              *pPS;                  // Pointer to PS object
ECLFieldList *pFieldList;       // Pointer to field list object
ECLField         *pField;              // Pointer to field object

try {
```

```
   pPS = new ECLPS('A');                        // Create PS object for 'A'

   pFieldList = pPS->GetFieldList();       // Get pointer to field list
   pFieldList->Refresh();                    // Build the field list

   printf("Start(Pos ,Row ,Col )    End(Pos ,Row ,Col ) Length(Len )\n");
   for (pField = pFieldList->GetFirstField();          // First field
      pField != NULL;                                              // While more
      pField = pFieldList->GetNextField(pField)) {    // Next field

     printf("Start(%04lu,%04lu,%04lu)    End(%04lu,%03lu,%04lu)
        Length(%04lu)\n",
        pField->GetStart(),  pField->GetStartRow(),
        pField->GetStartCol(),
        pField->GetEnd(),  pField->GetEndRow(),
        pField->GetEndCol(),  pField->GetLength());
  }
  delete  pPS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetEnd

This method returns the position in the presentation space of the last character of the field. There are two signatures for the GetEnd method. ULONG GetEnd returns the position as a linear value with the upper left corner of the presentation space being “1”. void GetEnd(ULONG *Row, ULONG *Col) returns the position as a row and column coordinate.

Prototype

ULONG GetEnd()

void GetEnd(ULONG *Row, ULONG *Col)

Parameters

ULONG *Row This output parameter is a pointer to the row value to be updated.

ULONG *Col This output parameter is a pointer to the column value to be updated.

Return Value

ULONG Position in the presentation space represented as a linear array.

Example

The following example shows how to return the position in the presentation space of the last character of the field.

```
/--------------------------------------------------------------------
// ECLField::GetEnd
//
// Iterate over list of fields and print each field
// starting pos, row, col, and ending pos, row, col.
```

```
//-------------------------------------------------------------------
void Sample34() {

ECLPS          *pPS;          // Pointer to PS object
ECLFieldList *pFieldList;     // Pointer to field list object
ECLField       *pField;       // Pointer to field object

try {
   pPS = new ECLPS('A');                    // Create PS object for 'A'

   pFieldList = pPS->GetFieldList();      // Get pointer to field list
   pFieldList->Refresh();                   // Build the field list

   printf("Start(Pos ,Row ,Col )    End(Pos ,Row ,Col ) Length(Len )\n");
   for (pField = pFieldList->GetFirstField();          // First field
      pField != NULL;                                          // While more
      pField = pFieldList->GetNextField(pField)) {    // Next field

     printf("Start(%04lu,%04lu,%04lu)    End(%04lu,%03lu,%04lu)
      Length(%04lu)\n",
        pField->GetStart(),  pField->GetStartRow(),
        pField->GetStartCol(),
        pField->GetEnd(),  pField->GetEndRow(),
        pField->GetEndCol(),  pField->GetLength());
  }
  delete  pPS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetEndRow

This method returns the ending row position of the field.

Prototype

ULONG GetEndRow()

Parameters

None

Return Value

ULONG This is the ending row in a given field.

Example

```
/-------------------------------------------------------------------
// ECLField::GetEndRow
//
// Iterate over list of fields and print each field
// starting pos, row, col, and ending pos, row, col.
//-------------------------------------------------------------------
void Sample34() {
```

```
ECLPS              *pPS;                  // Pointer to PS object
ECLFieldList *pFieldList;       // Pointer to field list object
ECLField         *pField;              // Pointer to field object

try {
   pPS = new ECLPS('A');                        // Create PS object for 'A'

   pFieldList = pPS->GetFieldList();       // Get pointer to field list
   pFieldList->Refresh();                     // Build the field list

   printf("Start(Pos ,Row ,Col )    End(Pos ,Row ,Col ) Length(Len )\n");
   for (pField = pFieldList->GetFirstField();           // First field
      pField != NULL;                                                  // While more
      pField = pFieldList->GetNextField(pField)) {    // Next field

     printf("Start(%04lu,%04lu,%04lu)    End(%04lu,%03lu,%04lu)
      Length(%04lu)\n",
        pField->GetStart(),  pField->GetStartRow(),
        pField->GetStartCol(),
        pField->GetEnd(),  pField->GetEndRow(),
        pField->GetEndCol(),  pField->GetLength());
  }
  delete  pPS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetEndCol

This method returns the ending column position of a field.

Prototype

ULONG GetEndCol()

Parameters

None

Return Value

ULONG This is the ending row in a given field.

Example

```
/--------------------------------------------------------------------
// ECLField::GetEndCol
//
// Iterate over list of fields and print each field
// starting pos, row, col, and ending pos, row, col.
//--------------------------------------------------------------------
void Sample34() {

ECLPS              *pPS;                  // Pointer to PS object
ECLFieldList *pFieldList;       // Pointer to field list object
ECLField         *pField;              // Pointer to field object
```

```
try {
   pPS = new ECLPS('A');                          // Create PS object for 'A'

   pFieldList = pPS->GetFieldList();       // Get pointer to field list
   pFieldList->Refresh();                    // Build the field list

   printf("Start(Pos ,Row ,Col )    End(Pos ,Row ,Col ) Length(Len )\n");
   for (pField = pFieldList->GetFirstField();          // First field
      pField != NULL;                                          // While more
      pField = pFieldList->GetNextField(pField)) {    // Next field

     printf("Start(%04lu,%04lu,%04lu)    End(%04lu,%03lu,%04lu)
      Length(%04lu)\n",
        pField->GetStart(),  pField->GetStartRow(),
        pField->GetStartCol(),
        pField->GetEnd(),  pField->GetEndRow(),
        pField->GetEndCol(),  pField->GetLength());
  }
  delete  pPS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetLength

This method returns the length of the field. The length includes the entire field even if it spans multiple lines of the presentation space. It does not include the field attribute character that starts the field.

Prototype

ULONG GetLength()

Parameters

None

Return Value

ULONG Length of the field.

Example

The following example shows how to return the length of the field.

```
/---------------------------------------------------------------------
// ECLField::GetEndCol
//
// Iterate over list of fields and print each field
// starting pos, row, col, and ending pos, row, col.
//---------------------------------------------------------------------
void Sample34() {

ECLPS              *pPS;                  // Pointer to PS object
ECLFieldList *pFieldList;       // Pointer to field list object
ECLField         *pField;              // Pointer to field object
```

```
try {
   pPS = new ECLPS('A');                          // Create PS object for 'A'

   pFieldList = pPS->GetFieldList();       // Get pointer to field list
   pFieldList->Refresh();                          // Build the field list

   printf("Start(Pos ,Row ,Col )    End(Pos ,Row ,Col ) Length(Len )\n");
   for (pField = pFieldList->GetFirstField();           // First field
      pField != NULL;                                                  // While more
      pField = pFieldList->GetNextField(pField)) {    // Next field

      printf("Start(%04lu,%04lu,%04lu)   End(%04lu,%03lu,%04lu)   Length(%04lu)\n",
         pField->GetStart(), pField->GetStartRow(), pField->GetStartCol(),
        pField->GetEnd(),  pField->GetEndRow(),
        pField->GetEndCol(),  pField->GetLength());
  }
  delete  pPS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetText

The GetText method fills an application-supplied buffer with data from the field. The type of data copied to the buffer is selected with the optional Plane parameter. The default is to return the text plane data. The data returned is the field as it existed at the time this field object was created; it will not reflect the current contents of the field if it has been updated since the ECLFieldList::Refresh function was called.

The length of the data returned is the length of the field (see the GetLength method). When the TextPlane is copied, an additional null terminating byte is added after the last data byte. Therefore, the application should provide a buffer that is at least 1 byte more than the field length when getting the text plane. If the application buffer is too small the returned data is truncated. The number of bytes of copied to the application buffer is returned as the function result (not including the null terminator for copies of the text plane).

The FieldPlane cannot be obtained with this function. The ECLField::GetAttribute can be used to obtain the field attribute value.

Prototype

ULONG GetText(char *Buff, ULONG BuffLen, PS_PLANE Plane=TextPlane)

Parameters

char * Buff Pointer to application buffer to be filled with field data.

ULONG BuffLen Length of application buffer.

PS_PLANE Plane Optional parameter. Enumeration which indicates what plane of field data is to be retrieved. Must be one of TextPlane, ColorPlane, or ExtendedFieldPlane.

Return Value

ULONG Number of bytes copied to application buffer, not including trailing null character for TextPlane data.

Example

The following example shows how to return a pointer to the field data indicated by the Plane parameter.

```
/-------------------------------------------------------------------
// ECLField::GetText
//
// Iterate over list of fields and print each fields text contents.
//-------------------------------------------------------------------
void Sample35() {

ECLPS              *PS;                 // Pointer to PS object
ECLFieldList *FieldList;       // Pointer to field list object
ECLField         *Field;              // Pointer to field object
char              *Buff;              //  Screen  data  buffer
ULONG             BuffLen;

try {
   PS = new ECLPS('A');                        // Create PS object for 'A'

   BuffLen = PS->GetSize() + 1;              // Make big enough for entire screen
   Buff = new char[BuffLen];                  // Allocate screen buffer

   FieldList = PS->GetFieldList();          // Get pointer to field list
   FieldList->Refresh();                       // Build the field list

   for (Field = FieldList->GetFirstField();         // First field
      Field != NULL;                                          // While more
      Field = FieldList->GetNextField(Field)) {    // Next field

         Field->GetText(Buff, BuffLen); // Get this fields text
         printf("%02lu,%02lu: %s\n",           // Print "row,col: text"
                  Field->GetStartRow(),
                  Field->GetStartCol(),
                  Buff);
  }
  delete  []Buff;
  delete  PS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

SetText

This method populates a given field in the presentation space with the character string passed in as text. If the text exceeds the length of the field, the text is truncated. If the text is shorter than the field, the field is paddled with nulls.

Prototype
void SetText(char *text)

Parameters
char *text Null terminated string to set in field.

Return Value
None

Example
The following example shows how to populate a given field in the presentation space with the character string passed in as text.

```
//-------------------------------------------------------------------
// ECLField::SetText
//
// Set the field that contains row 2, column 10 to a value.
//-------------------------------------------------------------------
void Sample36() {

ECLPS          *PS;              // Pointer to PS object
ECLFieldList *FieldList;     // Pointer to field list object
ECLField       *Field;           // Pointer to field object

try {
   PS = new ECLPS('A');                    // Create PS object for 'A'
   FieldList = PS->GetFieldList();      // Get pointer to field list
   FieldList->Refresh();                   // Build the field list

   // If the field at row 2 col 10 is an input field, set
   // it to a new value.
   Field = FieldList->FindField(2, 10);     // Find field at this location
   if (Field != NULL) {
      if (!Field->IsProtected())              // Make sure its an input field
         Field->SetText("Way cool!");          // Assign new field text
     else
         printf("Position 2,10 is protected.\n");
   }
   else printf("Cannot find field at position 2,10.\n");

  delete  PS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

IsModified, IsProtected, IsNumeric, IsHighIntensity, IsPenDetectable, IsDisplay

This method determines if a given field in the presentation space has a particular attribute. The method returns a TRUE value if the field has the attribute or a FALSE value if the field does not have the attribute.

Prototype

BOOL IsModified(),

BOOL IsProtected(),

BOOL IsNumeric(),

BOOL IsHighIntensity(),

BOOL IsPenDetectable(),

BOOL IsDisplay()

Parameters

None

Return Value

BOOL Returns a TRUE value if the attribute is present; a FALSE value if the attribute is not present.

Example

The following example shows how to determine if a given field has an attribute.

```
//-------------------------------------------------------------------
// ECLField::IsModified
// ECLField::IsProtected
// ECLField::IsNumeric
// ECLField::IsHighIntensity
// ECLField::IsPenDetectable
// ECLField::IsDisplay
//
// Iterate over list of fields and print each fields attributes.
//-------------------------------------------------------------------
void Sample37() {

ECLPS          *PS;          // Pointer to PS object
ECLFieldList *FieldList;     // Pointer to field list object
ECLField       *Field;       // Pointer to field object

try {
   PS = new ECLPS('A');                  // Create PS object for 'A'

   FieldList = PS->GetFieldList();       // Get pointer to field list
   FieldList->Refresh();                 // Build the field list

   for (Field = FieldList->GetFirstField();       // First field
      Field != NULL;                                  // While more
      Field = FieldList->GetNextField(Field)) {    // Next field
```

```
          printf("Field at %02lu,%02lu is: ",
                    Field->GetStartRow(),  Field->GetStartCol());

          if  (Field->IsProtected())
             printf("Protect  ");
          else
             printf("Input      ");

          if  (Field->IsModified())
             printf("Modified      ");
          else
             printf("Unmodified  ");

          if  (Field->IsNumeric())
             printf("Numeric    ");
          else
             printf("Alphanum  ");

          if  (Field->IsHighIntensity())
             printf("HiIntensity  ");
          else
             printf("Normal           ");

          if  (Field->IsPenDetectable())
             printf("Penable  ");
          else
             printf("NoPen      ");

          if  (Field->IsDisplay())
             printf("Display  \n");
          else
             printf("Hidden    \n");
    }
    delete  PS;
}
catch (ECLErr Err) {
     printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample

//---------------------------------------------------------------------
```

GetAttribute

This method returns the attribute of the field. The value returned contains the bit flags for each of the possible field attributes (modified, protected, numeric, high intensity, pen, and display). See Appendix B, “ECL Planes — Format and Content” on page 323 for more details on these bits. There is a method provided for each type of attribute (for example, IsModified or IsHighIntensity). This method can be used to obtain complete attribute information in a single call.

Prototype
unsigned char GetAttribute()

Parameters
None

Return Value
unsigned char Attribute bits of the field.

Example
The following example shows how to return the attribute of the field.

```
/ ECLField::GetAttribute
//
// Iterate over list of fields and print each fields attribute
// value.
//--------------------------------------------------------------------
void Sample38() {

ECLPS          *PS;              // Pointer to PS object
ECLFieldList *FieldList;     // Pointer to field list object
ECLField       *Field;           // Pointer to field object

try {
   PS = new ECLPS('A');                    // Create PS object for 'A'

   FieldList = PS->GetFieldList();       // Get pointer to field list
   FieldList->Refresh();                    // Build the field list

   for (Field = FieldList->GetFirstField();       // First field
      Field != NULL;                                        // While more
      Field = FieldList->GetNextField(Field)) {    // Next field

         printf("Attribute value for field at %02lu,%02lu is: 0x%02x\n",
                  Field->GetStartRow(),  Field->GetStartCol(),
                  Field->GetAttribute());
  }
  delete  PS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLFieldList Class

The ECLFieldList class performs operations on a list of fields in a host presentation space. An application should not create an ECLFieldList object directly, but only indirectly by creating an ECLPS object.

ECLFieldList contains a collection of all the fields in the presentation space. Each element of the collection is an ECLField object. See “ECLField Class” on page 63 for details on its properties and methods.

An ECLFieldList object provides a static snapshot of what the presentation space contained when the Refresh method was called. If the presentation space is updated after the call to Refresh(), the field list does not reflect those changes. An application must explicitly call Refresh to refresh the field list.

Once an application has called Refresh it can begin walking through the collection of fields using GetFirstField and GetNextField. If the location of a field is known, FindField can be used to locate it in the list directly.

Note: All ECLField object pointers returned by GetFirstField, GetNextField, and FindField become invalid when Refresh is called or the ECLFieldList object is destroyed.

Derivation

ECLBase>ECLFieldList

Properties

None

ECLFieldList Methods

The following section describes the methods that are valid for the ECLFieldList class.

void Refresh(PS_PLANE Planes)
ULONG GetFieldCount()
ECLField * GetFirstField()
ECLField * GetNextFieldInfo()
ECLField * FindField(ULONG Pos)
ECLField * FindField(ULONG Row, ULONG Col)

Refresh

This method gets a snapshot of all the fields currently in the presentation space. All ECLField object pointers previously returned by this object become invalid. To improve performance, the field data can be limited to the planes of interest. Note that the TextPlane and FieldPlane are always obtained.

Prototype

void Refresh(PS_PLANE Planes=TextPlane)

Parameters

PS_PLANE Planes Plane for which fields are built. Valid values are **TextPlane**, **ColorPlane**, **FieldPlane**, **ExfieldPlane**, and **AllPlanes** (to build for all). This is an enumeration defined in ECLPS.HPP. This optional parameter defaults to TextPlane.

Return Value

None

Example

The following example shows how to use the Refresh method to get a snapshot of all the fields currently in the presentation space.

```
///--------------------------------------------------------------------
// ECLFieldList::Refresh
//
// Display number of fields on the screen.
//--------------------------------------------------------------------
void Sample39() {

ECLPS          *PS;            // Pointer to PS object
ECLFieldList *FieldList;    // Pointer to field list object

try {
   PS = new ECLPS('A');                    // Create PS object for 'A'

   FieldList = PS->GetFieldList();      // Get pointer to field list
   FieldList->Refresh();                  // Build the field list

   printf("There are %lu fields on the screen of connection %c.\n",
     FieldList->GetFieldCount(),  PS->GetName());
```

```
  delete  PS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample

----------------------------------------
```

GetFieldCount

This method returns the number of fields present in the ECLFieldList collection (based on the most recent call to the Refresh method).

Prototype

ULONG GetFieldCount()

Parameters

None

Return Value

ULONG Number of fields in the ECLFieldList collection.

Example

The following example shows how to use the GetFieldCount method to return the number of fields present in the ECLFieldList collection.

```
//-------------------------------------------------------------------
// ECLFieldList::GetFieldCount
//
// Display number of fields on the screen.
//-------------------------------------------------------------------
void Sample40() {

ECLPS          *PS;              // Pointer to PS object
ECLFieldList *FieldList;     // Pointer to field list object

try {
   PS = new ECLPS('A');                    // Create PS object for 'A'

   FieldList = PS->GetFieldList();      // Get pointer to field list
   FieldList->Refresh();                  // Build the field list

   printf("There are %lu fields on the screen of connection %c.\n",
     FieldList->GetFieldCount(),  PS->GetName());

  delete  PS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetFirstField

This method returns a pointer to the first ECLField object in the collection. ECLFieldList contains a collection of ECLField objects. See “ECLField Class” on page 63 for more information. The method returns a NULL pointer if there are no fields in the collection.

Prototype

ECLField * GetFirstField();

Parameters

None

Return Value

ECLField * Pointer to an ECLField object. If there are no fields in the connection, a null is returned.

Example

The following example shows how to use the GetFirstField method to return a pointer to the first ECLField object in the collection.

```
/-------------------------------------------------------------------
// ECLFieldList::GetFirstField
//
// Display starting position of every input (unprotected) field.
//-------------------------------------------------------------------
void Sample41() {

ECLPS          *PS;          // Pointer to PS object
ECLFieldList *FieldList;     // Pointer to field list object
ECLField       *Field;       // Pointer to field object

try {
   PS = new ECLPS('A');                  // Create PS object for 'A'

   FieldList = PS->GetFieldList();       // Get pointer to field list
   FieldList->Refresh();                 // Build the field list

   // Interate over (only) unprotected fields
   printf("List of input fields:\n");
   for (Field = FieldList->GetFirstField(GetUnprotected);
      Field != NULL;
      Field = FieldList->GetNextField(Field, GetUnprotected)) {

      printf("Input field starts at %02lu,%02lu\n",
                 Field->GetStartRow(),  Field->GetStartCol());
  }
  delete  PS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetNextField

This method returns the next ECLField object in the collection after a given object. If there are no more objects in the collection after the given object, a NULL pointer is returned. An application can make repeated calls to this method to iterate over the ECLField objects in the collection.

Prototype

LField *GetNextField(ECLField *Prev)

Parameters

ECLField *Prev A pointer to any ECLField object in the collection. The returned pointer will be the next object after this one. If this value is NULL a pointer to the first object in the collection is returned. This pointer is a pointer returned by the GetFirstField, GetNextField, or FindField member functions.

Return Value

ECLField * A pointer to the next object in the collection. If there are no more objects in the collection after the Prev object, NULL is returned.

Example

The following example shows how to use the GetNextFieldInfo method to return a pointer to the next ECLField object in the collection.

```
///--------------------------------------------------------------------
// ECLFieldList::GetNextField
//
// Display starting position of every input (unprotected) field.
//--------------------------------------------------------------------
void Sample42() {

ECLPS          *PS;              // Pointer to PS object
ECLFieldList *FieldList;     // Pointer to field list object
ECLField       *Field;           // Pointer to field object

try {
   PS = new ECLPS('A');                    // Create PS object for 'A'

   FieldList = PS->GetFieldList();       // Get pointer to field list
   FieldList->Refresh();                    // Build the field list

   // Interate over (only) unprotected fields
   printf("List of input fields:\n");
   for (Field = FieldList->GetFirstField(GetUnprotected);
      Field != NULL;
      Field = FieldList->GetNextField(Field, GetUnprotected)) {

      printf("Input field starts at %02lu,%02lu\n",
                Field->GetStartRow(),  Field->GetStartCol());
  }
  delete  PS;
}
catch (ECLErr Err) {
```

```
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

FindField

This method finds a field in the ECLFieldList collection using either text or a position. The position can be either a linear position or a row,column position. If a field contains the text or the position, a pointer to an ECLField object for that field is returned. The returned pointer is to an object in the field list collection. NULL is returned if the field is not found. When searching for text, the search begins at row1 column1 unless you specify a starting position. Also for text, this method will search forward in the list as a default; however, you can specify the direction to search explicity.

Note: A search for text will be successful even if the text spans multiple fields. The field object returned will be the field where the found text begins.

Prototype

ECLField *FindField(ULONG Pos);

ECLField *FindField(ULONG Row, ULONG Col);

ECLField *FindField(char* text, PS_DIR DIR=SrchForward);

ECLField *FindField(char* text, ULONG Pos, PS_DIR DIR=SrchForward);

ECLField *FindField(char* text, ULONG Row, ULONG Col, PS_DIR DIR=SrchForward);

Parameters

ULONG Pos Linear position to search for OR linear position to begin text search.

ULONG Row Row position to search for OR row to begin text search.

ULONG Col Column position to search for OR column to begin text search.

char *text String to search

PS_DIR Dir Direction to search

Return Value

ECLField * Pointer to an ECLField object if field is found. NULL if field is not found. Returned pointer is invalid after the next call to Refresh.

Example

The following is an example of the FindField method.

```
//-------------------------------------------------------------------
// ECLFieldList::FindField
//
// Display the field which contains row 2 column 10.    Also find
// the first field containing a particular string.
//-------------------------------------------------------------------
void Sample43() {
```

```
ECLPS          *PS;                 // Pointer to PS object
ECLFieldList *FieldList;       // Pointer to field list object
ECLField       *Field;              // Pointer to field object
char           Buff[4000];

try {
   PS = new ECLPS('A');                        // Create PS object for 'A'

   FieldList = PS->GetFieldList();         // Get pointer to field list
   FieldList->Refresh();                      // Build the field list

   // Find by row,column coordinate

   Field = FieldList->FindField(2, 10);
   if (Field != NULL) {
     Field->GetText(Buff,  sizeof(Buff));
      printf("Field at 2,10: %s\n", Buff);
   }
   else printf("No field found at 2,10.\n");

   // Find by text.    Note that text may span fields, this
   // will find the field in which the text starts.

   Field = FieldList->FindField("IBM");
   if (Field != NULL) {
      printf("String 'IBM' found in field that starts at %lu,%lu.\n",
                Field->GetStartRow(),  Field->GetStartCol());
   }
   else printf("String 'IBM' not found.\n");

  delete  PS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample

//----------------------------
```

ECLKeyNotify Class

ECLKeyNotify is an abstract base class. An application cannot create an instance of this class directly. To use this class, the application must define its own class which is derived from ECLKeyNotify. The application must implement the NotifyEvent() member function in its derived class. It may also optionally implement NotifyError() and NotifyStop() member functions.

The ECLKeyNotify class is used to allow an application to be notified of keystroke events. The application can also choose to "filter" (remove) the keystrokes so they are not sent to the host screen, or replace them with other keystrokes. Keystroke notifications are queued so that the application will always recieve a notification for each and every keystroke. Only keystrokes made by the real physical keyboard are detected by this object; keystrokes sent to the host by other ECL objects (such as ECLPS::SendKeys) do not cause keystroke notification events.

To be notified of keystroke events, the application must perform the following steps:

1. Define a class derived from ECLKeyNotify.
2. Implement the derived class and implement the NotifyEvent() member function.
3. Optionally implement the NotifyError() and/or NotifyStop() functions.
4. Create an instance of the derived class.
5. Register the instance with the ECLPS::RegisterKeyEvent() function.

The example shown demonstrates how this may be done. When the above steps are complete, each keystroke in the emulator window will cause the applications NotifyEvent() member function to be called. The function is passed parameters indicating the type of keystroke (plain ASCII key, or special function key), and the value of the key (a single ASCII character, or a keyword representing a function key). The application may perform any functions required in the NotifyEvent() procedure, including calling other ECL functions such as ECLPS::SendKeys(). The application returns a value from NotifyEvent() to indicate if the keystroke is to be filtered or not (return 1 to filter (discard) the keystroke, return 0 to have it processed normally).

If an error is detected during keystroke event generation, the NotifyError() member function is called with an ECLErr object. Keystroke events may or may not continue to be generated after an error, depending on the nature of the error. When event generation terminates (either due to an error, by calling ECLPS::UnregisterKeyEvent, or by destruction of the ECLPS object) the NotifyStop() member function is called. However event notification is terminated, the NotifyStop() member function is always called , and the application object is unregistered.

If the application does not provide an implementation of the NotifyError() member function, the default implementation is used (a simple message box is displayed to the user). The application can override the default behavior by implementing the NotifyError() function in the applications derived class. Likewise, the default NotifyStop() function is used if the application does not provide this function (the default behavior is to do nothing).

Note that the application can also choose to provide its own constructor and destructor for the derived class. This can be useful if the application wants to store

some instance-specific data in the class and pass that information as a parameter on the constructor. For example, the application may want to post a message to an application window when a keystroke occurs. Rather than define the window handle as a global variable (so it would be visible to the NotifyEvent() function), the application can define a constructor for the class which takes the window handle and stores it in the class member data area.

The application must not destroy the notification object while it is registered to receive events.

The same instance of a keystroke notification object can be registered with multiple ECLPS objects to receive keystrokes for multiple connections. Thus an application can use a single instance of this object to process keystrokes on any number of sessions. The member functions are passed a pointer to the ECLPS object for which the event occured so an application can distingush between events on different connections. The sample shown uses the same object to process keystrokes on two connections.

Implementation Restriction: Currently the ECLPS object allows only one notification object to be registered for a given connection. The ECLPS::RegisterKeyEvent will throw an error if a notify object is already registered for that ECLPS object.

Derivation

ECLBase > ECLNotify > ECLKeyNotify

Usage Notes

The following is an example of how to construct and use an ECLKeyNotify object.

```
// ECLKeyNotify class
//
// This sample demonstrates the use of:
//
// ECLKeyNotify::NotifyEvent
// ECLKeyNotify::NotifyError
// ECLKeyNotify::NotifyStop
// ECLPS::RegisterKeyEvent
// ECLPS::UnregisterKeyEvent
//-----------------------------------------------------------------

//.....................................................................
// Define a class derived from ECLKeyNotify
//.....................................................................
class MyKeyNotify: public ECLKeyNotify
{
public:
   // Define my own constructor to store instance data
  MyKeyNotify(HANDLE  DataHandle);

   // We have to implement this function
   virtual int NotifyEvent(ECLPS *PSObj, char const KeyType[2],
                                   const char * const KeyString);

   // We choose to implement this function
   void NotifyStop (ECLPS *PSObj, int Reason);
```

```
   // We will take the default behaviour for this so we
   // don't implement it in our class:
   // void NotifyError (ECLPS *PSObj, ECLErr ErrObject);

private:
   // We will store our application data handle here
  HANDLE  MyDataH;
};

 //.....................................................................
MyKeyNotify::MyKeyNotify(HANDLE DataHandle)      // Constructor
//.....................................................................
{
   MyDataH = DataHandle;    // Save data handle for later use
}

//.....................................................................
int MyKeyNotify::NotifyEvent(ECLPS *PSObj,
                                        char const KeyType[2],
                                        const char * const KeyString)
//.....................................................................

{
   // This function is called whenever a keystroke occures.    We will
   // just do something simple: when the user presses PF1 we will
   // send a PF2 to the host instead.    All other keys will be unchanged.

   if (KeyType[0] == 'M') {                      // Is this a mnemonic keyword?
      if (!strcmp(KeyString, "[pf1]")) { // Is it a PF1 key?
         PSObj->SendKeys("[pf2]");               // Send PF2 instead
         printf("Changed PF1 to PF2 on connection %c.\n",
                  PSObj->GetName());
         return 1;                                      // Discard this PF1 key
      }
   }

   return 0;                                          // Process key normally
}

 //.....................................................................
void MyKeyNotify::NotifyStop (ECLPS *PSObj, int Reason)
//.....................................................................
{
   // When notification ends, display message
   printf("Keystroke intercept for connection %c stopped.\n", PSObj->GetName());
}

//.....................................................................
// Create the class and start keystroke processing on A and B.
//.....................................................................
void Sample44() {

ECLPS *PSA, *PSB;            // PS objects
MyKeyNotify *Event;         // Ptr to my event handling object
HANDLE InstData;             // Handle to application data block (for example)

try {
```

```
   PSA = new ECLPS('A');                    // Create PS objects
   PSB = new ECLPS('B');
   Event = new MyKeyNotify(InstData);    // Create event handler

   PSA->RegisterKeyEvent(Event);          // Register for keystroke events
   PSB->RegisterKeyEvent(Event);          // Register for keystroke events

   // At this point, any keystrokes on A or B will cause the
   // MyKeyEvent::NotifyEvent() function to execute.    For
   // this sample, we put this thread to sleep during this
  //  time.

   printf("Processing keystrokes for 60 seconds on A and B...\n");
  Sleep(60000);

   // Now stop event generation.    This will cause the NotifyStop
   // member to be called.
  PSA->UnregisterKeyEvent(Event);
  PSB->UnregisterKeyEvent(Event);

   delete Event;    // Don't delete until after unregister!
  delete  PSA;
  delete  PSB;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample

//---------------------------------------------------------------------
```

ECLKeyNotify Methods

The following section describes the methods that are valid for the ECLKeyNotify class.

```
            ECLKeyNotfiy()
  ~ECLKeyNotify()
virtual int NotifyEvent()
                ECLPS *PSObj,
                char const KeyType[2],
                const char * const KeyString) = 0
virtual void NotifyError (ECLPS *PSObj, ECLErr ErrObject)
virtual void NotifyStop  (ECLPS *PSObject, int Reason)
```

NotifyEvent

This method is a“pure virtual” member function (the application *must* implement this function in classes derived from ECLKeyNotify). This function is called whenever a keystroke event occurs and the object is registered for keystroke events. The return value indicates the disposition of the keystroke (return 1 to discard, 0 to process).

Prototype

virtual int NotifyEvent (ECLPS *PSObj, char const KeyType [2], const char * const KeyString) =0

Parameters

ECLPS *PSObj
This is a ptr to ECLPS object in which the event occurred.

char const KeyType[2]
This is a null terminated 1–char string indicating the type of key:

"A" = Plain ASCII keystroke
"M" = Mnemonic keyword

const char * const KeyString
This is a null terminated string containing the keystroke or mnemonic keyword. Keywords will always be in lower case (for example, "[enter]"). See Appendix A, “Sendkeys Mnemonic Keywords” on page 319 for a list of mnemonic keywords.

Return Value

int
This is the filter indicator.

1 = Filter (discard) keystroke
0 = Process keystroke (send to host)

NotifyError

This method is called whenever the ECLPS object detects an error during keystroke event generation. The error object contains information about the error (see the ECLErr class description). Keystroke events may continue to be generated after the error, depending on the nature of the error. If keystroke event generation stops due to an error, the NotifyStop() function will be called.

Prototype

virtual void NotifyError (ELLPS *PSobj, ECLErr ErrObject)

Parameters

ECLPS *PSObj This is the ptr to ECLPS object in which the error occurred.

ECLErr ErrObject This is the ECLErr object describing the error.

Return Value

None

NotifyStop

This method is called when keystroke event generation is stopped for any reason (for example, due to an error condition, a call to ECLPS::UnregisterKeyEvent, destruction of the ECLPS object, etc).

Prototype

virtual void NotifyStop (ELLPS *PSObj, int Reason)

Parameters

ECLPS *PSObj This is the ptr to ECLPS object in which events are stopping.

int Reason This is unused (zero).

Return Value

None

ECLOIA Class

ECLOIA provides Operator Information Area (OIA) services.

Because ECLOIA is derived from ECLConnection, you can obtain all the information contained in an ECLConnection object. See “ECLConnection Class” on page 23 for more information.

The ECLOIA object is created for the connection identified upon construction. You may create an ECLOIA object by passing either the connection name (a single, alphabetic character from A-Z) or the connection handle, which is usually obtained from the ECLConnList object. There can be only one Personal Communications connection with a given name or handle open at a time.

Derivation

ECLBase > ECLConnection > ECLOIA

Usage Notes

The ECLSession class creates an instance of this object. If the application does not need other services, this object may be created directly. Otherwise, consider using an ECLSession object to create all the objects needed.

ECLOIA Methods

The following section describes the methods that are valid for the ECLOIA class.

ECLOIA(char ConnName)
ECLOIA(long ConnHandle)
~ECLOIA()
BOOL IsAlphanuumeric()
BOOL IsAPL()
BOOL IsKatakana()
BOOL IsHiragana()
BOOL IsDBCS()
BOOL IsUpperShift()
BOOL IsNumLock()
BOOL IsCapsLock()
BOOL IsInsertMode()
BOOL IsCommErrorReminder()
BOOL IsMessageWaiting()
INHIBIT_REASON IsInputInhibited()

ECLOIA Constructor

This method creates an ECLOIA object from a connection name (a single, alphabetic character from A-Z) or a connection handle. There can be only one Personal Communications connection started with a given name.

Prototype

ECLOIA(char ConnName)

ECLOIA(long ConnHandle)

Parameters

char ConnName One-character short name of the connection (A-Z)

long ConnHandle Handle of an ECL connection.

Return Value

None

Example

The following example shows how to create an ECLOIA object using the connection name.

```
// ECLOIA::ECLOIA                    (Constructor)
//
// Build an OIA object from a name, and another from a handle.
//-------------------------------------------------------------------
void Sample45() {

ECLOIA *OIA1, *OIA2;       // Pointer to OIA objects
ECLConnList ConnList;     // Connection list object

try {
   // Create OIA object for connection 'A'
   OIA1 = new ECLOIA('A');
```

```
   // Create OIA object for first connection in conn list
   OIA2 = new ECLOIA(ConnList.GetFirstConnection()->GetHandle());

   printf("OIA #1 is for connection %c, OIA #2 is for connection %c.\n",
            OIA1->GetName(),  OIA2->GetName());
  delete  OIA1;
  delete  OIA2;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample

-----------------------------------------
```

IsAlphanumeric

This method checks to determine if the OIA indicates that the cursor is at an alphanumeric location.

Prototype

BOOL IsAlphanumeric()

Parameters

None

Return Value

BOOL TRUE if the keyboard is in alphanumeric mode; FALSE if the keyboard is not in alphanumeric mode.

Example

The following example shows how to determine if the OIA indicates that the keyboard is in alphanumeric mode.

```
//---------------------------------------------------------------------
// ECLOIA::IsAlphanumeric
//
// Determine status of connection 'A' OIA indicator
//---------------------------------------------------------------------
void Sample46() {

ECLOIA OIA('A');     // OIA object for connection A

if (OIA.IsAlphanumeric())
  printf("Alphanumeric.\n");
else
  printf("Not  Alphanumeric.\n");

} // end sample
```

IsAPL

This method checks to determine if the OIA indicates that the keyboard is in APL mode.

Prototype
BOOL IsAPL()

Parameters
None

Return Value
BOOL TRUE if the keyboard is in APL mode; FALSE if the keyboard is not in APL mode.

Example
The following example shows how to determine if the OIA indicates that the keyboard is in APL mode.

```
//-------------------------------------------------------------------
// ECLOIA::IsAPL
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample47() {

ECLOIA OIA('A');      // OIA object for connection A

if (OIA.IsAPL())
  printf("APL.\n");
else
  printf("Not  APL.\n");

} // end sample

//-------------------------
```

IsKatakana

This method checks to determine if the OIA indicates that Katakana characters are enabled.

Prototype
BOOL IsKatakana()

Parameters
None

Return Value
BOOL TRUE if Katakana characters are enabled; FALSE if Katakana characters are not enabled.

Example

The following example shows how to determine if the OIA indicates that Katakana characters are enabled.

```
// ECLOIA::IsKatakana
//
// Determine status of connection 'A' OIA indicator
//-----------------------------------------------------------------
void Sample48() {

ECLOIA OIA('A');      // OIA object for connection A

if (OIA.IsKatakana())
  printf("Katakana.\n");
else
  printf("Not  Katakana.\n");

} // end sample
```

IsHiragana

This method checks to determine if the OIA indicates that Hiragana characters are enabled.

Prototype

BOOL IsHiragana()

Parameters

None

Return Value

BOOL TRUE if Hiragana characters are enabled; FALSE if Hiragana characters are not enabled.

Example

```
//-----------------------------------------------------------------
// ECLOIA::IsHiragana
//
// Determine status of connection 'A' OIA indicator
//-----------------------------------------------------------------
void Sample49() {

ECLOIA OIA('A');      // OIA object for connection A

if (OIA.IsHiragana())
  printf("Hiragana.\n");
else
  printf("Not  Hiragana.\n");

} // end sample
```

IsDBCS

This method checks to determine if the OIA indicates that the cursor is at a Double Byte Character Set (DBCS) location.

Prototype
BOOL IsDBCS()

Parameters
None

Return Value
BOOL TRUE if the DBCS characters are enabled; FALSE if the DBCS characters are not enabled.

Example
The following example shows how to determine if the OIA indicates that Double Byte Character Set (DBCS) characters are enabled.

```
//-------------------------------------------------------------------
// ECLOIA::IsDBCS
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample50() {

ECLOIA OIA('A');      // OIA object for connection A

if (OIA.IsDBCS())
  printf("DBCS.\n");
else
  printf("Not  DBCS.\n");

} // end sample
```

IsUpperShift

This method checks to determine if the OIA indicates that the keyboard is in upper shift mode.

Prototype
BOOL IsUpperShift()

Parameters
None

Return Value
BOOL TRUE if the keyboard is in upper shift mode; FALSE if the keyboard is not in upper shift mode.

Example
The following example shows how to determine if the OIA indicates that the keyboard is in upper shift mode.

```
//-------------------------------------------------------------------
// ECLOIA::IsUpperShift
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample51() {

ECLOIA OIA('A');     // OIA object for connection A

if (OIA.IsUpperShift())
  printf("UpperShift.\n");
else
  printf("Not  UpperShift.\n");

} // end sample
```

IsNumeric

This method checks to determine if the OIA indicates that the cursor is at a numeric-only location.

Prototype
BOOL IsNumLock()

Parameters
None

Return Value
BOOL TRUE if Numeric is on; FALSE if not Numeric.

Example
The following example shows how to determine if the OIA indicates that the cursor is at a numeric location.

```
//-------------------------------------------------------------------
// ECLOIA::IsNumeric
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample52() {

ECLOIA OIA('A');     // OIA object for connection A

if (OIA.IsNumeric())
  printf("Numeric.\n");
else
  printf("Not  Numeric.\n");

} // end sample
```

IsCapsLock

This method checks to determine if the OIA indicates that the keyboard has Caps Lock on.

Prototype
BOOL IsCapsLock()

Parameters
None

Return Value
BOOL TRUE if Caps Lock is on; FALSE if Caps Lock is not on.

Example
The following example shows how to determine if the OIA indicates that the keyboard has Caps Lock on.

```
//-------------------------------------------------------------------
// ECLOIA::IsCapsLock
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample53() {

ECLOIA OIA('A');      // OIA object for connection A

if (OIA.IsCapsLock())
  printf("CapsLock.\n");
else
  printf("Not  CapsLock.\n");

} // end sample
```

IsInsertMode

This method checks to determine if the OIA indicates that the keyboard is in insert mode.

Prototype
BOOL IsInsertMode()

Parameters
None

Return Value
BOOL TRUE if the keyboard is in insert mode; FALSE if the keyboard is not in insert mode.

Example

The following example shows how to determine if the OIA indicates that the keyboard is in insert mode.

```
//-------------------------------------------------------------------
// ECLOIA::IsInsertMode
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample54() {

ECLOIA OIA('A');      // OIA object for connection A

if (OIA.IsInsertMode())
  printf("InsertMode.\n");
else
  printf("Not  InsertMode.\n");

} // end sample
```

IsCommErrorReminder

This method checks to determine if the OIA indicates that a communications error reminder condition exists.

Prototype

BOOL IsCommErrorReminder()

Parameters

None

Return Value

BOOL TRUE if a condition exists; FALSE if a condition does not exist.

Example

The following example shows how to determine if the OIA indicates that a communications error reminder condition exists.

```
//-------------------------------------------------------------------
// ECLOIA::IsCommErrorReminder
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample55() {

ECLOIA OIA('A');      // OIA object for connection A

if (OIA.IsCommErrorReminder())
  printf("CommErrorReminder.\n");
else
  printf("Not  CommErrorReminder.\n");

} // end sample

//
```

IsMessageWaiting

This method checks to determine if the OIA indicates that the message waiting indicator is on. This can only occur for 5250 connections.

Prototype

BOOL IsMessageWaiting()

Parameters

None

Return Value

BOOL TRUE if the message waiting indicator is on; FALSE if the indicator is not on.

Example

The following example shows how to determine if the OIA indicates that the message waiting indicator is on.

```
-------------------------------------------------------------------
// ECLOIA::IsMessageWaiting
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample56() {

ECLOIA OIA('A');      // OIA object for connection A

if (OIA.IsMessageWaiting())
  printf("MessageWaiting.\n");
else
  printf("Not  MessageWaiting.\n");

} // end sample
```

InputInhibited

This method returns an enumerated value that indicates whether input is inhibited or not. If input is inhibited, the reason for the inhibit can be determined. If input is inhibited for more than one reason the highest value enumeration is returned (for example, if there is a communications error and a protocol programming error, the ProgCheck value is returned).

Prototype

INHIBIT_REASON InputInhibited ()

Parameters

None

Return Value

INHIBIT_REASON Returns one of the INHIBIT_REASON values as defined in ECLOIA.HPP. The value NotInhibited is returned if input is currently not inhibited.

Example

The following example shows how to determine whether input is inhibited or not.

```
//-------------------------------------------------------------------
// ECLOIA::InputInhibited
//
// Determine status of connection 'A' OIA indicator
//-------------------------------------------------------------------
void Sample57() {

ECLOIA OIA('A');      // OIA object for connection A

switch (OIA.InputInhibited()) {
case NotInhibited:
    printf("Input not inhibited.\n");
   break;
case SystemWait:
    printf("Input inhibited for SystemWait.\n");
   break;
case CommCheck:
    printf("Input inhibited for CommCheck.\n");
   break;
case ProgCheck:
    printf("Input inhibited for ProgCheck.\n");
   break;
case MachCheck:
    printf("Input inhibited for MachCheck.\n");
   break;
case OtherInhibit:
    printf("Input inhibited for OtherInhibit.\n");
   break;
default:
    printf("Input inhibited for unknown reason.\n");
   break;
}
} // end sample
```

ECLPS Class

The ECLPS class performs operations on a host presentation space.

The ECLPS object is created for the connection identified upon construction. You may create an ECLPS object by passing either the connection name (a single, alphabetic character from A-Z) or the connection handle, which is usually obtained from an ECLConnection object. There can be only one Personal Communications connection with a given name or handle open at a time.

Derivation

ECLBase > ECLConnection > ECLPS

Properties

None

Usage Notes

The ECLSession class creates an instance of this object. If the application does not need other services, this object may be created directly. Otherwise, you may want to consider using an ECLSession object to create all the objects needed.

ECLPS Methods

The following section describes the methods available for ECLPS.

ECLPS(char ConnName)
ECLPS(long ConnHandle)
ULONG GetSize()
void GetSize(ULONG *rows, ULONG *cols)
ULONG GetSizeRows()
ULONG GetSizeCols()
ULONG GetCursorPos()
ULONG GetCursorPosRow()
ULONG GetCursorPosCol()
void GetCursorPos(ULONG *row, ULONG *col)
void SetCursorPos(ULONG pos)
void SetCursorPos(ULONG row, ULONG col)
void SendKeys(char *text)
void SendKeys(char *text, ULONG AtPos)
void SendKeys(char *text, ULONG AtRow, ULONG AtCol)
ULONG SearchText(char *text, PS_DIR Dir)
ULONG SearchText(char *text, PS_DIR Dir, ULONG *StartPos)
ULONG SearchText(char *text, PS_DIR Dir, ULONG *StartRow, ULONG *StartCol)
ULONG GetText(char *Buff, ULONG BuffLen, PS_PLANE Plane)
ULONG GetText(char *Buff, ULONG BuffLen, ULONG pos, PS_PLANE Plane)
ULONG GetText(char *Buff, ULONG BuffLEN, ULONG row, ULONG col, PS_PLANE Plane)
void SetText
(char *text)
void SetText(char *text)
void SetText(char *text, ULONG Pos)
void SetText(char *text, ULONG Row, ULONG Col)
ULONG GetTextRect(char *Buff, ULONG BuffLen, ULONG startpos, ULONG endpos,
PS_PLANE Plane)
ULONG GetTextRect(char *Buff, ULONG BuffLen, ULONG startrow, ULONG startcol,
ULONG endrow, ULONG endcol, PS_PLANE Plane)
void ConvertPosToRowCol(ULONG pos, ULONG *row, ULONG *col)
ULONG ConvertRowColToPos(ULONG row, ULONG col)
ULONG ConvertPosToRow(ULONG Pos)
ULONG ConvertPosToCol(ULONG Pos)
void RegisterKeyEvent(ECLKeyNotify *NotifyObject)
virtual UnregisterKeyEvent(ECLKeyNotify *NotifyObject)
ECLFieldList * GetFieldList()

ECLPS Constructor

This method uses a connection name or handle to create an ECLPS object.

Prototype

ECLPS(char ConnName)

ECLPS(long ConnHandle)

Parameters

char ConnName One-character short name of the connection (A-Z).

long ConnHandle Handle of an ECL connection.

Return Value

None

Example

The following example shows how to use a connection name to create an ECLPS object.

```
//-------------------------------------------------------------------
// ECLPS::ECLPS                    (Constructor)
//
// Build a PS object from a name, and another from a handle.
//-------------------------------------------------------------------
void Sample58() {

ECLPS *PS1, *PS2;             // Pointer to PS objects
ECLConnList ConnList;     // Connection list object

try {
   // Create PS object for connection 'A'
   PS1 = new ECLPS('A');

   // Create PS object for first connection in conn list
   PS2 = new ECLPS(ConnList.GetFirstConnection()->GetHandle());

   printf("PS #1 is for connection %c, PS #2 is for connection %c.\n",
            PS1->GetName(),  PS2->GetName());
  delete  PS1;
  delete  PS2;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}
} // end sample
```

ECLPS Destructor

This method destroys the ECLPS object.

Prototype

~ECLPS()

Parameters

None

Return Value
None

Example
The following example shows how to destroy an ECLPS object.

```
ULONG              RowPos,  ColPos;
ECLPS             *pPS;

try {
         pPS = new ECLPS("A");
         RowPos = pPS->ConvertPosToRow(544);
         ColPos = pPS->ConvertPostToCol(544);
         printf("PS position is at row %lu column %lu.",
        RowPos,  ColPos);
         // Done with PS object so kill it delete pPS;
}
catch (ECLErr HE) {
   // Just report the error text in a message box
   MessageBox( NULL, HE.GetMsgText(), "Error!", MB_OK );
}
```

GetSize

This method returns the size of the presentation space for the connection associated with the ECLPS object. There are two signatures of the GetSize method. Using ULONG GetSize(), the size is returned as a linear value and represents the total number of characters in the presentation space. With void GetSize(ULONG *Rows, ULONG *Cols), the number of rows and columns of the presentation space is returned.

Prototype
ULONG GetSize()

void GetSize(ULONG *Rows, ULONG *Cols)

Parameters
ULONG *Rows This output parameter is the number of rows in the presentation space.

ULONG *Cols This output parameter is the number of columns in the presentation space.

Return Value
ULONG Size of the presentation space as a linear value.

Example
The following is an example of using the GetSize method.

```
//-------------------------------------------------------------------
// ECLPS::GetSize
//
// Display dimensions of connection 'A'
//-------------------------------------------------------------------
void Sample59() {
```

```
ECLPS PS('A');              // PS object for connection A
ULONG Rows, Cols, Len;

PS.GetSize(&Rows, &Cols);      // Get num of rows and cols
// Could also write as:
Rows = PS.GetSizeRows();       // Redundant
Cols = PS.GetSizeCols();       // Redundant

Len = PS.GetSize();              // Get total size

printf("Connection A has %lu rows and %lu columns (%lu total length)\n",
          Rows, Cols, Len);

} // end sample
```

GetSizeRows

This method returns the number of rows in the Presentation Space for the connection associated with the ECLPS object.

Prototype

ULONG GetSizeRows()

Parameters

None

Return Value

ULONG This is the number of rows in the Presentation Space.

Example

The following is an example of using the GetSizeRows method.

```
//-------------------------------------------------------------------
// ECLPS::GetSizeRows
//
// Display dimensions of connection 'A'
//-------------------------------------------------------------------
void Sample59() {

ECLPS PS('A');              // PS object for connection A
ULONG Rows, Cols, Len;

PS.GetSize(&Rows, &Cols);      // Get num of rows and cols
// Could also write as:
Rows = PS.GetSizeRows();       // Redundant
Cols = PS.GetSizeCols();       // Redundant

Len = PS.GetSize();              // Get total size

printf("Connection A has %lu rows and %lu columns (%lu total length)\n",
          Rows, Cols, Len);

} // end sample
```

GetSizeCols

This method returns the number of columns in the Presentation Space for the connection associated with the ECLPS object.

Prototype
ULONG GetSizeCols()

Parameters
None

Return Value
ULONG This is the number of columns in the Presentation Space.

Example
The following is an example of using the GetSizeCols method.

```
//-------------------------------------------------------------------
// ECLPS::GetSizeCols
//
// Display dimensions of connection 'A'
//-------------------------------------------------------------------
void Sample59() {

ECLPS PS('A');              // PS object for connection A
ULONG Rows, Cols, Len;

PS.GetSize(&Rows, &Cols);      // Get num of rows and cols
// Could also write as:
Rows = PS.GetSizeRows();       // Redundant
Cols = PS.GetSizeCols();       // Redundant

Len = PS.GetSize();                // Get total size

printf("Connection A has %lu rows and %lu columns (%lu total length)\n",
          Rows, Cols, Len);

} // end sample
```

GetCursorPos

This method returns the position of the cursor in the presentation space for the connection associated with the ECLPS object. There are two signatures for the GetCursorPos method. Using ULONG GetCursorPos(), the position is returned as a linear (1-based) position. With void GetCursorPos(ULONG *Row, ULONG * Col), the position is returned as a row and column coordinate.

Prototype
ULONG GetCursorPos()

void GetCursorPos(ULONG *Row, ULONG *Col)

Parameters

ULONG *Row This output parameter is the row coordinate of the host cursor.

ULONG *Col This output parameter is the column coordinate of the host cursor.

Return Value

ULONG Cursor position represented as a linear value.

Example

The following is an example of using the GetCursorPos method.

```
//-------------------------------------------------------------------
// ECLPS::GetCursorPos
//
// Display position of host cursor in connection 'A'
//-------------------------------------------------------------------
void Sample60() {

ECLPS PS('A');              // PS object for connection A
ULONG Row, Col, Pos;

PS.GetCursorPos(&Row, &Col);      // Get row/col position
// Could also write as:
Row = PS.GetCursorPosRow();       // Redundant
Col = PS.GetCursorPosCol();       // Redundant

Pos = PS.GetCursorPos();            // Get linear position

printf("Host cursor of connection A is at row %lu column %lu
 (linear position %lu)\n", Row, Col, Pos);

} // end sample

/
```

GetCursorPosRow

This method returns the row position of the cursor in the Presentation Space for the connection associated with the ECLPS object.

Prototype

ULONG GetCursorPosRow()

Parameters

None

Return Value

ULONG This is the row position of the cursor in the Presentation Space.

Example

The following is an example of using the GetCursorPosRow method.

```
//-------------------------------------------------------------------
// ECLPS::GetCursorPosRow
//
// Display position of host cursor in connection 'A'
//-------------------------------------------------------------------
void Sample60() {

ECLPS PS('A');              // PS object for connection A
ULONG Row, Col, Pos;

PS.GetCursorPos(&Row, &Col);      // Get row/col position
// Could also write as:
Row = PS.GetCursorPosRow();       // Redundant
Col = PS.GetCursorPosCol();       // Redundant

Pos = PS.GetCursorPos();             // Get linear position

printf("Host cursor of connection A is at row %lu column %lu
 (linear position %lu)\n", Row, Col, Pos);

} // end sample
```

GetCursorPosCol

This method returns the column position of the cursor in the Presentation Space for the connection associated with the ECLPS object.

Prototype

ULONG GetCursorPosCol()

Parameters

None

Return Value

ULONG This is the column position of the cursor in the Presentation Space.

Example

The following is an example of using the GetCursorPosCol method.

```
//-------------------------------------------------------------------
// ECLPS::GetCursorPosCol
//
// Display position of host cursor in connection 'A'
//-------------------------------------------------------------------
void Sample60() {

ECLPS PS('A');              // PS object for connection A
ULONG Row, Col, Pos;

PS.GetCursorPos(&Row, &Col);      // Get row/col position
// Could also write as:
Row = PS.GetCursorPosRow();       // Redundant
Col = PS.GetCursorPosCol();       // Redundant
```

```
Pos = PS.GetCursorPos();              // Get linear position

printf("Host cursor of connection A is at row %lu column %lu
 (linear position %lu)\n", Row, Col, Pos);

} // end sample

//---------------------------------------------------------------------
```

SetCursorPos

The SetCursorPos method sets the position of the cursor in the presentation space for the connection associated with the ECLPS object. There are two signatures for the SetCursorPos method. The position can be specified as a linear (1-based) position using void SetCursorPos(ULONG pos), or as a row and column coordinate using void SetCursorPos(ULONG Row, ULONG Col).

Prototype

void SetCursorPos(ULONG pos),

void SetCursorPos(ULONG Row, ULONG Col)

Parameters

ULONG pos Cursor position as a linear position.

ULONG Row Cursor row coordinate.

ULONG Col Cursor column coordinate.

Return Value

None

Example

The following is an example of using the SetCursorPos method.

```
--
// ECLPS::SetCursorPos
//
// Set host cursor to row 2 column 1.
//---------------------------------------------------------------------
void Sample61() {

ECLPS PS('A');                  // PS object for connection A

PS.SetCursorPos(2, 1);    // Put cursor at row 2, column 1
printf("Cursor of connection A set to row 2 column 1.\n");

} // end sample

/
```

SendKeys

The SendKeys method sends a null-terminated string of keys to the presentation space for the connection associated with the ECLPS object. There are three signatures for the SendKeys method. If no position is specified, the keystrokes are entered starting at the current host cursor position. A position may be specified (in linear or row and column coordinates), in which case the host cursor is first moved to the given position.

The text string may contain plain text characters, which are written to the presentation space exactly as given. In addition, the string can contain imbedded keywords (mnemonics) that represent various control keystrokes such as 3270 ENTER keys and 5250 PageUp keys. Keywords are enclosed in square brackets (for example, [enter]). When such a keyword is encountered in the string it is translated into the proper emulator command and sent. A text string may contain any number of plain characters and imbedded keywords. The keywords are processed from left to right until the end of the string is reached. For example, the following string would cause the characters “ABC” to be typed at the current cursor position, followed by a 3270 Erase-end-of-field keystroke, followed by a 3270 Tab keystroke, followed by “XYZ” and a PF1 key:

```
ABC[eraseeof][tab]XYZ[pf1]
```

Note: Blank characters in the string are written to the host presentation space like any other plain text character. Therefore, blanks should not be used to separate keywords or text.

To send a left or right square bracket character to the host, it must be doubled in the text string (for example, it must occur twice to cause a single bracket to be written). The following example causes the string “A [:]” to be written to the presentation space.

```
A[[:]]
```

If you attempt to write keystrokes to a protected position on the screen, the keyboard locks and the remainder of the keystrokes are discarded.

Refer to Appendix A, “Sendkeys Mnemonic Keywords” on page 319 for a list of keywords.

Prototype

void SendKeys(Char * text),

void SendKeys(Char *text, ULONG AtPos),

void SendKeys(Char *text, ULONG AtRow, ULONG AtCol)

Parameters

Char *text String of keys to send to the presentation space.

ULONG AtPos Position at which to start writing keystrokes.

ULONG AtRow Row at which to start writing keystrokes.

ULONG AtCol Row at which to start writing keystrokes.

Return Value
None

Example
The following is an example of using the SendKeys method.

```
//------------------------------------------------------------------
// ECLPS::SendKeys
//
// Sends a series of keystrokes, including 3270 function keys, to
// the host on connection A.
//------------------------------------------------------------------
void Sample62() {

ECLPS PS('A');                  // PS object for connection A

// The following key string will erase from the current cursor
// position to the end of the field, and then type the given
// characters into the field.
char          SendStr[] = "[eraseeof]PCOMM is really cool";

// Note that an ECL error is thrown if we try to send keys to
// a protected field.

try {
   PS.SendKeys(SendStr);                // Do it at the current cursor position
   PS.SendKeys(SendStr, 3, 10); // Again at row 3 column 10
}
catch (ECLErr Err) {
   printf("Failed to send keys: %s\n", Err.GetMsgText());
}

} // end sample
```

SearchText

The SearchText method searches for text in the presentation space of the connection associated with the ECLPS object. The method returns the linear position at which the text is found, or zero if the text is not found. The search may be made in the forward (left to right, top to bottom) or backward (right to left, bottom to top) directions using the optional Dir parameter. The search can be case sensative or case folded (insensative) using the optional FoldCase parameter.

If no starting position is given, the search starts at the beginning of the screen for forward searches, or at the end of the screen for backward searches. A starting position may be given in terms of a linear position or row and column coordinates. If a starting position is given it indicates the position at which to begin the search. Forward searches search from the starting position (inclusive) to the last character of the screen. Backward searches search from the starting position (inclusive) to the first character of the screen.

The search string must exist completely within the search area for the search to be sucessfull (for example, if the search string spans over the specified starting position it will not be found).

The returned linear position may be converted to row and column coordinates using the base class ConvertPosToRowCol method.

Prototype

ULONG SearchText(const char * const text, PS_DIR Dir=SrchForward, BOOL FoldCase=FALSE)

ULONG SearchText(const char * const text, ULONG StartPos, PS_DIR Dir=SrchForward, BOOL FoldCase=FALSE)

ULONG SearchText(const char char * const text, ULONG StartRow, ULONG StartCol, PS_DIR Dir=SrchForward, BOOL FoldCase=FALSE)

Parameters

char *text Null-terminated string to search for.

PS_DIR Dir Optional parameter indicating the direction in which to search. If specified, must be one of **SrchForward** or **SrchBackward**. The default is **SrchForward**.

BOOL FoldCase Optional parameter indicating the case-sensitivity of the search. If specified as FALSE the text string must exactly match the presentation space including the use of upper and lower case characters. If specified as TRUE, the text string will be found without regard to upper or lower case. The default is FALSE.

ULONG StartPos Indicates the starting linear position of the search. This position will be included in the search.

ULONG StartRow Indicates the row in which to start the search.

ULONG StartCol Indicates the column in which to start the search.

Return Value

ULONG Linear position of the found string, or zero if not found.

Example

The following is an example of using the SearchText method.

```
/----------------------------------------------------------------------
// ECLPS::SearchText
//
// Search for a string in various parts of the screen.
//----------------------------------------------------------------------
void Sample63() {

ECLPS PS('A');                    // PS object
char    FindStr[] = "IBM";    // String to search for
ULONG LastOne;                    // Position of search result

// Case insensative search of entire screen

printf("Searching for '%s'...\n", FindStr);
printf("    Anywhere, any case: ");
if (PS.SearchText(FindStr, TRUE) != 0)
  printf("Yes\n");
else
```

```
  printf("No\n");

// Backward, case sensative search on line 1

printf("    Line 1, exact match: ");
if (PS.SearchText(FindStr, 1, 80, SrchBackward) != 0)
  printf("Yes\n");
else
  printf("No\n");

// Backward, full screen search

LastOne = PS.SearchText(FindStr, SrchBackward, TRUE);
if (LastOne != 0)
  printf("  Last occurance on the screen is at row %lu, column %lu.\n",
            PS.ConvertPosToRow(LastOne),  PS.ConvertPosToCol(LastOne));

} // end sample
```

GetText

This method retrieves data from the presentation space of the connection associated with the ECLPS object. The data is returned as a linear array of byte values, one byte per presentation space character position. The array is not null terminated except when data is retrieved from the TextPlane, in which case a single null termination byte is appended.

The application must supply a buffer for the returned data, and the length of the buffer. If the requested data does not fit into the buffer it is truncated. For TextPlane data, the buffer must include at least one extra byte for the terminating null. The method returns the number of bytes copied to the application buffer (not including the terminating null for TextPlane copies).

The application must specify the number of bytes of data to retrieve from the presentation space. If the starting position plus this length exceeds the size of the presentation space an error is thrown. Data is returned starting at the given starting position or row 1, column 1 if no starting position is specified. Returned data is copied from the presentation space in a linear fashion from left to right, top to bottom spanning multiple rows up to the length specified. If the application wants to get screen data for a rectangular area of the screen, the GetTextRect method should be used.

The application can specify any plane for which to retrieve data. If no plane is specified, the TextPlane is retrieved. See Appendix B, “ECL Planes — Format and Content” on page 323 for details on the different ECL planes.

Prototype

ULONG GetText(char * Buff, ULONG BuffLen, PS_PLANE Plane=TextPlane)

ULONG GetText(char * Buff, ULONG BuffLen, ULONG StartPos, ULONG Length, PS_PLANE Plane=TextPlane)

ULONG GetText(char * Buff, ULONG BuffLen, ULONG StartRow, ULONG StartCol, ULONG Length, PS_PLANE Plane=TextPlane)

Parameters

char *Buff Pointer to application supplied buffer of at lest BuffLen size.

ULONG BuffLen Number of bytes in the supplied buffer.

ULONG StartPos Linear position in the presentation space at which to start the copy.

ULONG StartRow Row in the presentation space at which to start the copy.

ULONG StartCol Column in the presentation space at which to start the copy.

ULONG Length Linear number of bytes to copy from the presentation space.

PS_PLANE plane Optional parameter specifing which presentation space plane is to be copied. If specified, must be one of **TextPlane**, **ColorPlane**, **FieldPlane**, and **ExtendedFieldPlane**. The default is **TextPlane**. See Appendix B, “ECL Planes — Format and Content” on page 323 for the content and format of the different ECL planes.

Return Value

ULONG Number of data bytes copied from the presentation space. This value does not include the trailing null byte for TextPlane copies.

Example

The following is an example of using the GetText method.

```
//-------------------------------------------------------------------
// ECLPS::GetText
//
// Get text and other planes of data from the presentation space.
//-------------------------------------------------------------------
void Sample64() {

ECLPS PS('A');                  // PS object
char    *Text;                  // Text plane data
char    *Field;                 // Field plane data
ULONG Len;                          // Size of PS

Len = PS.GetSize();

// Note text buffer needs extra byte for null terminator

Text    = new char[Len + 1];
Field = new char[Len];

PS.GetText(Text, Len+1);                        // Get entire screen (text)
PS.GetText(Field, Len, FieldPlane);         // Get entire field plane
PS.GetText(Text, Len+1, 1, 1, 80);          // Get line 1 of text

printf("Line 1 of the screen is:\n%s\n", Text);

delete []Text;
delete []Field;
```

```
} // end sample
```

GetTextRect

This method retrieves data from the presentation space of the connection associated with the ECLPS object. The data is returned as a linear array of byte values, one byte per presentation space character position. The array is not null terminated.

The application supplies a starting and ending coordinate in the presentation space. These coordinates form the opposing corner points of a rectangular area. The presentation space within the rectanglar area is copied to the application buffer as a single linear array. The starting and ending points may be in any spatial relationship to each other. The copy is defined to start from the row containg the upper-most point to the row containing the lower-most point, and from the left-most column to the right-most column. Both coordinates must be within the bounds of the size of the presentation space or an error is thrown. The coordinates may be specified in terms of linear position or row and column numbers.

The supplied application buffer must be at least large enough to contain the number of bytes in the rectangle. If the buffer is too small, no data is copied and zero is returned as the method result. Otherwise the method returns the number of bytes copied.

The application can specify any plane for which to retrieve data. If no plane is specified, the TextPlane is retrieved. See Appendix B, “ECL Planes — Format and Content” on page 323 for details on the different ECL planes.

Prototype

ULONG GetTextRect(char * Buff, ULONG BuffLen, ULONG StartPos, ULONG EndPos, PS_PLANE Plane=TextPlane)

ULONG GetTextRect(char * Buff, ULONG BuffLen, ULONG StartRow, ULONG StartCol, ULONG EndRow, ULONG EndCol, PS_PLANE Plane=TextPlane)

Parameters

char *Buff Pointer to application supplied buffer of at lest BuffLen size.

ULONG BuffLen Number of bytes in the supplied buffer.

ULONG StartPos Linear position in the presentation space of one corner of the copy rectangle.

ULONG EndPos Linear position in the presentation space of one corner of the copy rectangle.

ULONG StartRow Row in the presentation space of one corner of the copy rectangle.

ULONG StartCol Column in the presentation space of one corner of the copy rectangle.

ULONG EndRow Row in the presentation space of one corner of the copy rectangle.

ULONG EndCol Column in the presentation space of one corner of the copy rectangle.

PS_PLANE plane Optional parameter specifing which presentation space plane is to be copied. If specified, must be one of **TextPlane**, **ColorPlane**, **FieldPlane**, or **ExtendedFieldPlane**. The default is **TextPlane**. See Appendix B, “ECL Planes — Format and Content” on page 323 for the content and format of the different ECL planes.

Return Value

ULONG Number of data bytes copied from the presentation space.

Example

The following is an example of using the GetText method.

```
------------------------------
// ECLPS::GetTextRect
//
// Get rectangular parts of the host screen.
//---------------------------------------------------------------------
void Sample66() {

ECLPS PS('A');              // PS object for connection A
char Buff[4000];           // Big buffer

// Get first 2 lines of the screen text
PS.GetTextRect(Buff, sizeof(Buff), 1, 1, 2, 80);

// Get last 2 lines of the screen
PS.GetTextRect(Buff, sizeof(Buff),
                        PS.GetSizeRows()-1,
                        1,
                        PS.GetSizeRows(),
                        PS.GetSizeCols());

// Get just a part of the screen (OfficeVision/VM main menu calendar)
PS.GetTextRect(Buff, sizeof(Buff),
                        5,   51,
                        13,  76);

// Same as previous (specify any 2 oposite corners of the rectangle)
PS.GetTextRect(Buff, sizeof(Buff),
                        13,  51,
                        5,   76);

// Note results are placed in buffer end-to-end with no line delimiters
printf("Contents of rectangular screen area:\n%s\n", Buff);

} // end sample
```

SetText

The SetText method sends a character array to the Presentation Space for the connection associated with the ECLPS object. Although this is similar to the SendKeys method, it is different in that it does not send mneumonic keystrokes (for example, [enter] or [pf1]).

If a position is not specified, the text is written starting at the current cursor position.

Prototype

```
void SetText(char *text);

void SetText(char *text, ULONG AtPos);

void SetText(char *text, ULONG AtRow, ULONG AtCol);
```

Parameters

char *text	Null terminated string of characters to copy to the presentation space.
ULONG AtPos	Linear position in the presentation space at which to begin the copy.
ULONG AtRow	Row in the presentation space of which to begin the copy.
ULONG AtCol	Column in the presentation space at which to begin the copy.

Return Value

None

Example

The following is an example of using the SetText method.

```
//------------------------------------------------------------------
// ECLPS::SetText
//
// Update various input fields of the screen.
//------------------------------------------------------------------
void Sample65() {

ECLPS PS('A');              // PS object for connection A

// Note that an ECL error is thrown if we try to write to
// a protected field.

try {
   // Update first 2 input fields of the screen.    Note
   // fields are not erased before update.
  PS.SendKeys("[home]");
  PS.SetText("Field  1");
  PS.SendKeys("[tab]");
  PS.SetText("Field  2");
   // Note: Above 4 lines could also be written as:
   // PS.SendKeys("[home]Field 1[tab]Field 2");
   // But SetText() is faster, esp for long strings
}
catch (ECLErr Err) {
```

```
   printf("Failed to send keys: %s\n", Err.GetMsgText());
}

} // end sample

//-------------------------------------
```

ConvertPosToRowCol

The ConvertPosToRowCol method converts a position in the presentation space represented as a linear array to a position in the presentation space given in row and column coordinates. The position converted is in the presentation space for the connection associated with the ECLPS object.

Prototype

void ConvertPosToRowCol(ULONG pos, ULONG *row, ULONG *col)

Parameters

ULONG pos Position to convert in the presentation space represented as a linear array.

ULONG *row Converted row coordinate in the presentation space.

ULONG *col Converted column coordinate in the presentation space.

Return Value

None

Example

The following example shows how to convert a position in the presentation space represented as a linear array to a position shown in row and column coordinates.

```
///----------------------------------------------------------------------
// ECLPS::ConvertPosToRowCol
//
// Find a string in the presentation space and display the row/column
// coordinate of its location.
//----------------------------------------------------------------------
void Sample67() {

ECLPS PS('A');                  // PS Object
ULONG FoundPos;                 // Linear position
ULONG FoundRow,FoundCol;


FoundPos = PS.SearchText("IBM", TRUE);
if (FoundPos != 0) {
   PS.ConvertPosToRowCol(FoundPos, &FoundRow, &FoundCol);
   // Another way to do the same thing:
   FoundRow = PS.ConvertPosToRow(FoundPos);
   FoundCol = PS.ConvertPosToCol(FoundPos);

   printf("String found at row %lu column %lu (position %lu)\n",
            FoundRow, FoundCol, FoundPos);
}
else printf("String not found.\n");

} // end sample
```

ConvertRowColToPos

The ConvertRowColToPos method converts a position in the presentation space in row and column coordinates to a position in the presentation space represented as a linear array. The position converted is in the presentation space for the connection associated with the ECLPS object.

Prototype

ULONG ConvertRowColToPos(ULONG row, ULONG col)

Parameters

ULONG row Row coordinate to convert in the presentation space.

ULONG col Column coordinate to convert in the presentation space.

Return Value

ULONG Converted position in the presentation space represented as a linear array.

Example

The following example shows how to convert a position in the presentation space shown in row and column coordinates to a linear array position.

```
///--------------------------------------------------------------------
// ECLPS::ConvertRowColToPos
//
// Find a string in the presentation space and display the row/column
// coordinate of its location.
//--------------------------------------------------------------------
void Sample67() {

ECLPS PS('A');                  // PS Object
ULONG FoundPos;                    // Linear position
ULONG FoundRow,FoundCol;


FoundPos = PS.SearchText("IBM", TRUE);
if (FoundPos != 0) {
   PS.ConvertPosToRowCol(FoundPos, &FoundRow, &FoundCol);
   // Another way to do the same thing:
   FoundRow = PS.ConvertPosToRow(FoundPos);
   FoundCol = PS.ConvertPosToCol(FoundPos);

   printf("String found at row %lu column %lu (position %lu)\n",
            FoundRow, FoundCol, FoundPos);
}
else printf("String not found.\n");

} // end sample
```

ConvertPosToRow

This method takes a linear position value in the Presentation Space and returns the row in which it resides for the connection associated with the ECLPS object.

Prototype

ULONG ConvertPosToRow(ULONG Pos)

Parameters

ULONG Pos This is the linear position in the Presentation Space to convert.

Return Value

ULONG This is the row position for the linear position.

Example

The following is an example of using the ConvertPosToRow method.

```
///--------------------------------------------------------------------
// ECLPS::ConvertPosToRow
//
// Find a string in the presentation space and display the row/column
// coordinate of its location.
//--------------------------------------------------------------------
void Sample67() {

ECLPS PS('A');                  // PS Object
ULONG FoundPos;                    // Linear position
ULONG FoundRow,FoundCol;


FoundPos = PS.SearchText("IBM", TRUE);
if (FoundPos != 0) {
   PS.ConvertPosToRowCol(FoundPos, &FoundRow, &FoundCol);
   // Another way to do the same thing:
   FoundRow = PS.ConvertPosToRow(FoundPos);
   FoundCol = PS.ConvertPosToCol(FoundPos);

   printf("String found at row %lu column %lu (position %lu)\n",
            FoundRow, FoundCol, FoundPos);
}
else printf("String not found.\n");

} // end sample
```

ConvertPosToCol

This method takes a linear position value in the Presentation Space and returns the column in which it resides for the connection associated with the ECLPS object.

Prototype
ULONG ConvertPosToCol(ULONG Pos)

Parameters
ULONG Pos This is the linear position in the Presentation Space to convert.

Return Value
ULONG This is the column position for the linear position.

Example
The following is an example of using the ConvertPosToCol method.

```
///----------------------------------------------------------------------
/// ECLPS::ConvertPosToCol
//
// Find a string in the presentation space and display the row/column
// coordinate of its location.
//----------------------------------------------------------------------
void Sample67() {

ECLPS PS('A');                  // PS Object
ULONG FoundPos;                    // Linear position
ULONG FoundRow,FoundCol;


FoundPos = PS.SearchText("IBM", TRUE);
if (FoundPos != 0) {
   PS.ConvertPosToRowCol(FoundPos, &FoundRow, &FoundCol);
   // Another way to do the same thing:
   FoundRow = PS.ConvertPosToRow(FoundPos);
   FoundCol = PS.ConvertPosToCol(FoundPos);

   printf("String found at row %lu column %lu (position %lu)\n",
            FoundRow, FoundCol, FoundPos);
}
else printf("String not found.\n");

} // end sample
```

RegisterKeyEvent

The RegisterKeyEvent function registers an application-supplied object to receive notification of operator keystroke events. The application must construct an object derived from the ECLKeyNotify abstract base class. When an operator keystroke occurs, the NotifyEvent() method of the application supplied object is called. The application can choose to have the keystroke filtered or passed on and processed in the usual way. See “ECLKeyNotify Class” on page 84 for more details.

Implementation Restriction: Only one object may be registered to receive keystroke events at a time.

Prototype

void RegisterKeyEvent(ECLKeyNotify *NotifyObject)

Parameters

ECLKeyNotify *NotifyObject Application object derived from ECLKeyNotify class.

Return Value

None

Example

The following example shows how to register an application-supplied object to receive notification of operator keystroke events. See the“ECLKeyNotify Class” on page 84 for a RegisterKeyEvent example.

```
// This is the declaration of your class derived from ECLKeyNotify....
class MyKeyNotify: public ECLKeyNotify
{
public:
   // App can put parms on constructors if needed
   MyKeyNotify();                                          // Constructor
   ~MyKeyNotify();                                         // Destructor

   // App must define the NotifyEvent method
   int NotifyEvent(char KeyType[2], char KeyString[7]);    // Keystroke callback

private:
   // Whatever you like...
};
// this is the implementation of app methods...

int MyKeyNotify::NotifyEvent( char *KeyType, char *Keystring )
{
   if (...) {
      ...
      return 0;    // Remove keystroke (filter)
  }
  else
      ...
      return 1;    // Pass keystroke to emulator as usual
  }
}

// this would be the code in say, WinMain...

ECLPS        *pPS;                                      // Pointer to ECLPS object
MyKeyNotify *MyKeyNotifyObject;       // My key notification object,derived
                                      // from ECLKeyNotify

try {
   pPS = new ECLPS('A');                  // Create PS object for 'A' session

   // Register for keystroke events
   MyKeyNotifyObject = new MyKeyNotify();
  pPS->RegisterKeyEvent(MyKeyNotifyObject);

   // After this, MyKeyNotifyObject->NotifyEvent() will be called
```

```
   // for each operator keystroke...
}
catch (ECLErr HE) {
   // Just report the error text in a message box
   MessageBox( NULL, HE.GetMsgText(), "Error!", MB_OK );
}
```

UnregisterKeyEvent

The UnregisterKeyEvent method unregisters an application object previously registered for keystroke events with the RegisterKeyEvent function. A registered application notify object should not be destroyed without first calling this function to unregister it. If there is no notify object currently registered, or the registered object is not the NotifyObject passed in, this function does nothing (no error is thrown).

Prototype

virtual UnregisterKeyEvent(ECLKeyNotify *NotifyObject)

Parameters

ECLKeyNotify *NotifyObject Object currently registered for keystroke events.

Return Value

None

Example

See the“ECLKeyNotify Class” on page 84 for a UnregisterKeyEvent example.

(See RegisterKeyNotify for an example.)

GetFieldList

This method returns a pointer to an ECLFieldList object. The field list object can be used to iterate over the list of fields in the host presentation space. The ECLFieldList object returned by this function is automatically destroyed when the ECLPS object is destroyed. See “ECLFieldList Class” on page 77 for more information about this object.

Prototype

ECLFieldList *GetFieldList()

Parameters

None

Return Value

ECLFieldList * Pointer to ECLFieldList object.

Example

The following example shows how to return a pointer to an ECLFieldList object.

```
// ECLPS::GetFieldList
//
// Display number of fields on the screen.
//-------------------------------------------------------------------
void Sample68() {
```

```
ECLPS              *PS;                  // Pointer to PS object
ECLFieldList *FieldList;       // Pointer to field list object

try {
   PS = new ECLPS('A');                        // Create PS object for 'A'

   FieldList = PS->GetFieldList();         // Get pointer to field list
   FieldList->Refresh();                       // Build the field list

   printf("There are %lu fields on the screen of connection %c.\n",
     FieldList->GetFieldCount(),  PS->GetName());

  delete  PS;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}
} // end sample
```

ECLSession Class

ECLSession provides general emulator connection-related services and contains pointers to instances of other objects in the Host Access Class Library.

Derivation

ECLConnection

Properties

None

Usage Notes

Because ECLSession is derived from ECLConnection, you can obtain all the information contained in an ECLConnection object. See “ECLConnection Class” on page 23 for more information.

Although the objects ECLSession contains are capable of standing on their own, pointers to them exist in the ECLSession class. When an ECLSession object is created, ECLPS, ECLOIA, ECLXfer, and ECLWinMetrics objects are also created.

ECLSession Methods

The following section describes the methods that are valid for the ECLSession class.

ECLSession(char Name)
ECLSession(Long Handle)
~ECLSession()
ECLPS *GetPS()
ECLOIA *GetOIA()
ECLXfer *GetXfer()
ECLWinMetrics *GetWinMetrics()
void RegisterUpdateEvent(UPDATETYPE Type, ECLUpdateNotify *UpdateNotifyClass,
  BOOL InitEvent)
void UnregisterUpdateEvent(ECLUpdateNotify *UpdateNotifyClass,)

ECLSession Constructor

This method creates an ECLSession object from a connection name (a single, alphabetic character from A-Z) or a connection handle. There can be only one Personal Communications connection open with a given name. For example, there can only be one connection "A" open at a time.

Prototype

ECLSession(char Name)

ECLSession(long Handle)

Parameters

char Name One-character short name of the connection (A-Z).

long Handle Handle of an ECL connection.

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLSession::ECLSession          (Constructor)
//
// Build PS object from name.
//-------------------------------------------------------------------
void Sample73() {

ECLSession *Sess;           // Pointer to Session object for connection A
ECLPS          *PS;             // PS object pointer

try {
   Sess = new ECLSession('A');

   PS = Sess->GetPS();
   printf("Size of presentation space is %lu.\n", PS->GetSize());

  delete  Sess;
}
catch (ECLErr Err) {
```

```
   printf("ECL Error: %s\n", Err.GetMsgText());
}
} // end sample
```

ECLSession Destructor

This method destroys an ECLSession object.

Prototype

~ECLSession();

Parameters

None

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLSession:: ECLSession          (Destructor)
//
// Build PS object from name and then delete it.
//-------------------------------------------------------------------
void Sample74() {

ECLSession *Sess;            // Pointer to Session object for connection A
ECLPS          *PS;              // PS object pointer

try {
   Sess = new ECLSession('A');

   PS = Sess->GetPS();
   printf("Size of presentation space is %lu.\n", PS->GetSize());

  delete  Sess;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}
} // end sample
```

GetPS

This method returns a pointer to the ECLPS object contained in the ECLSession object. Use this method to access the ECLPS object methods. See “ECLPS Class” on page 101 for more information.

Prototype
ECLPS *GetPS()

Parameters
None

Return Value
ECLPS * ECLPS object pointer.

Example
```
//-------------------------------------------------------------------
// ECLSession::GetPS
//
// Get PS object from session object and use it.
//-------------------------------------------------------------------
void Sample69() {

ECLSession *Sess;           // Pointer to Session object for connection A
ECLPS      *PS;             // PS object pointer

try {
   Sess = new ECLSession('A');

   PS = Sess->GetPS();
   printf("Size of presentation space is %lu.\n", PS->GetSize());

  delete  Sess;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}
} // end sample
```

GetOIA

This method returns a pointer to the ECLOIA object contained in the ECLSession object. Use this method to access the ECLOIA methods. See “ECLOIA Class” on page 90 for more information.

Prototype
ECLOIA *GetOIA()

Parameters
None

Return Value
ECLOIA * ECLOIA object pointer.

Example

```
//-------------------------------------------------------------------
// ECLSession::GetOIA
//
// Get OIA object from session object and use it.
//-------------------------------------------------------------------
void Sample70() {

ECLSession *Sess;            // Pointer to Session object for connection A
ECLOIA         *OIA;              // OIA object pointer

try {
   Sess = new ECLSession('A');

   OIA = Sess->GetOIA();
   if (OIA->InputInhibited() == NotInhibited)
      printf("Input is not inhibited.\n");
  else
      printf("Input is inhibited.\n");

  delete  Sess;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}
} // end sample
```

GetXfer

This method returns a pointer to the ECLXfer object contained in the ECLSession object. Use this method to access the ECLXfer methods. See “ECLXfer Class” on page 164 for more information.

Prototype

ECLXfer *GetXfer()

Parameters

None

Return Value

ECLXfer * ECLXfer object pointer.

Example

```
//-------------------------------------------------------------------
// ECLSession::GetXfer
//
// Get OIA object from session object and use it.
//-------------------------------------------------------------------
void Sample71() {

ECLSession *Sess;            // Pointer to Session object for connection A
ECLXfer       *Xfer;           // Xfer object pointer

try {
```

```
   Sess = new ECLSession('A');

   Xfer = Sess->GetXfer();
   Xfer->SendFile("c:\\autoexec.bat", "AUTOEXEC BAT A", "(ASCII CRLF");

  delete  Sess;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}
} // end sample
```

GetWinMetrics

This method returns a pointer to the ECLWinMetrics object contained in the ECLSession object. Use this method to access the ECLWinMetrics methods. See “ECLWinMetrics Class” on page 143 for more information.

Prototype

ECLWinMetrics *GetWinMetrics()

Parameters

None

Return Value

ECLWinMetrics * ECLWinMetrics object pointer.

Example

```
//-------------------------------------------------------------------
// ECLSession::GetWinMetrics
//
// Get WinMetrics object from session object and use it.
//-------------------------------------------------------------------
void Sample72() {

ECLSession *Sess;             // Pointer to Session object for connection A
ECLWinMetrics *Metrics; // WinMetrics object pointer

try {
   Sess = new ECLSession('A');

   Metrics = Sess->GetWinMetrics();
   printf("Window height is %lu pixels.\n", Metrics->GetHeight());

  delete  Sess;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}
} // end sample
```

RegisterUpdateEvent

This member function registers an application object to receive notification of PS and/or OIA update events. To use this function, the application must create an object derivied from the ECLUpdateNotify class. A pointer to that object is then passed to this registration function. (*Implementation Restriction:* An application can register only one object for update event notification.)

The Type parameter allows an application to choose to receive events for PS updates, OIA updates, or both.

After a notify object has been registered with this function, it will be called whenever the PS and/or OIA is updated. The object will receive notification for all PS/OIA update events whether they are caused by the host or operator. Update events are not queued. An application should not assume it will receive a distinct event for every update. An update event is defined as one or more changes in the PS/OIA since the last time an update event was generated (or since registration if there is no prior update event).

The optional InitEvent parameter causes an initial event to be generated when the object is registered. The generated event type will be identical to the Type parameter specified on this function. This can be useful to synchronize an event object with the current state of the PS or OIA. If InitEvent is specified as FALSE, no initial event is generated when the object is registered. The default for this parameter is TRUE.

The application must call UnregisterUpdateEvent() before destroying the notification object. The object is automatically unregistered if the ECLSession object on which it is registered is destroyed.

See “ECLUpdateNotify Class” on page 138 for more information.

Prototype

void RegisterUpdateEvent(UPDATETYPE Type, ECLUpdateNotify *NotifyObject, BOOL InitEvent=TRUE)

Parameters

UPDATETYPE Type — Type of updates to generate events for:

- PSUpdate: PS updates only
- OIAUpdate: OIA updates only
- PSOIAUpdate: PS and OIA updates

ECLUpdateNotify *AppObject — This is a ptr to an object derived from the ECLUpdateNotify class.

BOOL InitEvent — This generates an initial event with the current state.

Return Value
None

Example
See “ECLUpdateNotify Class” on page 138 for an example.

UnregisterUpdateEvent
This member function unregisters an application object previously registered for PS/OIA update events with the RegisterUpdateEvent() function. A registered application notify object should not be destroyed without first calling this function to unregister it. If there is no notify object currently registered, or the registered object is not the *NotifyObject* passed in, this function does nothing (no error is thrown).

When a notify object is unregistered, its NotifyStop() member function will be called.

See “ECLUpdateNotify Class” on page 138 for more information.

Prototype
void UnregisterUpdateEvent(ECLUpdateNotify *NotifyObject)

Parameters
ECLUpdateNotify *NotifyObject This is the currently registered application notification object.

Return Value
None

Example
See “ECLUpdateNotify Class” on page 138 for an example.

ECLStartNotify Class

ECLStartNotify is an abstract base class. An application cannot create an instance of this class directly. To use this class, the application must define its own class which is derived from ECLStartNotify. The application must implement the NotifyEvent( ) member function in its derived class. It may also optionally implement NotifyError() and NotifyStop() member functions.

The ECLStartNotify class is used to allow an application to be notified of the starting and stopping of PComm connections. Start/stop events are generated whenever a PComm connection (window) is started or stopped by any means, including the ECLConnMgr start/stop methods.

To be notified of start/stop events, the application must perform the following steps:

1. Define a class derived from ECLStartNotify.
2. Implement the derived class and implement the NotifyEvent() member function.
3. Optionally implement the NotifyError() and/or NotifyStop() functions.
4. Create an instance of the derived class.
5. Register the instance with the ECLConnMgr::RegisterStartEvent() function.

The example shown demonstrates how this may be done. When the above steps are complete, each time a connection is started or stopped the applications NotifyEvent() member function will be called. The function is passed two parameters giving the handle of the connection, and a BOOL start/stop indicator. The application may perform any functions required in the NotifyEvent() procedure, including calling other ECL functions. Note that the application cannot prevent the stopping of a connection; the notification is made after the session is already stopped.

If an error is detected during event generation, the NotifyError() member function is called with an ECLErr object. Events may or may not continue to be generated after an error, depending on the nature of the error. When event generation terminates (either due to an error, by calling the ECLConnMgr::UnregisterStartEvent, or by destruction of the ECLConnMgr object) the NotifyStop() member function is called. However event notification is terminated, the NotifyStop() member function is always called, and the application object is unregistered.

If the application does not provide an implementation of the NotifyError() member function, the default implementation is used (a simple message box is displayed to the user). The application can override the default behavior by implementing the NotifyError() function in the applications derived class. Likewise, the default NotifyStop() function is used if the application does not provide this function (the default behavior is to do nothing).

Note that the application can also choose to provide its own constructor and destructor for the derived class. This can be useful if the application wants to store some instance-specific data in the class and pass that information as a parameter on the constructor. For example, the application may want to post a message to an application window when a start/stop event occurs. Rather than define the window handle as a global variable (so it would be visible to the NotifyEvent() function), the

applicati on can define a constructor for the class which takes the window handle and stores it in the class member data area.

The application must not destroy the notification object while it is registered to receive events.

Implementation Restriction: Currently the ECLConnMgr object allows only one notification object to be registered for a start/stop event notification. The ECLConnMgr::RegisterStartEvent will throw an error if a notify object is already registered for that ECLConnMgr object.

Derivation

ECLBase > ECLNotify > ECLStartNotify

Usage Notes

```
//-------------------------------------------------------------------
// ECLStartNotify class
//
// This sample demonstrates the use of:
//
// ECLStartNotify::NotifyEvent
// ECLStartNotify::NotifyError
// ECLStartNotify::NotifyStop
// ECLConnMgr::RegisterStartEvent
// ECLConnMgr::UnregisterStartEvent
//-------------------------------------------------------------------

//.....................................................................
// Define a class derived from ECLStartNotify
//.....................................................................
class MyStartNotify: public ECLStartNotify
{
public:
   // Define my own constructor to store instance data
  MyStartNotify(HANDLE  DataHandle);

   // We have to implement this function
   void       NotifyEvent(ECLConnMgr *CMObj, long ConnHandle,
                          BOOL  Started);

   // We will take the default behaviour for these so we
   // don't implement them in our class:
   // void NotifyError (ECLConnMgr *CMObj, long ConnHandle, ECLErr ErrObject);
   // void NotifyStop (ECLConnMgr *CMObj, int Reason);

private:
   // We will store our application data handle here
  HANDLE  MyDataH;
};

 //.....................................................................
MyStartNotify::MyStartNotify(HANDLE DataHandle)      // Constructor
//.....................................................................
{
   MyDataH = DataHandle;    // Save data handle for later use
}
```

```
//.....................................................................
void MyStartNotify::NotifyEvent(ECLConnMgr *CMObj, long ConnHandle,
                                BOOL  Started)
//.....................................................................
{
   // This function is called whenever a connection start or stops.

  if  (Started)
      printf("Connection %c started.\n", CMObj->ConvertHandle2ShortName(ConnHandle));
  else
      printf("Connection %c stopped.\n", CMObj->ConvertHandle2ShortName(ConnHandle));

  return;
}

 //.....................................................................
// Create the class and begin start/stop monitoring.
//.....................................................................
void Sample75() {

ECLConnMgr       CMgr;          // Connection manager object
MyStartNotify *Event;      // Ptr to my event handling object
HANDLE  InstData;              // Handle to application data block (for example)

try {
   Event = new MyStartNotify(InstData);    // Create event handler

   CMgr.RegisterStartEvent(Event);             // Register to get events

   // At this point, any connection start/stops will cause the
   // MyStartEvent::NotifyEvent() function to execute.    For
   // this sample, we put this thread to sleep during this
  //  time.

   printf("Monitoring connection start/stops for 60 seconds...\n");
  Sleep(60000);

   // Now stop event generation.
  CMgr.UnregisterStartEvent(Event);
   printf("Start/stop monitoring ended.\n");

   delete Event;    // Don't delete until after unregister!
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLStartNotify Methods

The following section describes the methods that are valid for the ECLStartNotify class.

```
            ECLStartNotfiy()
          ~ECLStartNotify()
virtual int NotifyEvent (
                    ECLConnMgr  *CMObj,
                    long ConnHandle,
                    BOOL Started) = 0
virtual void NotifyError  (
                    ECLConnMgr  *CMObj,
                    long ConnHandle,
                    ECLErr ErrObject)
virtual void NotifyStop   (
                    ECLConnMgr *CMObj
                    int Reason)
```

NotifyEvent

This method is a“pure virtual” member function (the application *must* implement this function in classes derived from ECLStartNotify). This function is called whenever a connection starts or stops and the object is registered for start/stop events. The Started BOOL is TRUE if the connection is started, or FALSE if is stopped.

Parameters

ECLConnMgr *CMObj — This is the pointer to ECLConnMgr object in which the event occurred.

long ConnHandle — This is the handle of the connection that started or stopped.

BOOL Started — This is TRUE if the connection is started, or FALSE if the connection is stopped.

Return Value

None

NotifyError

This method is called whenever the ECLConnMgr object detects an error event generation. The error object contains information about the error (see the ECLErr class description). Events may continue to be generated after the error, depending on the nature of the error. If event generation stops due to an error, the NotifyStop() function is called.

The ConnHandle contains the handle of the connection that is related to the error. This value may be zero if the error is not related to any specific connection.

An application can choose to implement this function or allow the ECLStartNotify base class to handle the error. The base class will display the error in a message box using the text supplied by the ECLErr::GetMsgText() function. If the application implements this function in its derived class it will override the base class function.

Parameters

ECLConnMgr *CMObj This is the ptr to ECLConnMgr object in which the error occurred.

long ConnHandle This is the handle of the connection related to the error or zero.

ECLErr ErrObject This is theECLErr object describing the error.

Return Value

None

NotifyStop

This method is called when event generation is stopped for any reason (for example, due to an error condition or a call to ECLConnMgr::UnregisterStartEvent).

Parameters

ECLConnMgr *CMObj This is the ptr to ECLConnMgr object that is stopping notification.

int Reason This is the unused zero.

Return Value

None

ECLUpdateNotify Class

ECLUpdateNotify Class ECLUpdateNotify is an abstract base class. An application cannot create an instance of this class directly. To use this class, the application must define its own class which is derived from ECLUpdateNotify. The application must implement the NotifyEven t() member function in its derived class. It may also optionally implement NotifyError() and NotifyStop() member functions.

The ECLUpdateNotify class is used to allow an application to be notified of updates to the presentation space (screen) or the Operator Information Area (OIA) indicators. Events are generated whenever the host screen is updated (by the host or the operator) and/or when any status indicator in the OIA changes.

To be notified of PS/OIA update events, the application must perform the following steps:

1. Define a class derived from ECLUpdateNotify.
2. Implement the derived class and implement the NotifyEvent() member function.
3. Optionally implement the NotifyError() and/or NotifyStop() functions.
4. Create an instance of the derived class.
5. Register the instance with the ECLSession::RegisterUpdateEvent() function. A parameter on this call can cause event registration to be restricted to just PS events or just OIA events.

The example shown demonstrates how this may be done. When the previous steps are complete, each time a PS/OIA update occures the applications NotifyEvent() member function will be called.

If an error is detected during event generation, the NotifyError() member function is called with an ECLErr object. Events may or may not continue to be generated after an error, depending on the nature of the error. When event generation terminates (eit her due to an error, by calling the ECLSession::UnregisterUpdateEvent, or by destruction of the ECLSession object) the NotifyStop() member function is called. However event notification is terminated, the NotifyStop() member function is always called, an d the application object is unregistered.

If the application does not provide an implementation of the NotifyError() member function, the default implementation is used (a simple message box is displayed to the user). The application can override the default behaviour by implementing the NotifyE rror() function in the applications derived class. Likewise, the default NotifyStop() function is used if the application does not provide this function (the default behaviour is to do nothing).

Note that the application can also choose to provide its own constructor and destructor for the derived class. This can be useful if the application wants to store some instance-specific data in the class and pass that information as a parameter on the c onstructor. For example, the application may want to post a message to an application window when an update occures. Rather than define the window handle as a global variable (so it would be visible to the NotifyEvent() function), the application can de fine a constructor for the class which takes the window handle and stores it in the class member data area.

The application must not destroy the notification object while it is registered to recieve events.

Implementation Restriction: Currently the ECLSession object allows only one notification object to be registered for update event notification. The ECLSession::RegisterUpdateEvent will throw an error if a notify object is already registered for that ECLS ession object.

Derivation

ECLBase>ECLNotify>ECLUpdate

Usage Notes

```
//-------------------------------------------------------------------
// ECLUpdateNotify class
//
// This sample demonstrates the use of:
//
// ECLUpdateNotify::NotifyEvent
// ECLUpdateNotify::NotifyError
// ECLUpdateNotify::NotifyStop
// ECLSession::RegisterStartEvent
// ECLSession::UnregisterStartEvent
//-------------------------------------------------------------------

//.....................................................................
// Define a class derived from ECLUpdateNotify
//.....................................................................
class MyUpdateNotify: public ECLUpdateNotify
{
public:
   // Define my own constructor to store instance data
  MyUpdateNotify(HANDLE  DataHandle);

   // We have to implement this function
  void       NotifyEvent(ECLSession  *SessObj,
                         UPDATETYPE  Type);

   // We will take the default behaviour for these so we
   // don't implement them in our class:
   // void NotifyError (ECLSession *SessObj, ECLErr ErrObject);
   // void NotifyStop (ECLSession *SessObj, int Reason);

private:
   // We will store our application data handle here
  HANDLE  MyDataH;
};

 //.....................................................................
MyUpdateNotify::MyUpdateNotify(HANDLE DataHandle)      // Constructor
//.....................................................................
{
   MyDataH = DataHandle;    // Save data handle for later use
}

//.....................................................................
void MyUpdateNotify::NotifyEvent(ECLSession *SessObj,
```

ECLUpdateNotify

```
                                        UPDATETYPE  Type)
//.....................................................................
{
   // This function is called whenever the PS/OIA is updated.
   // We just display a message whenever the OIA is updated.

   if ((Type == OIAUpdate) ||              // OIA was updated
         (Type == PSOIAUpdate))            // Both PS and OIA updated
      printf("OIA updated on connection %c.\n", SessObj->GetName());

   if ((Type == PSUpdate) ||               // PS was updated
         (Type == PSOIAUpdate))            // Both PS and OIA updated
      printf("PS    updated on connection %c.\n", SessObj->GetName());

  return;
}

 //.....................................................................
// Create the class and start monitoring OIA changes
//.....................................................................
void Sample76() {

ECLSession          *Session; // Ptr to Session object
MyUpdateNotify *Event;      // Ptr to my event handling object
HANDLE  InstData;                // Handle to application data block (for example)

try {
   Session = new ECLSession('A');              // Create a session object
   Event = new MyUpdateNotify(InstData); // Create event handler

   // Register for both PS and OIA updates
  Session->RegisterUpdateEvent(PSOIAUpdate,  Event);

   // At this point, any updates to PS or OIA by the
   // opreator or the host will cause the
   // MyUpdateEvent::NotifyEvent() function to execute.    For
   // this sample, we put this thread to sleep during this
  //  time.

   printf("Monitoring updates on connection A for 60 seconds...\n");
  Sleep(60000);

  Session->UnregisterUpdateEvent(Event);
   printf("Update monitoring ended.\n");

   delete Event;    // Don't delete until after unregister!
  delete  Session;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLUpdateNotify Methods

The following section describes the methods that are valid for the ECLUpdateNotify class.

```
                ECLUpdateNotify()
               ~ECLUpdateNotify()
virtual void NotifyEvent (
                ECLSession *SessObj,
                UPDATETYPE Type) = 0
virtual void NotifyError (
                ECLSession *SessionObj,
                ECLErr ErrObject)
virtual void NotifyStop  (
                ECLSession *SessObj,
                int Reason)
```

NotifyEvent

This method is a“pure virtual” member function (the application *must* implement this function in classes derived from ECLUpdateNotify). This function is called whenever the PS/OIA is updated and the object is registered for update events. The Type is an enumeration indicating what type of update occurred (PSUpdate for PS updates, OIAUpdate for OIA updates, and the PSOIAUpdate when both are updated).

Note that only event types requested in the ECLSession::RegisterUpdateEvent() call will actually cause events. For example, if PSUpdate is specified in the RegisterUpdateEvent() call, only PS updates will cause events and OIA updates will be ignored.

Parameters

ECLSession *SessObj This is the ptr to ECLSession object in which the event occurred.

UPPDATETYPE Type This is the enumeration indicating the type of event that occurred.

Return Value

None

NotifyError

This method is called whenever the ECLSession object detects an error during event generation. The error object contains information about the error (see the ECLErr class description). Events may continue to be generated after the error, depending on the nature of the error. If the event generation stops due to an error, the NotifyStop() function is called.

An application can choose to implement this function or allow the ECLUpdateNotify base class to handle the error. The base class will display the error in a message box using the text supplied by the ECLErr::GetMsgText() function. If the application implements this function in its derived class, it overrides the base function.

Parameters

ECLSession *SessObj This is the ptr to ECLSession object in which the error occurred.

ECLErr ErrObject This is the ECLErr object describing the error.

Return Value

None

NotifyStop

This method is called when event generation is stopped for any reason (for example, due to an error condition or a call to ECLSession::UnregisterUpdateEvent, etc.)

Implementation Note: The Reason code is currently unused and will be zero.

Parameters

ECLSession *SessObj This is the ptr to ECLSession object in which the error occurred.

int Reason This is unused (zero).

Return Value

None

ECLWinMetrics Class

The ECLWinMetrics class performs operations on a Personal Communications connection window. It allows you to perform window rectangle and position manipulation (for example, SetWindowRect, GetXpos or SetWidth), as well as window state manipulation (for example, SetVisible or IsRestored).

Derivation

ECLBase>ECLConnection>ECLWinMetrics

Properties

None

Usage Notes

Because ECLWinMetrics is derived from ECLConnection, you can obtain all the information contained in an ECLConnection object. See “ECLConnection Class” on page 23 for more information.

The ECLWinMetrics object is created for the connection identified upon construction. You may create an ECLWinMetrics object by passing either the connection ID (a single, alphabetical character from A-Z) or the connection handle, which is usually obtained from the ECLConnection object. There can be only one Personal Communications connection with a given name or handle open at a time.

Note: There is a pointer to the ECLWinMetrics object in the ECLSession class. If you just want to manipulate the connection window, create ECLWinMetrics on its own. If you want to do more, you may want to create an ECLSession object.

ECLWinMetrics Methods

The following methods apply to the ECLWinMetrics class.

ECLWinMetrics(char Name)
ECLWinMetrics(long Handle)
const char *GetWindowTitle()
void SetWindowTitle(char *NewTitle)
long GetXpos()
void SetXpos(long NewXpos)
long GetYpos()
void SetYpos(long NewYpos)
long GetWidth
void SetWidth(long NewWidth)
long GetHeight()
void SetHeight(long NewHeight)
void GetWindowRect(Long *left, Long *top, Long *right, Long *bottom)
void SetWindowRect(Long left, Long top, Long right, Long bottom)
BOOL IsVisible()
void SetVisible(BOOL SetFlag)
BOOL Active()
void SetActive(BOOL SetFlag)
BOOL IsMinimized()
void SetMinimized()
BOOL IsMaximized()
void SetMaximized()
BOOL IsRestored()
void SetRestored()

ECLWinMetrics Constructor

This method creates an ECLWinMetrics object from a connection name or connection handle. There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time.

Prototype

ECLWinMetrics(char Name)

ECLWinMetrics(long Handle)

Parameters

char Name One-character short name of the connection (A-Z).

long Handle Handle of an ECL connection.

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLWinMetrics::ECLWinMetrics      (Constructor)
//
// Build WinMetrics object from name.
//-------------------------------------------------------------------
void Sample77() {

ECLWinMetrics *Metrics;       // Ptr to object

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   printf("Window of connection A is %lu pixels wide.\n",
     Metrics->GetWidth());

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLWinMetrics Destructor

This method destroys a ECLWinMetrics object.

Prototype

~ECLWinMetrics()

Parameters

None

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLWinMetrics::ECLWinMetrics      (Destructor)
//
// Build WinMetrics object from name.
//-------------------------------------------------------------------
void Sample78() {

ECLWinMetrics *Metrics;       // Ptr to object

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   printf("Window of connection A is %lu pixels wide.\n",
     Metrics->GetWidth());
```

```
  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetWindowTitle

The GetWindowTitle method returns a pointer to a null terminate string containing the title that is currently in the title bar for the connection associated with the ECLWinMetrics object. Do not assume that the string returned is persistent over time. You must either make a copy of the string or make a call to this method each time you need it.

Prototype

```
const char *GetWindowTitle()
```

Parameters

None

Return Value

Pointer to null terminated string that contains the title.

Example

```
//-------------------------------------------------------------------
// ECLWinMetrics::GetWindowTitle
//
// Display current window title of connection A.
//-------------------------------------------------------------------
void Sample79() {

ECLWinMetrics *Metrics;       // Ptr to object

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   printf("Title of connection A is: %s\n",
     Metrics->GetWindowTitle());

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

SetWindowTitle

The SetWindowTitle method changes the title currently in the title bar for the connection associated with the ECLWinMetrics object to the title passed in the input parameter. A null string can be used to reset the title to the default title.

Prototype

void SetWindowTitle(char *NewTitle)

Parameters

char* NewTitle Null terminated title string.

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLWinMetrics::SetWindowTitle
//
// Change current window title of connection A.
//-------------------------------------------------------------------
void Sample80() {

ECLWinMetrics *Metrics;       // Ptr to object

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   // Get current title
   printf("Title of connection A is: %s\n", Metrics->GetWindowTitle());

   // Set new title
  Metrics->SetWindowTitle("New  Title");
   printf("New title is: %s\n", Metrics->GetWindowTitle());

   // Reset back to original title
  Metrics->SetWindowTitle("");
   printf("Returned title to: %s\n", Metrics->GetWindowTitle());

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

Usage Notes

If NewTitle is a nullstring, SetWindowTitle will restore the window title to its original setting.

GetXpos

The GetXpos method returns the *x* position of the upper left point of the connection window rectangle.

Prototype
long GetXpos()

Parameters
None

Return Value
long *x* Position of connection window.

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::GetXpos
//
// Move window 10 pixels.
//-------------------------------------------------------------------
void Sample81() {

ECLWinMetrics *Metrics;       // Ptr to object
long X, Y;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot move minimized or maximized window.\n");
  }
  else  {
      X = Metrics->GetXpos();
      Y = Metrics->GetYpos();
     Metrics->SetXpos(X+10);
     Metrics->SetYpos(Y+10);
  }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

SetXpos

The SetXpos method sets the *x* position of the upper left point of the connection window rectangle.

Prototype
void SetXpos(long NewXpos)

Parameters
long NewXpos The new *x* coordinate of the window rectangle.

Return Value
None

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::SetXpos
//
// Move window 10 pixels.
//-------------------------------------------------------------------
void Sample83() {

ECLWinMetrics *Metrics;       // Ptr to object
long X, Y;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot move minimized or maximized window.\n");
  }
  else  {
      X = Metrics->GetXpos();
      Y = Metrics->GetYpos();
     Metrics->SetXpos(X+10);
     Metrics->SetYpos(Y+10);
  }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetYpos

The GetYpos method returns the *y* position of the upper left point of the connection window rectangle.

Prototype
long GetYpos()

Parameters
None

Return Value
long *y* position of the connection window.

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::GetYpos
//
// Move window 10 pixels.
//-------------------------------------------------------------------
void Sample82() {

ECLWinMetrics *Metrics;       // Ptr to object
long X, Y;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot move minimized or maximized window.\n");
  }
  else  {
      X = Metrics->GetXpos();
      Y = Metrics->GetYpos();
     Metrics->SetXpos(X+10);
     Metrics->SetYpos(Y+10);
  }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

SetYpos

The SetYpos method sets the *y* position of the upper left point of the connection window rectangle.

Prototype
void SetYpos(long NewYpos)

Parameters
long NewYpos New *y* coordinate of the window rectangle.

Return Value
None

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::SetYpos
//
// Move window 10 pixels.
//-------------------------------------------------------------------
void Sample84() {

ECLWinMetrics *Metrics;       // Ptr to object
long X, Y;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot move minimized or maximized window.\n");
  }
  else  {
      X = Metrics->GetXpos();
      Y = Metrics->GetYpos();
     Metrics->SetXpos(X+10);
     Metrics->SetYpos(Y+10);
  }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetWidth

This method returns the width of the connection window rectangle.

Prototype
long GetWidth

Parameters
None

Return Value
long Width of the connection window.

Example

```
//---------------------------------------------------------------------
// ECLWinMetrics::GetWidth
//
// Make window 1/2 its current size.    Depending on display settings
// (Appearance->Display Setup menu) it may snap to a font that is
// not exactly the 1/2 size we specify.
//---------------------------------------------------------------------
void Sample85() {

ECLWinMetrics *Metrics;        // Ptr to object
long X, Y;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot size minimized or maximized window.\n");
  }
  else  {
      X = Metrics->GetWidth();
      Y = Metrics->GetHeight();
     Metrics->SetWidth(X/2);
     Metrics->SetHeight(Y/2);
  }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

SetWidth

The SetWidth method sets the width of the connection window rectangle.

Prototype

void SetWidth(long NewWidth)

Parameters

long NewWidth New width of the window rectangle.

Return Value

None

Example

```
//---------------------------------------------------------------------
// ECLWinMetrics::SetWidth
//
// Make window 1/2 its current size.    Depending on display settings
// (Appearance->Display Setup menu) it may snap to a font that is
// not exactly the 1/2 size we specify.
```

```
//-------------------------------------------------------------------
void Sample87() {

ECLWinMetrics *Metrics;       // Ptr to object
long X, Y;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot size minimized or maximized window.\n");
  }
  else  {
      X = Metrics->GetWidth();
      Y = Metrics->GetHeight();
     Metrics->SetWidth(X/2);
     Metrics->SetHeight(Y/2);
  }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

GetHeight

The GetHeight method returns the height of the connection window rectangle.

Prototype

long GetHeight()

Parameters

None

Return Value

long Height of the connection window.

Example

```
//-------------------------------------------------------------------
// ECLWinMetrics::GetHeight
//
// Make window 1/2 its current size.    Depending on display settings
// (Appearance->Display Setup menu) it may snap to a font that is
// not exactly the 1/2 size we specify.
//-------------------------------------------------------------------
void Sample86() {

ECLWinMetrics *Metrics;       // Ptr to object
long X, Y;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A
```

```
   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot size minimized or maximized window.\n");
   }
   else  {
       X = Metrics->GetWidth();
       Y = Metrics->GetHeight();
      Metrics->SetWidth(X/2);
      Metrics->SetHeight(Y/2);
   }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

SetHeight

This method sets the height of the connection window rectangle.

Prototype

void SetHeight(Long NewHeight)

Parameters

long NewHeight New height of the window rectangle.

Return Value

None

Example

The following example shows how to use the SetHeight method to set the height of the connection window rectangle.

```
ECLWinMetrics     *pWM;
ECLConnList   ConnList();

// Create using connection handle of first connection in the list of
// active connections
try {
   if ( ConnList.Count() != 0 ) {
      pWM = new ECLWinMetrics(ConnList.GetFirstSession()->GetHandle());

      // Set the height
      pWM->SetHeight(6081);
   }
}
catch (ECLErr ErrObj) {
   // Just report the error text in a message box
   MessageBox( NULL, ErrObj.GetMsgText(), "Error!", MB_OK );
}
```

GetWindowRect

This method returns the bounding points of the connection window rectangle.

Prototype

void GetWindowRect(Long *left, Long *top, Long *right, Long *bottom)

Parameters

long *left This output parameter is set to the left coordinate of the window rectangle.

long *top This output parameter is set to the top coordinate of the window rectangle.

long *right This output parameter is set to the right coordinate of the window rectangle.

long *bottom This output parameter is set to the bottom coordinate of the window rectangle.

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLWinMetrics::GetWindowRect
//
// Make window 1/2 its current size.    Depending on display settings
// (Appearance->Display Setup menu) it may snap to a font that is
// not exactly the 1/2 size we specify.    Also move the window.
//-------------------------------------------------------------------
void Sample88() {

ECLWinMetrics *Metrics;       // Ptr to object
long X, Y, Width, Height;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot size/move minimized or maximized window.\n");
  }
  else  {
      Metrics->GetWindowRect(&X, &Y, &Width, &Height);
      Metrics->SetWindowRect(X+10, Y+10,                // Move window
                             Width/2, Height/2); // Size window
  }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

SetWindowRect

This method sets the bounding points of the connection window rectangle.

Prototype

void SetWindowRect(long left, long top, long right, long bottom)

Parameters

long left The left coordinate of the window rectangle.

long top The top coordinate of the window rectangle.

long right The right coordinate of the window rectangle.

long bottom The bottom coordinate of the window rectangle.

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLWinMetrics::SetWindowRect
//
// Make window 1/2 its current size.    Depending on display settings
// (Appearance->Display Setup menu) it may snap to a font that is
// not exactly the 1/2 size we specify.    Also move the window.
//-------------------------------------------------------------------
void Sample89() {

ECLWinMetrics *Metrics;       // Ptr to object
long X, Y, Width, Height;

try {
   Metrics = new ECLWinMetrics('A');    // Create for connection A

   if (Metrics->IsMinimized() || Metrics->IsMaximized()) {
      printf("Cannot size/move minimized or maximized window.\n");
  }
  else  {
      Metrics->GetWindowRect(&X, &Y, &Width, &Height);
      Metrics->SetWindowRect(X+10, Y+10,                // Move window
                             Width/2, Height/2); // Size window
  }

  delete  Metrics;
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

IsVisible

This method returns the visibility state of the connection window.

Prototype
BOOL IsVisible()

Parameters
None

Return Value
Visibility state. TRUE value if the window is visible, FALSE value if the window is not visible

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::IsVisible
//
// Get current state of window, and then toggle it.
//-------------------------------------------------------------------
void Sample90() {

ECLWinMetrics Metrics('A');          // Window metrics class
BOOL   CurrState;

CurrState = Metrics.IsVisible(); // Get state
Metrics.SetVisible(!CurrState);    // Set state

} // end sample
```

SetVisible

This method sets the visibility state of the connection window.

Prototype
void SetVisible(BOOL SetFlag)

Parameters
BOOL SetFlag. TRUE for visible, FALSE for invisible.

Return Value
None

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::SetVisible
//
// Get current state of window, and then toggle it.
//-------------------------------------------------------------------
void Sample91() {

ECLWinMetrics Metrics('A');          // Window metrics class
BOOL   CurrState;
```

```
CurrState = Metrics.IsVisible(); // Get state
Metrics.SetVisible(!CurrState);    // Set state

} // end sample

//---------------------------------------------------------------------
```

Active

This method returns the focus state of the connection window.

Prototype
BOOL Active()

Parameters
None

Return Value
BOOL Focus state. TRUE if active, FALSE if not active.

Example

```
// ECLWinMetrics::IsActive
//
// Get current state of window, and then toggle it.
//---------------------------------------------------------------------
void Sample92() {

ECLWinMetrics Metrics('A');          // Window metrics class
BOOL   CurrState;

CurrState = Metrics.IsActive();    // Get state
Metrics.SetActive(!CurrState);      // Set state

} // end sample
```

SetActive

This method sets the focus state of the connection window.

Prototype
void SetActive(BOOL SetFlag)

Parameters
Bool SetFlag New state. TRUE for active, FALSE for inactive.

Return Value
None

Example
The following is an example of the SetActive method.

```
ECLWinMetrics   *pWM;
ECLConnList    ConnList();

// Create using connection handle of first connection in the list of
// active connections
try {
   if ( ConnList.Count() != 0 ) {
      pWM = new ECLWinMetrics(ConnList.GetFirstSession()->GetHandle());

      // Set to inactive if active
      if ( pWM->Active() )
         pWM->SetActive(FALSE);
  }
}
catch (ECLErr ErrObj) {
   // Just report the error text in a message box
   MessageBox( NULL, ErrObj.GetMsgText(), "Error!", MB_OK );
}
```

IsMinimized

This method returns the minimize state of the connection window.

Prototype
BOOL IsMinimized()

Parameters
None

Return Value
BOOL Minimize state. TRUE value returned if the window is minimized; FALSE value returned if the window is not minimized.

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::IsMinimized
//
// Get current state of window, and then toggle it.
//-------------------------------------------------------------------
void Sample93() {

ECLWinMetrics Metrics('A');          // Window metrics class
BOOL   CurrState;

CurrState = Metrics.IsMinimized();    // Get state
if (!CurrState)
   Metrics.SetMinimized();                  // Set state
else
  Metrics.SetRestored();

} // end sample
```

SetMinimized

This method sets the connection window to minimized

Prototype
void SetMinimized()

Parameters
None

Return Value
None

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::SetMinimized
//
// Get current state of window, and then toggle it.
//-------------------------------------------------------------------
void Sample94() {

ECLWinMetrics Metrics('A');           // Window metrics class
BOOL   CurrState;

CurrState = Metrics.IsMinimized();    // Get state
if (!CurrState)
   Metrics.SetMinimized();                  // Set state
else
  Metrics.SetRestored();

} // end sample
```

IsMaximized

This method returns the maximize state of the connection window.

Prototype
BOOL IsMaximized()

Parameters
None

Return Value
BOOL Maximize state. TRUE value if the window is maximized; FALSE value if the window is not maximized.

Example
```
// ECLWinMetrics::IsMaximized
//
// Get current state of window, and then toggle it.
//-------------------------------------------------------------------
void Sample97() {
```

```
ECLWinMetrics Metrics('A');          // Window metrics class
BOOL   CurrState;

CurrState = Metrics.IsMaximized();   // Get state
if (!CurrState)
   Metrics.SetMaximized();                 // Set state
else
  Metrics.SetMinimized();

} // end sample
```

SetMaximized

This method sets the connection window to maximized.

Prototype

void SetMaximized()

Parameters

None

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLWinMetrics::SetMaximized
//
// Get current state of window, and then toggle it.
//-------------------------------------------------------------------
void Sample98() {

ECLWinMetrics Metrics('A');          // Window metrics class
BOOL   CurrState;

CurrState = Metrics.IsMaximized();   // Get state
if (!CurrState)
   Metrics.SetMaximized();                 // Set state
else
  Metrics.SetMinimized();

} // end sample
```

IsRestored

This method returns the restore state of the connection window.

Prototype
BOOL IsRestored()

Parameters
None

Return Value
BOOL Restore state. TRUE value if the window is restored; FALSE value if the window is not restored.

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::IsRestored
//
// Get current state of window, and then toggle it.
//-------------------------------------------------------------------
void Sample95() {

ECLWinMetrics Metrics('A');          // Window metrics class
BOOL   CurrState;

CurrState = Metrics.IsRestored();    // Get state
if (!CurrState)
   Metrics.SetRestored();              // Set state
else
  Metrics.SetMinimized();

} // end sample
```

SetRestored

The SetRestored method sets the connection window to restored.

Prototype
void SetRestored()

Parameters
None

Return Value
None

Example
```
//-------------------------------------------------------------------
// ECLWinMetrics::SetRestored
//
// Get current state of window, and then toggle it.
//-------------------------------------------------------------------
void Sample96() {

ECLWinMetrics Metrics('A');          // Window metrics class
BOOL   CurrState;
```

```
CurrState = Metrics.IsRestored();    // Get state
if (!CurrState)
    Metrics.SetRestored();                    // Set state
else
   Metrics.SetMinimized();

} // end sample

//-------------------------------------------------------------------
```

ECLXfer Class

ECLXfer provides file transfer services.

Derivation

ECLBase>ECLConnection>ECLXfer

Properties

None

Usage Notes

Because ECLXfer is derived from ECLConnection, you can obtain all the information contained in an ECLConnection object. See “ECLConnection Class” on page 23 for more information.

The ECLXfer object is created for the connection identified upon construction. You may create an ECLXfer object by passing either the connection ID (a single, alphabetic character from A-Z) or the connection handle, which is usually obtained from the ECLConnList object. There can be only one Personal Communications connection with a given name 0or handle open at a time.

Note: There is a pointer to the ECLXfer object in the ECLSession class. If you only want to manipulate the connection window, create an ECLXfer object on its own. If you want to do more, you may want to create an ECLSession object.

ECLXfer Methods

The following section describes the methods that are valid for the ECLXfer class.

ECLXfer(char Name)
ECLXfer(long Handle)
int SendFile(char *PCFile, char *HostFile, char *Options)
int ReceiveFile(char *PCFile, char *HostFile, char *Options)

ECLXfer Constructor

This method creates an ECLXfer object from a connection ID (a single, alphabetic character from A-Z) or a connection handle. There can be only one Personal Communications connection open with a given ID. For example, there can be only one connection “A” open at a time.

Prototype

ECLXfer(char Name)

ECLXfer(long Handle)

Parameters

char Name One-character short name of the connection (A-Z).

long Handle Handle of an ECL connection.

Return Value

None

Example

```
//------------------------------------------------------------------------
// ECLXfer::ECLXfer                (Constructor)
//
// Build ECLXfer object from a connection name.
//------------------------------------------------------------------------
void Sample99() {

ECLXfer    *Xfer;                   // Pointer to Xfer object

try {
   Xfer = new ECLXfer('A');    // Create object for connection A
   printf("Created ECLXfer for connection %c.\n", Xfer->GetName());

   delete Xfer;                     // Delete Xfer object
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

ECLXfer Destructor

This method destroys an ECLXfer object.

Prototype

~ECLXfer();

Parameters

None

Return Value

None

Example

```
//-------------------------------------------------------------------
// ECLXfer:: ECLXfer                (Destructor)
//
// Build ECLXfer object from a connection name.
//-------------------------------------------------------------------
void Sample100() {

ECLXfer    *Xfer;                    // Pointer to Xfer object

try {
   Xfer = new ECLXfer('A');    // Create object for connection A
   printf("Created ECLXfer for connection %c.\n", Xfer->GetName());

   delete Xfer;                      // Delete Xfer object
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

SendFile

This method sends a file from the workstation to the host

Prototype

int SendFile(char *PCFile, char *HostFile, char *Options)

Parameters

char *PCFile Pointer to a string containing the workstation file name to be sent to the host.

char *HostFile Pointer to a string containing the host file name to be created or updated on the host.

char *Options Pointer to a string containing the options to be used during the transfer.

Return Value

int EHLLAPI return code as documented in the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for the SendFile EHLLAPI function.

Example

```
//-------------------------------------------------------------------
// ECLXfer::SendFile
//
// Send a file to a VM/CMS host with ASCII translation.
//-------------------------------------------------------------------
void Sample101() {

ECLXfer    *Xfer;                    // Pointer to Xfer object
int Rc;

try {
   Xfer = new ECLXfer('A');    // Create object for connection A

  printf("Sending  file...\n");
   Rc = Xfer->SendFile("c:\\autoexec.bat", "autoexec bat a", "(ASCII CRLF QUIET");
   switch (Rc) {
  case  2:
      printf("File transfer failed, error in parameters.\n", Rc);
     break;
  case  3:
      printf("File transfer sucessfull.\n");
     break;
  case  4:
      printf("File transfer sucessfull, some records were segmented.\n");
     break;
  case  5:
      printf("File transfer failed, workstation file not found.\n");
     break;
  case  27:
      printf("File transfer cancelled or timed out.\n");
     break;
  default:
      printf("File transfer failed, code %u.\n", Rc);
     break;
   } // case

   delete Xfer;                    // Delete Xfer object
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

Usage Notes

File transfer options are host-dependent. The following is a list of some of the valid host options for a VM/CMS host:

ASCII
CRLF
APPEND
LRECL
RECFM
CLEAR/NOCLEAR
PROGRESS
QUIET

Refer to the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for the list of supported hosts and associated file transfer options.

ReceiveFile

This method receives a file from the host and sends the file to the workstation.

Prototype

int ReceiveFile(char *PCFile, char *HostFile, char *Options)

Parameters

char *PCFile Pointer to a string containing the workstation file name to be sent to the host.

char *HostFile Pointer to a string containing the host file name to be created or updated on the host.

char *Options Pointer to a string containing the options to be used during the transfer.

Return Value

int EHLLAPI return code as documented in the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for the ReceiveFile EHLLAPI function.

Example

```
//-------------------------------------------------------------------
// ECLXfer::ReceiveFile
//
// Receive file from a VM/CMS host with ASCII translation.
//-------------------------------------------------------------------
void Sample102() {

ECLXfer    *Xfer;                    // Pointer to Xfer object
int Rc;

try {
   Xfer = new ECLXfer('A');    // Create object for connection A

  printf("Receiving  file...\n");
   Rc = Xfer->ReceiveFile("c:\\temp.txt", "temp text a", "(ASCII CRLF QUIET");
```

```
   switch (Rc) {
  case  2:
       printf("File transfer failed, error in parameters.\n", Rc);
      break;
  case  3:
       printf("File transfer sucessfull.\n");
      break;
  case  4:
       printf("File transfer sucessfull, some records were segmented.\n");
      break;
  case  27:
       printf("File transfer cancelled or timed out.\n");
      break;
  default:
       printf("File transfer failed, code %u.\n", Rc);
      break;
   } // case

   delete Xfer;                        // Delete Xfer object
}
catch (ECLErr Err) {
   printf("ECL Error: %s\n", Err.GetMsgText());
}

} // end sample
```

Usage Notes

File transfer options are host-dependent. The following is a list of some of the valid host options for a VM/CMS host:

ASCII
CRLF
APPEND
LRECL
RECFM
CLEAR/NOCLEAR
PROGRESS
QUIET

Refer to the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for the list of supported hosts and associated file transfer options.

ECLXfer

Chapter 3. Host Access Class Library Automation Objects

The Host Access Class Library Automation Objects allow the Personal Communications product to support Microsoft's COM-based automation technology (formerly known as OLE automation). The ECL Automation Objects are a series of automation servers that allow automation controllers, for example, Microsoft's Visual Basic, to programmatically access Personal Communications' data and functionality.

An example of this would be sending keys to Personal Communications' presentation space. This can be accomplished by manually typing keys in the Personal Communications window, but it can also be automated through the appropriate Personal Communications automation server (autECLPS in this case). Using Visual Basic you can create the autECLPS object and then call the SendKeys method in that object with the string that is to be placed in the presentation space.

In other words, applications that are enabled for controlling the automation protocol (automation controller) can control some of Personal Communications' operations (automation server). Personal Communications supports Visual Basic Script, which uses ECL Automation objects. Refer to Personal Communications' Macro/Script support for more details.

Personal Communications offers several automation servers to accomplish this. These servers are implemented as real-world, intuitive objects with methods and properties that control Personal Communications' operability. Each object begins with autECL, for automation Host Access Class Library. The objects are as follows:

autECLConnList, Connection List, on page 173 contains a list of Personal Communications connections for a given system. This is contained by autECLConnMgr, but may be created independently of autECLConnMgr.

autECLConnMgr, Connection Manager, on page 181 provides methods and properties to manage Personal Communications connections for a given system. A connection in this context is a Personal Communications window.

autECLFieldList, Field List, on page 184 performs operations on fields in an emulator presentation space.

autECLOIA, Operator Information Area, on page 195 provides methods and properties to query and manipulate the Operator Information Area. This is contained by autECLSession, but may be created independently of autECLSession.

autECLPS, Presentation Space, on page 207 provides methods and properties to query and manipulate the presentation space for the related Personal Communications connection. This contains a list of all the fields in the presentation space. It is contained by autECLSession, but may be created independently of autECLSession.

autECLSession, Session, on page 221 provides general session-related functionality and information. For convenience, it contains the autECLPS, autECLOIA, autECLXfer, and autECLWinMetrics objects.

autECLWinMetrics, Window Metrics, on page 230 provides methods to query the window metrics of the Personal Communications session associated with

© Copyright IBM Corp. 1997

this object. For example, use this object to minimize or maximize a Personal Communications window. This is contained by autECLSession, but may be created independently of autECLSession.

autECLXfer, File Transfer, on page 243 provides methods and properties to transfer files between the host and the workstation over the Personal Communications connection associated with this file transfer object. This is contained by autECLSession, but may be created independently of autECLsession.

The following is a graphical representation of the autECL objects:

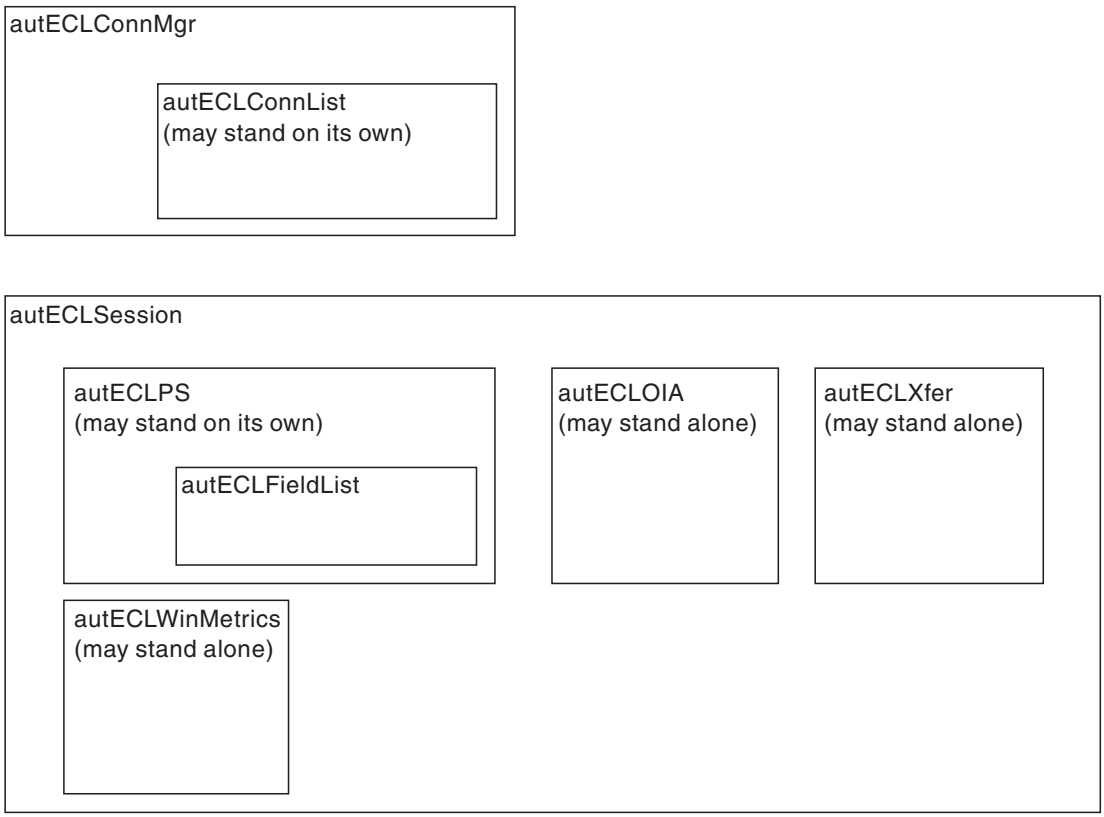


Figure 3. Host Access Class Library Automation Objects

This chapter describes each object's methods and properties in detail and is intended to cover all potential users of the automation object. Because the most common way to use the object is through a scripting application such as Visual Basic, all examples are shown using a Visual Basic format.

autECLConnList Class

autECLConnList contains information about all started connections. Its name in the registry is PCOMM.autECLConnList.

The autECLConnList object contains a collection of information about connections to a host. Each element of the collection represents a single connection (emulator window). A connection in this list may be in any state (for example, stopped or disconnected). All started connections appear in this list. The list element contains the state of the connection.

An autECLConnList object provides a static snapshot of current connections. The list is not dynamically updated as connections are started and stopped. The Refresh method is automatically called upon construction of the autECLConnList object. If you use the autECLConnList object right after its construction, your list of connections is current. However, you should call the Refresh method in the autECLConnList object before accessing its other methods if some time has passed since its construction to ensure that you have current data. Once you have called Refresh you may begin walking through the collection

Properties

This section describes the properties for the autECLConnList object.

Type	Name	Attributes
Long	Count	Read-only

Count

This is the number of connections present in the autECLConnList collection for the last call to the Refresh method. The Count property is a Long data type and is read-only. The following example uses the Count property.

```
Dim autECLConnList as Object
Dim Num as Long

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

autECLConnList.Refresh
Num = autECLConnList.Count
```

The following table shows Collection Element Properties, which are valid for each item in the list.

Type	Name	Attributes
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Boolean	Started	Read-only
Boolean	CommStarted	Read-only
Boolean	APIEnabled	Read-only
Boolean	Ready	Read-only

Name

This collection element property is the connection name string of the connection. Personal Communications only returns the short character ID (A-Z) in the string. There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time. Name is a String data type and is read-only. The following example uses the Name collection element property.

```
Dim    Str as String
Dim autECLConnList as Object
Dim Num as Long

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

autECLConnList.Refresh
Str = autECLConnList(1).Name
```

Handle

This collection element property is the handle of the connection. There can be only one Personal Communications connection open with a given handle. Handle is a Long data type and is read-only. The following example uses the Handle property.

```
Dim autECLConnList as Object
Dim Hand as Long

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

autECLConnList.Refresh
Hand = autECLConnList(1).Handle
```

ConnType

This collection element property is the connection type. This type may change over time. ConnType is a String data type and is read-only. The following example shows the ConnType property.

```
Dim    Type as String
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

autECLConnList.Refresh
Type = autECLConnList(1).ConnType
```

Connection types for the Conntype property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation

CodePage

This collection element property is the code page of the connection. This code page may change over time. CodePage is a Long data type and is read-only. The following example shows the CodePage property.

```
Dim    CodePage as Long
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

autECLConnList.Refresh
CodePage = autECLConnList(1).CodePage
```

Started

This collection element property indicates whether the emulator window is started. The value is True if the window is open; otherwise, it is False. Started is a Boolean data type and is read-only. The following example shows the Started property.

```
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
autECLConnList.Refresh

' This code segment checks to see if is started.
' The results are sent to a text box called Result.
If Not autECLConnList(1).Started Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

CommStarted

This collection element property indicates the status of the connection to the host. The value is True if the host is connected; otherwise, it is False. CommStarted is a Boolean data type and is read-only. The following example shows the CommStarted property.

```
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
autECLConnList.Refresh

' This code segment checks to see if communications are connected
' The results are sent to a text box called CommConn.
If Not autECLConnList(1).CommStarted Then
      CommConn.Text = "No"
Else
      CommConn.Text = "Yes"
End If
```

APIEnabled

This collection element property indicates whether the emulator is API-enabled. A connection may be enabled or disabled depending on the state of its API settings (in a Personal Communications window, choose File -> API Settings). The value is True if the emulator is enabled; otherwise, it is False. APIEnabled is a Boolean data type and is read-only. The following example shows the APIEnabled property.

```
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
autECLConnList.Refresh

' This code segment checks to see if API is enabled.
' The results are sent to a text box called Result.
If Not autECLConnList(1).APIEnabled Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

Ready

This collection element property indicates whether the emulator window is started, API-enabled, and connected. This property checks for all three properties. The value is True if the emulator is ready; otherwise, it is False. Ready is a Boolean data type and is read-only. The following example shows the Ready property.

```
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
autECLConnList.Refresh

' This code segment checks to see if X is ready.
' The results are sent to a text box called Result.
If Not autECLConnList(1).Ready Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

autECLConnList Methods

The following section describes the methods that are valid for the autECLConnList object.

void Refresh()
Object FindConnectionByHandle(Long Handle)
Object FindConnectionByName((String Name)

Collection Element Methods

The following collection element methods are valid for each item in the list.

void StartCommunication()
void StopCommunication()

Refresh

The Refresh method gets a snapshot of all the started connections.

Note: You should call this method before accessing the autECLConnList collection to ensure that you have current data.

Prototype

void Refresh()

Parameters

None

Return Value

None

Example

The following example shows how to use the Refresh method to get a snapshot of all the started connections.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object

Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
```

FindConnectionByHandle

This method finds an element in the autECLConnList object for the handle passed in the **Hand** parameter. This method is commonly used to see if a given connection is “alive” in the system.

Prototype
Object FindConnectionByHandle(Long Hand)

Parameters
Long Hand Handle to search for in the list.

Return Value
Object Collection element dispatch object.

Example
The following example shows how to find an element by the connection handle.

```
Dim Hand as Long
Dim autECLConnList as Object
Dim ConnObj as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the collection
autECLConnList.Refresh
' Assume Hand obtained earlier
Set ConnObj = autECLConnList.FindConnectionByHandle(Hand)
Hand = ConnObj.Handle
```

FindConnectionByName
This method finds an element in the autECLConnList object for the name passed in the **Name** parameter. This method is commonly used to see if a given connection is “alive” in the system.

Prototype
Object FindConnectionByName(String Name)

Parameters
String Name Name to search for in the list.

Return Value
Object Collection element dispatch object.

Example
The following example shows how to find an element in the autECLConnList object by the connection name.

```
Dim Hand as Long
Dim autECLConnList as Object
Dim ConnObj as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the collection
autECLConnList.Refresh
' Assume Hand obtained earlier
Set ConnObj = autECLConnList.FindConnectionByName("A")
Hand = ConnObj.Handle
```

StartCommunication

The StartCommunication collection element method connects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Connect.

Prototype
void StartCommunication()

Parameters
None

Return Value
None

Example
The following example shows how to connect a PCOMM emulator session to the host.

```
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the session
autECLConnList.Refresh
StartCommunication()
autECLConnList(1)
```

StopCommunication

The StopCommunication collection element method disconnects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Disconnect.

Prototype
void StopCommunication()

Parameters
None

Return Value
None

Example
The following example shows how to connect a PCOMM emulator session to the host.

autECLConnList

```
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the session
autECLConnList.Refresh
StartCommunication()
autECLConnList(1)
```

autECLConnMgr Class

autECLConnMgr manages all Personal Communications connections on a given machine. It contains methods relating to the connection management such as starting and stopping connections. It also creates an autECLConnList object to enumerate the list of all known connections on the system (see “autECLConnList Class” on page 173). Its name in the registry is PCOMM.autECLConnMgr.

Properties

This section describes the properties for the autECLConnMgr object.

Type	Name	Attributes
autECLConnList Object	autECLConnList	Read-only

autECLConnList

The autECLConnMgr object contains an autECLConnList object. See “autECLConnList Class” on page 173 for details on its methods and properties. The property has a value of autECLConnList, which is an autECLConnList dispatch object. The following example shows this property.

```
Dim Mgr as Object
Dim Num as Long

Set Mgr = CreateObject("PCOMM.autECLConnMgr ")

Mgr.autECLConnList.Refresh
Num = Mgr.autECLConnList.Count
```

autECLConnMgr Methods

The following section describes the methods that are valid for autECLConnMgr.

void StartConnection(String ConfigParms)
void StopConnection(Long Handle, String StopParms)

StartConnection

This member function starts a new Personal Communications emulator window. The ConfigParms string contains connection configuration information as explained under 182.

Prototype

void StartConnection(String ConfigParms)

Parameters

String ConfigParms Configuration string.

Return Value

None

Usage Notes

The configuration string is implementation-specific. Different implementations of the autECL objects may require different formats or information in the configuration string. The new emulator is started upon return from this call, but it may or may not be connected to the host.

For Personal Communications, the configuration string has the following format:

PROFILE=['']<filename>['] [CONNNAME=<c>] [WINSTATE=<MAX|MIN|RESTORE|HIDE>]

Optional parameters are enclosed in square brackets []. The parameters are separated by at least one blank. Parameters may be in upper, lower, or mixed case and may appear in any order. The meaning of each parameter is as follows:

PROFILE=<filename>: Names the Personal Communications workstation profile (.WS file), which contains the configuration information. This parameter is not optional; a profile name must be supplied. If the file name contains blanks the name must be enclosed in single quotation marks. The <filename> value may be either the profile name with no extension, the profile name with the .WS extension, or the fully qualified profile name path.

CONNNAME=<c> specifies the short ID of the new connection. This value must be a single, alphabetic character (A-Z). If this value is not specified, the next available connection ID is assigned automatically.

WINSTATE=<MAX|MIN|RESTORE|HIDE> specifies the initial state of the emulator window. The default if this parameter is not specified is RESTORE.

Example
The following example shows how to start a new Personal Communications emulator window.

```
Dim Mgr as Object
Dim Obj as Object
Dim Hand as Long

Set Mgr = CreateObject("PCOMM.autECLConnMgr ")
Mgr.StartConnection("profile=coax connname=e")
```

StopConnection
The StopConnection method stops (terminates) the emulator window identified by the connection handle. See 183 for contents of the StopParms string.

Prototype
void StopConnection(Long Handle, String StopParms)

Parameters
Long Handle Handle of the connection to be stopped.

String StopParms Stop parameters.

Return Value
None

Usage Notes
The stop parameter string is implementation-specific. Different implementations of the autECL objects may require a different format and contents of the parameter string. For Personal Communications the string has the following format:

```
[SAVEPROFILE=<YES|NO|DEFAULT>]
```

Optional parameters are enclosed in square brackets []. The parameters are separated by at least one blank. Parameters may be in upper, lower, or mixed case and may appear in any order. The meaning of each parameter is as follows:

SAVEPROFILE=<YES|NO|DEFAULT> controls the saving of the current configuration back to the workstation profile (.WS file). This causes the profile to be updated with any configuration changes you may have made. If NO is specified, the connection is stopped and the profile is not updated. If YES is specified, the connection is stopped and the profile is updated with the current (possibly changed) configuration. If DEFAULT is specified, the update option is controlled by the File->Save On Exit emulator menu option. If this parameter is not specified, DEFAULT is used.

Example
The following example shows how to stop the emulator window identified by the connection handle.

```
Dim Mgr as Object
Dim Hand as Long

Set Mgr = CreateObject("PCOMM.autECLConnMgr ")
 ' Assume Hand obtained earlier
Mgr.StopConnection(Hand, "saveprofile=no");
```

autECLFieldList Class

autECLFieldList performs operations on fields in an emulator presentation space. This object does not stand on its own. It is contained by autECLPS, and can only be accessed through an autECLPS object. autECLPS can stand alone or be contained by autECLSession.

autECLFieldList contains a collection of all the fields on a given presentation space. Each element of the collection contains the elements shown in Collection Element Properties.

An autECLFieldList object provides a static snapshot of what the presentation space contained when the Refresh method was called.

Note: You should call the Refresh method in the autECLFieldList object before accessing its elements to ensure that you have current field data. Once you have called Refresh, you may begin walking through the collection.

Properties

This section describes the properties and the collection element properties for the autECLFieldList object.

Type	Name	Attributes
Long	Count	Read-only

Count

This property is the number of fields present in the autECLFieldList collection for the last call to the Refresh method. Count is a Long data type and is read-only. The following example shows this property.

```
Dim NumFields as long
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
NumFields = autECLPSObj.autECLFieldList.Count
```

The following properties are collection element properties and are valid for each item in the list.

Type	Name	Attributes
Long	StartRow	Read-only
Long	StartCol	Read-only
Long	EndRow	Read-only
Long	EndCol	Read-only
Long	Length	Read-only
Boolean	Modified	Read-only
Boolean	Protected	Read-only
Boolean	Numeric	Read-only
Boolean	HighIntensity	Read-only
Boolean	PenDetectable	Read-only
Boolean	Display	Read-only

StartRow

This collection element property is the row position of the first character in a given field in the autECLFieldList collection. StartRow is a Long data type and is read-only. The following example shows this property.

```
Dim StartRow as Long
Dim StartCol as Long
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   StartRow = autECLPSObj.autECLFieldList(1).StartRow
   StartCol = autECLPSObj.autECLFieldList(1).StartCol
Endif
```

StartCol

This collection element property is the column position of the first character in a given field in the autECLFieldList collection. StartCol is a Long data type and is read-only. The following example shows this property.

```
Dim StartRow as Long
Dim StartCol as Long
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   StartRow = autECLPSObj.autECLFieldList(1).StartRow
   StartCol = autECLPSObj.autECLFieldList(1).StartCol
Endif
```

EndRow

This collection element property is the row position of the last character in a given field in the autECLFieldList collection. EndRow is a Long data type and is read-only. The following example shows this property.

```
Dim EndRow as Long
Dim EndCol as Long
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   EndRow = autECLPSObj.autECLFieldList(1).EndRow
   EndCol = autECLPSObj.autECLFieldList(1).EndCol
Endif
```

EndCol

This collection element property is the column position of the last character in a given field in the autECLFieldList collection. EndCol is a Long data type and is read-only. The following example shows this property.

```
Dim EndRow as Long
Dim EndCol as Long
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   EndRow = autECLPSObj.autECLFieldList(1).EndRow
   EndCol = autECLPSObj.autECLFieldList(1).EndCol
Endif
```

Length

This collection element property is the length of a given field in the autECLFieldList collection. Length is a Long data type and is read-only. The following example shows this property.

```
Dim Len as Long
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   Len = autECLPSObj.autECLFieldList(1).Length
Endif
```

Modified

This collection element property indicates if a given field in the autECLFieldList collection has a modified attribute. Modified is a Boolean data type and is read-only. The following example shows this property.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   If ( autECLPSObj.autECLFieldList(1).Modified ) Then
      ' do whatever
  Endif
Endif
```

Protected

This collection element property indicates if a given field in the autECLFieldList collection has a protected attribute. Protected is a Boolean data type and is read-only. The following example shows this property.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   If ( autECLPSObj.autECLFieldList(1).Protected ) Then
      ' do whatever
  Endif
Endif
```

Numeric

This collection element property indicates if a given field in the autECLFieldList collection has a numeric input only attribute. Numeric is a Boolean data type and is read-only. The following example shows this property.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   If ( autECLPSObj.autECLFieldList(1).Numeric ) Then
      ' do whatever
  Endif
Endif
```

HighIntensity

This collection element property indicates if a given field in the autECLFieldList collection has a high intensity attribute. HighIntensity is a Boolean data type and is read-only. The following example shows this property.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   If ( autECLPSObj.autECLFieldList(1).HighIntensity ) Then
      ' do whatever
  Endif
Endif
```

PenDetectable

This collection element property indicates if a given field in the autECLFieldList collection has a pen detectable attribute. PenDetectable is a Boolean data type and is read-only. The following example shows this property.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   If ( autECLPSObj.autECLFieldList(1).PenDetectable ) Then
      ' do whatever
  Endif
Endif
```

Display

This collection element property indicates whether a given field in the autECLFieldList collection has a display attribute. Display is a Boolean data type and is read-only. The following example shows this property.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh(1)
If (Not autECLPSObj.autECLFieldList.Count = 0 ) Then
   If ( autECLPSObj.autECLFieldList(1).Display ) Then
      ' do whatever
  Endif
Endif
```

autECLFieldList Methods

The following section describes the methods that are valid for the autECLFieldList object.

void Refresh()
Object FindFieldByRowCol(Long Row, Long Col)
Object FindFieldByText(String text, [optional] Long Direction, [optional] Long StartRow, [optional] Long StartCol)

Collection Element Methods

The following collection element methods are valid for each item in the list.

String GetText()
void SetText(String Text)

Refresh

The Refresh method gets a snapshot of all the fields.

Note: You should call the Refresh method before accessing the field collection to ensure that you have current field data.

Prototype

void Refresh()

Parameters

None

Return Value

None

Example

The following example shows how to get a snapshot of all the fields for a given presentation space for a given plane.

```
Dim NumFields as long
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and get the number of fields
autECLPSObj.autECLFieldList.Refresh()
NumFields = autECLPSObj.autECLFieldList.Count
```

FindFieldByRowCol

This method searches the autECLFieldList object for a field containing the given row and column coordinates. The value returned is a collection element object in the autECLFieldList collection.

Prototype

Object FindFieldByRowCol(Long Row, Long Col)

Parameters

Long Row Field row to search for.

Long Col Field column to search for.

Return Value

Object Dispatch object for the autECLFieldList collection item.

Example

The following example shows how to search the autECLFieldList object for a field containing the given row and column coordinates.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object
Dim FieldElement as Object

Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList)

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and search for field at row 2 col 1
autECLPSObj.autECLFieldList.Refresh(1)
Set FieldElement = autECLPSObj.autECLFieldList.FindFieldByRowCol( 2, 1 )
FieldElement.SetText("IBM")
```

FindFieldByText

This method searches the autECLFieldList object for a field containing the string passed in as **Text**. The value returned is a collection element object in the autECLFieldList collection.

Prototype

Object FindFieldByText(String Text, [optional] Long Direction, [optional] Long StartRow, [optional] Long StartCol)

Parameters

String Text The text string to search for.

Long StartRow Row position in the presentation space at which to begin the search.

Long StartCol Column position in the presentation space at which to begin the search.

Long Direction Direction in which to search. Values are **1** for search forward, **2** for search backward

Return Value
Object Dispatch object for the autECLFieldList collection item.

Example
The following example shows how to search the autECLFieldList object for a field containing the string passed in as text.

```
Dim autECLPSObj as Object
Dim autECLConnList as Object
Dim FieldElement as Object

Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and search for field with text
autECLPSObj.autECLFieldList.Refresh(1)
set FieldElement = autECLPSObj.autECLFieldList.FindFieldByText "IBM"

' Or... search starting at row 2 col 1
set FieldElement = autECLPSObj.autECLFieldList.FindFieldByText "IBM", 2, 1
' Or... search starting at row 2 col 1 going backwards
set FieldElement = autECLPSObj.autECLFieldList.FindFieldByText "IBM", 2, 2, 1

FieldElement.SetText("Hello.")
```

GetText

The collection element method GetText retrieves the characters of a given field in the autECLFieldList collection.

Prototype
String GetText()

Parameters
None

Return Value
String Field text.

Example
The following example shows how to use the GetText method.

```
Dim autECLPSObj as Object
Dim TestStr as String

' Initialize the connection
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
autECLPSObj.SetConnectionByName("A")

autECLPSObj.autECLFieldList.Refresh()
TextStr = autECLPSObj.autECLFieldList(1).GetText()
```

SetText

This method populates a given field in the autECLFieldList collection with the character string passed in as text. If the text exceeds the length of the field, the text is truncated.

Prototype

void SetText(String Text)

Parameters

String text String to set in field

Return Value

None

Example

The following example shows how to populate the field in the autECLFieldList collection with the character string passed in as text.

```
Dim NumFields as Long
Dim autECLPSObj as Object
Dim autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the list and set the first field with some text
autECLPSObj.autECLFieldList.Refresh(1)
autECLPSObj.autECLFieldList(1).SetText("IBM is a cool company")
```

autECLOIA Class

The autECLOIA object retrieves status from the Host Operator Information Area. Its name in the registry is PCOMM.autECLOIA.

You must initially set the connection for the object you create. Use SetConnectionByName or SetConnectionByHandle to initialize your object. The connection may be set only once. After the connection is set, any further calls to the set connection methods cause an exception. If you do not set the connection and try to access a property or method, an exception is also raised.

Note: The autECLOIA object in the autECLSession object is set by the autECLSession object.

The following example shows how to create and set the autECLOIA object in Visual Basic.

```
DIM    autECLOIA as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
autECLOIA.SetConnectionByName("A")
```

Properties

This section describes the properties for the autECLOIA object.

Type	Name	Attributes
Boolean	Alphanumeric	Read-only
Boolean	APL	Read-only
Boolean	Katakana	Read-only
Boolean	Hiragana	Read-only
Boolean	DBCS	Read-only
Boolean	UpperShift	Read-only
Boolean	Numeric	Read-only
Boolean	CapsLock	Read-only
Boolean	InsertMode	Read-only
Boolean	CommErrorReminder	Read-only
Boolean	MessageWaiting	Read-only
Long	InputInhibited	Read-only
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Boolean	Started	Read-only
Boolean	CommStarted	Read-only
Boolean	APIEnabled	Read-only
Boolean	Ready	Read-only

Alphanumeric

This property queries the operator information area to determine whether the field at the cursor location is alphanumeric. Alphanumeric is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

If autECLOIA.Alphanumeric Then...
```

APL

This property queries the operator information area to determine whether the keyboard is in APL mode. APL is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if the keyboard is in APL mode
if autECLOIA.APL Then...
```

Katakana

This property queries the operator information area to determine whether Katakana characters are enabled. Katakana is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if Katakana characters are available
if autECLOIA.Katakana Then...
```

Hiragana

This property queries the operator information area to determine whether Hiragana characters are enabled. Hiragana is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if Hiragana characters are available
if autECLOIA.Hiragana Then...
```

DBCS

This property queries the operator information area to determine whether the field at the cursor location is DBCS. DBCS is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if DBCS is available
if autECLOIA.DBCS Then...
```

UpperShift

This property queries the operator information area to determine whether the keyboard is in uppershift mode. Uppershift is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if the keyboard is in uppershift mode
If autECLOIA.UpperShift then...
```

Numeric

This property queries the operator information area to determine whether the field at the cursor location is numeric. Numeric is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if the cursor location is a numeric field
If autECLOIA.Numeric Then...
```

CapsLock

This property queries the operator information area to determine if the keyboard CapsLock key is on. CapsLock is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if the caps lock
If autECLOIA.CapsLock Then...
```

InsertMode

This property queries the operator information area to determine whether if the keyboard is in insert mode. InsertMode is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if in insert mode
If autECLOIA.InsertMode Then...
```

CommErrorReminder

This property queries the operator information area to determine whether a communications error reminder condition exists. CommErrorReminder is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if comm error
If autECLOIA.CommErrorReminder Then...
```

MessageWaiting

This property queries the operator information area to determine whether the message waiting indicator is on. This can only occur for 5250 connections. MessageWaiting is a Boolean data type and is read-only. The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object
Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if message waiting
If autECLOIA.MessageWaiting Then...
```

InputInhibited

This property queries the operator information area to determine whether keyboard input is inhibited. InputInhibited is a Long data type and is read-only. The following table shows valid values for InputInhibited.

Not Inhibited	0
System Wait	1
Communication Check	2
Program Check	3
Machine Check	4
Other Inhibit	5

The following example shows this property.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object
Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)

' Check if input inhibited
If not autECLOIA.InputInhibited = 0 Then...
```

Name

This property is the connection name string of the connection for which autECLOIA was set. Personal Communications only returns the short character ID (A-Z) in the string. There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time. Name is a String data type and is read-only. The following example shows this property.

```
DIM    Name as String
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
Obj.SetConnectionByName("A")

' Save the name
Name = Obj.Name
```

Handle

This is the handle of the connection for which the autECLOIA object was set. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time. Handle is a Long data type and is read-only. The following example shows this property.

```
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
Obj.SetConnectionByName("A")

' Save the handle
Hand = Obj.Handle
```

ConnType

This is the connection type for which autECLOIA was set. This type may change over time. ConnType is a String data type and is read-only. The following example shows this property.

```
DIM    Type as String
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
Obj.SetConnectionByName("A")
' Save the type
Type = Obj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation

CodePage

This is the code page of the connection for which autECLOIA was set. This code page may change over time. CodePage is a Long data type and is read-only. The following example shows this property.

```
DIM    CodePage as Long
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
Obj.SetConnectionByName("A")
' Save the code page
CodePage = Obj.CodePage
```

Started

This indicates whether the emulator window is started. The value is True if the window is open; otherwise, it is False. Started is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is started.
' The results are sent to a text box called Result.
If Obj.Started = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

CommStarted

This indicates the status of the connection to the host. The value is True if the host is connected; otherwise, it is False. CommStarted is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if communications are connected
' for A.    The results are sent to a text box called
' CommConn.
If Obj.CommStarted = False Then
      CommConn.Text = "No"
Else
      CommConn.Text = "Yes"
End If
```

APIEnabled

This indicates whether the emulator is API-enabled. A connection may be enabled or disabled depending on the state of its API settings (in a Personal Communications window, choose File -> API Settings). The value is True if the emulator is enabled; otherwise, it is False. APIEnabled is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is API enabled.
' The results are sent to a text box called Result.
If Obj.APIEnabled = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

Ready

This indicates whether the emulator window is started, API-enabled, and connected. This property checks for all three properties. The value is True if the emulator is ready; otherwise, it is False. Ready is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is ready.
' The results are sent to a text box called Result.
If Obj.Ready = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

autECLOIA Methods

The following section describes the methods that are valid for autECLOIA.

void SetConnectionByName (String Name)
void SetConnectionByHandle (Long Handle)
void StartCommunication()
void StopCommunication()

SetConnectionByName

The SetConnectionByName method uses the connection name to set the connection for a newly created autECLOIA object. In Personal Communications this connection name is the short connection ID (character A-Z). There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time.

Note: Do not call this if using the autECLOIA object in autECLSession.

Prototype

void SetConnectionByName(String Name)

Parameters

String Name One-character string short name of the connection (A-Z).

Return Value

None

Example

The following example shows how to use the connection name to set the connection for a newly created autECLOIA object.

```
DIM    autECLOIA as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")

' Initialize the connection
autECLOIA.SetConnectionByName("A")
' For example, see if its num lock is on
If ( autECLOIA.NumLock = True ) Then
   'your logic here...
Endif
```

SetConnectionByHandle

The SetConnectionByHandle method uses the connection handle to set the connection for a newly created autECLOIA object. In Personal Communications this connection handle is a Long integer. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time.

Note: Do not call this if using the autECLOIA object in autECLSession.

Prototype
void SetConnectionByHandle(Long Handle)

Parameters
Long Handle Long integer value of the connection to be set for the object.

Return Value
None

Example
The following example shows how to use the connection handle to set the connection for a newly created autELCOIA object.

```
DIM    autECLOIA as Object
DIM    autECLConnList as Object

Set autECLOIA = CreateObject("PCOMM.autECLOIA")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLOIA.SetConnectionByHandle(autECLConnList(1).Handle)
' For example, see if its num lock is on
If ( autECLOIA.NumLock = True ) Then
   'your logic here...
Endif
```

StartCommunication
The StartCommunication collection element method connects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Connect.

Prototype
void StartCommunication()

Parameters
None

Return Value
None

Example
None

```
Dim OIAObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set OIAObj = CreateObject("PCOMM.autECLOIA")

' Initialize the session
autECLConnList.Refresh
OIAObj.SetConnectionByHandle(autECLConnList(1).Handle)

OIAObj.StartCommunication()
```

StopCommunication

The StopCommunication collection element method disconnects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Disconnect.

Prototype

void StopCommunication()

Parameters

None

Return Value

None

Example

The following example shows how to connect a PCOMM emulator session to the host.

```
Dim OIAObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set OIAObj = CreateObject("PCOMM.autECLOIA")

' Initialize the session
autECLConnList.Refresh
OIAObj.SetConnectionByHandle(autECLConnList(1).Handle)

OIAObj.StopCommunication()
```

autECLPS Class

autECLPS performs operations on a presentation space. Its name in the registry is PCOMM.autECLPS.

You must initially set the connection for the object you create. Use SetConnectionByName or SetConnectionByHandle to initialize your object. The connection may be set only once. After the connection is set, any further calls to the SetConnection methods cause an exception. If you do not set the connection and try to access a property or method, an exception is also raised.

Notes:

1. In the presentation space, the first row coordinate is row 1 and the first column coodinate is column 1. Therefore, the top, left position has a coordinate of row 1, column 1.
2. The autECLPS object in the autECLSession object is set by the autECLSession object.

The following is an example of how to create and set the autECLPS object in Visual Basic.

```
DIM    autECLPSObj as Object
DIM    NumRows as Long
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
' Initialize the connection
autECLPSObj .SetConnectionByName("A")
' For example, get the number of rows in the PS
NumRows = autECLPSObj.NumRows
```

Properties

This section describes the properties of the autECLPS object.

Type	Name	Attributes
Object	autECLFieldList	Read-only
Long	NumRows	Read-only
Long	NumCols	Read-only
Long	CursorPosRow	Read-only
Long	CursorPosCol	Read-only
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Boolean	Started	Read-only
Boolean	CommStarted	Read-only
Boolean	APIEnabled	Read-only
Boolean	Ready	Read-only

NumRows

This is the number of rows in the presentation space for the connection associated with the autECLPS object. NumRows is a Long data type and is read-only. The following example shows this property.

```
Dim    autECLPSObj as Object
Dim    autECLConnList as Object
Dim    Rows as Long
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
Rows = autECLPSObj.NumRows
```

NumCols

This is the number of columns in the presentation space for the connection associated with the autECLPS object. NumCols is a Long data type and is read-only. The following example shows this property.

```
Dim    autECLPSObj as Object
Dim    autECLConnList as Object
Dim    Cols as Long
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
Cols = autECLPSObj.NumCols
```

CursorPosRow

This is the current row position of the cursor in the presentation space for the connection associated with the autECLPS object. CursorPosRow is a Long data type and is read-only. The following example shows this property.

```
Dim    autECLPSObj as Object
Dim    autECLConnList as Object
Dim    CurPosRow as Long
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
CurPosRow = autECLPSObj.CursorPosRow
```

CursorPosCol

This is the current column position of the cursor in the presentation space for the connection associated with the autECLPS object. CursorPosCol is a Long data type and is read-only. The following example shows this property.

```
Dim    autECLPSObj as Object
Dim    autECLConnList as Object
Dim    CurPosCol as Long
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
CurPosCol = autECLPSObj.CursorPosColumn
```

autECLFieldList

This is the field collection object for the connection associated with the autECLPS object. See “autECLFieldList Class” on page 184 for details. The following example shows this object.

```
Dim    autECLPSObj as Object
Dim    autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

' Build the field list
CurPosCol = autECLPSObj.autECLFieldList.Refresh(1)
```

Name

This is the connection name string of the connection for which autECLPS was set. Personal Communications only returns the short character ID (A-Z) in the string. There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time. Name is a String data type and is read-only. The following example shows this property.

```
DIM    Name as String
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
Obj.SetConnectionByName("A")

' Save the name
Name = Obj.Name
```

Handle

This is the handle of the connection for which the autECLPS object was set. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time. Handle is a Long data type and is read-only. The following example shows this property.

```
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
Obj.SetConnectionByName("A")

' Save the connection handle
Hand = Obj.Handle
```

ConnType

This is the connection type for which autECLPS was set. This connection type may change over time. ConnType is a String data type and is read-only. The following example shows this property.

```
DIM    Type as String
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
Obj.SetConnectionByName("A")
' Save the type
Type = Obj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation

CodePage

This is the code page of the connection for which autECLPS was set. This code page may change over time. CodePage is a Long data type and is read-only. The following example shows this property.

```
DIM    CodePage as Long
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
Obj.SetConnectionByName("A")
' Save the code page
CodePage = Obj.CodePage
```

Started

This indicates if the connection emulator window is started. The value is True if the window is open; otherwise, it is False. Started is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is started.
' The results are sent to a text box called Result.
If Obj.Started = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

CommStarted

This indicates the status of the connection to the host. The value is True if the host is connected; otherwise, it is False. CommStarted is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if communications are connected
' for A.    The results are sent to a text box called
' CommConn.
If Obj.CommStarted = False Then
      CommConn.Text = "No"
Else
      CommConn.Text = "Yes"
End If
```

APIEnabled

This indicates if the emulator is API-enabled. A connection may be enabled or disabled depending on the state of its API settings (in a Personal Communications window, choose File -> API Settings). The value is True if API is enabled; otherwise, it is False. APIEnabled is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is API enabled.
' The results are sent to a text box called Result.
If Obj.APIEnabled = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

Ready

This indicates whether the emulator window is started, API enabled and connected. This checks for all three properties. The value is True if the emulator is ready; otherwise, it is False. Ready is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is ready.
' The results are sent to a text box called Result.
If Obj.Ready = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

autECLPS Methods

The following section describes the methods that are valid for the autECLPS object.

void SetConnectionByName (String Name)
void SetConnectionByHandle (Long Handle)
void SetCursorPos(Long Row, Long Col)
void SendKeys(String text, [optional] Long row, [optional] Long col)
boolean SearchText(String text, [optional] Long Dir, [optional] Long row, [optional] Long col)
String GetText([optional] Long row, [optional] Long col, [optional] Long lenToGet)
void SetText(String Text, [optional] Long Row, [optional] Long Col)
String GetTextRect(Long StartRow, Long StartCol, Long EndRow, Long EndCol)
void StartCommunication()
void StopCommunication()

SetConnectionByName

This method uses the connection name to set the connection for a newly created autECLPS object. In Personal Communications this connection name is the short ID (character A-Z). There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time.

Note: Do not call this if using the autECLPS object in autECLSession.

Prototype

void SetConnectionByName(String Name)

Parameters

String Name One-character string short name of the connection (A-Z).

Return Value

None

Example

The following example shows how to set the connection for a newly created autECLPS object using the connection name.

```
DIM    autECLPSObj as Object
DIM    NumRows as Long
Set autECLPSObj = CreateObject("PCOMM.autECLPS")

' Initialize the connection
autECLPSObj.SetConnectionByName("A")
' For example, get the number of rows in the PS
NumRows = autECLPSObj.NumRows
```

SetConnectionByHandle

This method uses the connection handle to set the connection for a newly created autECLPS object. In Personal Communications this connection handle is a Long integer. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time.

Note: Do not call this if using the autECLPS object in autECLSession.

Prototype
void SetConnectionByHandle(Long Handle)

Parameters
Long Handle Long integer value of the connection to be set for the object.

Return Value
None

Example
The following example shows how to set the connection for a newly created autECLPS object using the connection handle.

```
DIM    autECLPSObj as Object
DIM    autECLConnList as Object
DIM    NumRows as Long
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection with the first in the list
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
' For example, get the number of rows in the PS
NumRows = autECLPSObj.NumRows
```

SetCursorPos

The SetCursorPos method sets the position of the cursor in the presentation space for the connection associated with the autECLPS object. The position set is in row and column units.

Prototype
void SetCursorPos(Long Row, Long Col)

Parameters
Long Row The row position of the cursor in the presentation space.

Long Col The column position of the cursor in the presentation space.

Return Value
None

Example
The following example shows how to set the position of the cursor in the presentation space for the connection associated with the autECLPS object.

```
DIM    autECLPSObj as Object
DIM    autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection with the first in the list
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
autECLPSObj.SetCursorPos 2, 1
```

SendKeys

The SendKeys method sends a string of keys to the presentation space for the connection associated with the autECLPS object. This method allows you to send mnemonic keystrokes to the presentation space. See Appendix A, “Sendkeys Mnemonic Keywords” on page 319 for a list of these keystrokes.

Prototype

void SendKeys(String text, [optional] Long row, [optional] Long col)

Parameters

String text String of keys to send to the presentation space.

Long Row Row position to send keys to the presentation space. This parameter is optional. The default is the current cursor row position. If row is specified, col must also be specified.

Long Col Column position to send keys to the presentation space. This parameter is optional. The default is the current cursor column position. If col is specified, row must also be specified.

Return Value

None

Example

The following example shows how to use the SendKeys method to send a string of keys to the presentation space for the connection associated with the autECLPS object.

```
Dim    autECLPSObj as Object
Dim    autECLConnList as Object
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
autECLPSObj.SendKeys "IBM is a really cool company", 3, 1
```

SearchText

The SearchText method searches for the first occurrence of text in the presentation space for the connection associated with the autECLPS object. The search is case sensitive. If text is found, the method returns a TRUE value. It returns a FALSE value if no text is found. If the optional row and column parameters are used, **row** and **col** are also returned, indicating the position of the text if it was found.

Prototype

boolean SearchText(String text, [optional] Long Dir, [optional] Long Row, [optional] Long Col)

Parameters

String text String to search for.

Long Dir Direction in which to search. Must either be **1** for search forward or **2** for search backward. This parameter is optional. The default is 1 for Forward.

Long Row Row position at which to start the search in the presentation space. The row of found text is returned if the search is successful. This parameter is optional. If row is specified, col must also be specified.

Long Col Column position at which to start the search in the presentation space. The column of found text is returned if the search is successful. This parameter is optional. If col is specified, row must also be specified.

Return Value

TRUE if text is found, FALSE if text is not found.

Example

The following example shows how to search for text in the presentation space for the connection associated with the autECLPS object.

```
Dim    autECLPSObj as Object
Dim    autECLConnList as Object
Dim    Row as Long
Dim    Col as Long
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)

// Search forward in the PS from the start of the PS.    If found
// then call a hypothetical found routine, if not found, call a hypothetical

// not found routine.
row = 3
col = 1
If ( autECLPSObj.SearchText "IBM", 1, row, col) Then
   Call FoundFunc (row, col)
Else
  Call  NotFoundFunc
Endif
```

GetText

The GetText method retrieves characters from the presentation space for the connection associated with the autECLPS object.

Prototype
String GetText([optional] Long Row, [optional] Long Col, [optional] Long LenToGet)

Parameters
Long Row Row position at which to start the retrieval in the presentation space. This parameter is optional.

Long Col Column position at which to start the retrieval in the presentation space. This parameter is optional.

Long LenToGet Number of characters to retrieve from the presentation space. This parameter is optional. The default is the length of the array passed in as BuffLen.

Return Value
String Text from the PS.

Example
The following example shows how to retrieve a string from the presentation space for the connection associated with the autECLPS object.

```
Dim   autECLPSObj as Object
Dim   PSText as String

' Initialize the connection
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
autECLPSObj.SetConnectionByName("A")

PSText = autECLPSObj.GetText(2,1,50)
```

SetText

The SetText method sends a string to the presentation space for the connection associated with the autECLPS object. Althought this method is similar to the SendKeys method, this method does not send mnemonic keystrokes (for example, [enter] or [pf1]).

Prototype
void SetText(String Text, [optional] Long Row, [optional] Long Col)

Parameters
String Text Character array to send.

Long Row The row at which to begin the retrieval from the presentation space. This parameter is optional. The default is the current cursor row position.

Long Col The column position at which to begin the retrieval from the presentation space. This parameter is optional. The default is the current cursor column position.

Return Value
None

Example
The following example shows how to search for text in the presentation space for the connection associated with the autECLPS object.

```
Dim autECLPSObj as Object

'Initialize the connection
Set autECLPSObj = CreateObject("PCOMM.autECLPS")
autECLPSObj.SetConnectionByName("A")
autECLPSObj.SetText"IBM is great", 2, 1
```

GetTextRect

The GetTextRect method retrieves characters from a rectangular area in the presentation space for the connection associated with the autECLPS object. No wrapping takes place in the text retrieval; only the rectangular area is retrieved.

Prototype
String GetTextRect(Long StartRow, Long StartCol, Long EndRow, Long EndCol)

Parameters

Long Startrow	Row at which to begin the retrieval in the presentation space.
Long Startcol	Column at which to begin the retrieval in the presentation space.
Long Endrow	Row at which to end the retrieval in the presentation space.
Long Endcol	Column at which to end the retrieval in the presentation space.

Return Value
String PS Text.

Example
The following example shows how to retrieve characters from a rectangular area in the presentation space for the connection associated with the autECLPS object.

```
Dim    autECLPSObj as Object
Dim   PSText  String

' Initialize the connection
Set autECLPSObj = CreateObject ("PCOMM.autELCPS")
autECLPSObj.SetConnectionByName("A")

PSText = GetTextRect(1,1,2,80)
```

StartCommunication

The StartCommunication collection element method connects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Connect.

Prototype

void StartCommunication()

Parameters

None

Return Value

None

Example

The following example shows how to connect a PCOMM emulator session to the host.

```
Dim PSObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set PSObj = CreateObject("PCOMM.autECLPS")

' Initialize the session
autECLConnList.Refresh
PSObj.SetConnectionByHandle(autECLConnList(1).Handle)

PSObj.StartCommunication()
```

StopCommunication

The StopCommunication collection element method disconnects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Disconnect.

Prototype

void StopCommunication()

Parameters

None

Return Value

None

Example

The following example shows how to connect a PCOMM emulator session to the host.

```
Dim PSObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set PSObj = CreateObject("PCOMM.autECLPS")

' Initialize the session
autECLConnList.Refresh
PSObj.SetConnectionByHandle(autECLConnList(1).Handle)

PSObj.StopCommunication()
```

autECLSession Class

The autECLSession object provides general emulator related services and contains pointers to other key objects in the Host Access Class Library. Its name in the registry is PCOMM.autECLSession.

Although the objects that autECLSession contains are capable of standing on their own, pointers to them exist in the autECLSession class. When an autECLSession object is created, autECLPS, autECLOIA, autECLXfer and autECLWindowMetrics objects are also created. Refer to them as you would any other property.

Note: You must initially set the connection for the object you create. Use SetConnectionByName or SetConnectionByHandle to initialize your object. The connection may be set only once. After the connection is set, any further calls to the SetConnection methods cause an exception. If you do not set the connection and try to access an autECLSession property or method, an exception is also raised.

The following example shows how to create and set the autECLSession object in Visual Basic.

```
DIM    SessObj as Object
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' For example, set the host window to minimized
SessObj.autECLWinMetrics.Minimized = True
```

Properties

This section describes the properties for the autECLSession object.

Type	Name	Attributes
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Boolean	Started	Read-only
Boolean	CommStarted	Read-only
Boolean	APIEnabled	Read-only
Boolean	Ready	Read-only
Object	autECLPS	Read-only
Object	autECLOIA	Read-only
Object	autECLXfer	Read-only
Object	autECLWinMetrics	Read-only

Name

This property is the connection name string of the connection for which autECLSession was set. Personal Communications only returns the short character ID (A-Z) in the string. There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time. Name is a String data type and is read-only. The following example shows this property.

```
DIM    Name as String
DIM    SessObj as Object
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")

' Save the name
Name = SessObj.Name
```

Handle

This is the handle of the connection for which the autECLSession object was set. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time. Handle is a Long data type and is read-only. The following example shows this property.

```
DIM    SessObj as Object
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")

' Save the session handle
Hand = SessObj.Handle
```

ConnType

This is the connection type for which autECLXfer was set. This type may change over time. ConnType is a String data type and is read-only. The following example shows this property.

```
DIM    Type as String
DIM    SessObj as Object

Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' Save the type
Type = SessObj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation

CodePage

This is the code page of the connection for which autECLXfer was set. This code page may change over time. CodePage is a Long data type and is read-only. The following example shows this property.

```
DIM    CodePage as Long
DIM    SessObj as Object
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' Save the code page
CodePage = SessObj.CodePage
```

Started

This indicates whether the emulator window is started. The value is True if the window is open; otherwise, it is False. Started is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    SessObj as Object

Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' This code segment checks to see if A is started.
' The results are sent to a text box called Result.
If SessObj.Started = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

CommStarted

This indicates the status of the connection to the host. The value is True if the host is connected; otherwise, it is False. CommStarted is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    SessObj as Object

Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")

' This code segment checks to see if communications are connected
' for session A.    The results are sent to a text box called
' CommConn.
If SessObj.CommStarted = False Then
      CommConn.Text = "No"
Else
      CommConn.Text = "Yes"
End If
```

APIEnabled

This indicates whether the emulator is API-enabled. A connection may be enabled or disabled depending on the state of its API settings (in a Personal Communications window, choose File -> API Settings). The value is True if the emulator is enabled; otherwise, it is False. APIEnabled is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    SessObj as Object

Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")

' This code segment checks to see if A is API enabled.
' The results are sent to a text box called Result.
If SessObj.APIEnabled = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

Ready

This indicates whether the emulator window is started, API-enabled, and connected. This property checks for all three properties. The value is True if the emulator is ready; otherwise, it is False. Ready is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    SessObj as Object

Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")

' This code segment checks to see if A is ready.
' The results are sent to a text box called Result.
If SessObj.Ready = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

autECLPS object

The autECLPS object allows you to access the methods contained in the PCOMM.autECLPS class. See “autECLPS Class” on page 207 for more information. The following example shows this object.

```
DIM    SessObj as Object
DIM    PSSize as Long
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' For example, get the PS size
PSSize = SessObj.autECLPS.GetSize()
```

autECLOIA object

The autECLOIA object allows you to access the methods contained in the PCOMM.autECLOIA class. See “autECLOIA Class” on page 195 for more information. The following example shows this object.

```
DIM    SessObj as Object
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' For example, set the host window to minimized
If (SessObj.autECLOIA.Katakana) Then
      'whatever
Endif
```

autECLXfer object

The autECLXfer object allows you to access the methods contained in the PCOMM.autECLXfer class. See “autECLXfer Class” on page 243 for more information. The following example shows this object.

```
DIM    SessObj as Object
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' For example
SessObj.Xfer.Sendfile "c:\temp\filename.txt",
                      "filename  text  a0",
                      "CRLF  ASCII"
```

autECLWinMetrics object

The autECLWinMetrics object allows you to access the methods contained in the PCOMM.autECLWinMetrics class. See “autECLWinMetrics Class” on page 230 for more information. The following example shows this object.

```
DIM    SessObj as Object
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' For example, set the host window to minimized
SessObj.autECLWinMetrics.Minimized = True
```

autECLSession Methods

The following section describes the methods that are valid for the autECLSession object.

void SetConnectionByName (String Name)
void SetConnectionByHandle (Long Handle)
void StartCommunication()
void StopCommunication()

SetConnectionByName

This method uses the connection name to set the connection for a newly created autECLSession object. In Personal Communications this connection name is the short ID (character A-Z). There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time.

Prototype

void SetConnectionByName(String Name)

Parameters

String Name One-character string short name of the connection (A-Z).

Return Value

None

Example

The following example shows how to use the connection name to set the connection for a newly created autECLSession object.

```
DIM    SessObj as Object
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
SessObj.SetConnectionByName("A")
' For example, set the host window to minimized
SessObj.autECLWinMetrics.Minimized = True
```

SetConnectionByHandle

This method uses the connection handle to set the connection for a newly created autECLSession object. In Personal Communications this connection handle is a long integer. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time.

Prototype

void SetConnectionByHandle(Long Handle)

Parameters
Long Handle Long integer value of the connection to be set for the object.

Return Value
None

Example
The following example shows how to use the connection handle to set the connection for a newly created autECLSession object.

```
Dim    SessObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
autECLConnList.Refresh
autECLPSObj.SetConnectionByHandle(autECLConnList(1).Handle)
```

StartCommunication
The StartCommunication collection element method connects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Connect.

Prototype
void StartCommunication()

Parameters
None

Return Value
None

Example
The following example shows how to connect a PCOMM emulator session to the host.

```
Dim SessObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
autECLConnList.Refresh
SessObj.SetConnectionByHandle(autECLConnList(1).Handle)

SessObj.StartCommunication()
```

StopCommunication

The StopCommunication collection element method disconnects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Disconnect.

Prototype

void StopCommunication()

Parameters

None

Return Value

None

Example

The following example shows how to connect a PCOMM emulator session to the host.

```
Dim SessObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set SessObj = CreateObject("PCOMM.autECLSession")

' Initialize the session
autECLConnList.Refresh
SessObj.SetConnectionByHandle(autECLConnList(1).Handle)

SessObj.StopCommunication()
```

autECLWinMetrics Class

The autECLWinMetrics object performs operations on an emulator window. It allows you to perform window rectangle and position manipulation (for example, SetWindowRect, Ypos and Width), as well as window state manipulation (for example, Visible or Restored). Its name in the registry is PCOMM.autECLWinMetrics.

You must initially set the connection for the object you create. Use SetConnectionByName or SetConnectionByHandle to initialize your object. The connection may be set only once. After the connection is set, any further calls to the set connection methods cause an exception. If you do not set the connection and try to access a property or method, an exception is also raised.

Note: The autECLSession object in the autECL object is set by the autECL object.

The following example shows how to create and set the autECLWinMetrics object in Visual Basic.

```
DIM    autECLWinObj as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
autECLWinObj.SetConnectionByName("A")
' For example, set the host window to minimized
autECLWinObj.Minimized = True
```

Properties

This section describes the properties for the autECLWinMetrics object.

Type	Name	Attributes
String	WindowTitle	Read/Write
Long	Xpos	Read/Write
Long	Ypos	Read/Write
Long	Width	Read/Write
Long	Height	Read/Write
Boolean	Visible	Read/Write
Boolean	Active	Read/Write
Boolean	Minimized	Read/Write
Boolean	Maximized	Read/Write
Boolean	Restored	Read/Write
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Boolean	Started	Read-only
Boolean	CommStarted	Read-only
Boolean	APIEnabled	Read-only
Boolean	Ready	Read-only

WindowTitle

This is the title that is currently in the title bar for the connection associated with the autECLWinMetrics object. This property may be both changed and retrieved. WindowTitle is a String data type and is read/write enabled. The following example shows this process. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Dim    WinTitle as String
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

WinTitle = autECLWinObj.WindowTitle 'get the window title

' or...

autECLWinObj.WindowTitle = "Flibberdeejibbet" 'set the window title
```

Usage Notes: If WindowTitle is set to blank, the window title of the connection is restored to its original setting.

Xpos

This is the *x* position of the upper left point of the emulator window rectangle. This property may be both changed and retrieved. Xpos is a Long data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Dim    x as Long
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

x = autECLWinObj.Xpos 'get the x position

' or...

autECLWinObj.Xpos = 6081 'set the x position
```

Ypos

This is the *y* position of the upper left point of the emulator window rectangle. This property may be both changed and retrieved. Ypos is a Long data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Dim    y as Long
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

y = autECLWinObj.Ypos 'get the y position

' or...

autECLWinObj.Ypos = 6081 'set the y position
```

Width

This is the width of the emulator window rectangle. This property may be both changed and retrieved. Width is a Long data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Dim    cx as Long
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

cx = autECLWinObj.Width 'get the width

' or...

autECLWinObj.Width = 6081 'set the width
```

Height

This is the height of the emulator window rectangle. This property may be both changed and retrieved. Height is a Long data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Dim    cy as Long
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

cy = autECLWinObj.Height 'get the height

' or...

autECLWinObj.Height = 6081 'set the height
```

Visible

This is the visibility state of the emulator window. This property may be both changed and retrieved. Visible is a Boolean data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

' Set to Visible if not, and vice versa
If ( autECLWinObj.Visible) Then
       autECLWinObj.Visible = False
Else
       autECLWinObj.Visible = True
End If
```

Active

This is the focus state of the emulator window. This property may be both changed and retrieved. Active is a Boolean data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

' Set to Active if not, and vice versa
If ( autECLWinObj.Active) Then
        autECLWinObj.Active = False
Else
        autECLWinObj.Active = True
End If
```

Minimized

This is the minimize state of the emulator window. This property may be both changed and retrieved. Minimized is a Boolean data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

' Set to minimized if not, if minimized set to maximized
If ( autECLWinObj.Minimized) Then
        autECLWinObj.Maximized = True
Else
        autECLWinObj.Minimized = True
End If
```

Maximized

This is the maximize state of the emulator window. This property may be both changed and retrieved. Maximized is a Boolean data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)

' Set to maximized if not, if maximized set to minimized
If ( autECLWinObj.Maximized) Then
       autECLWinObj.Minimized = False
Else
       autECLWinObj.Maximized = True
End If
```

Restored

This is the restore state of the emulator window. Restored is a Boolean data type and is read/write enabled. However, if the connection you are attached to is an inplace, embedded object, this property is read-only. The following example shows this property.

```
Dim    autECLWinObj as Object
Dim    SessList as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set SessList = CreateObject("PCOMM.autECLConnList")

' Initialize the session
SessList.Refresh
autECLWinObj.SetSessionByHandle(SessList(1).Handle)

' Set to restored if not, if restored set to minimized
If ( autECLWinObj.Restored) Then
       autECLWinObj.Minimized = False
Else
       autECLWinObj.Restored = True
End If
```

Name

This property is the connection name string of the connection for which autECLWinMetrics was set. Currently, Personal Communications only returns the short character ID (A-Z) in the string. There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time. Name is a String data type and is read-only. The following example shows this property.

```
DIM    Name as String
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
Obj.SetConnectionByName("A")

' Save the name
Name = Obj.Name
```

Handle

This is the handle of the connection for which the autECLWinMetrics object was set. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time. Handle is a Long data type and is read-only. The following example shows this property.

```
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
Obj.SetConnectionByName("A")

' Save the handle
Hand = Obj.Handle
```

ConnType

This is the connection type for which autECLWinMetrics was set. This type may change over time. ConnType is a String data type and is read-only. The following example shows this property.

```
DIM    Type as String
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
Obj.SetConnectionByName("A")
' Save the type
Type = Obj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation

CodePage

This is the code page of the connection for which autECLWinMetrics was set. This code page may change over time. CodePage is a Long data type and is read-only. The following example shows this property.

```
DIM    CodePage as Long
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
Obj.SetConnectionByName("A")
' Save the code page
CodePage = Obj.CodePage
```

Started

This indicates whether the emulator window is started. The value is True if the window is open; otherwise, it is False. Started is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is started.
' The results are sent to a text box called Result.
If Obj.Started = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

CommStarted

This indicates the status of the connection to the host. The value is True if the host is connected; otherwise, it is False. CommStarted is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if communications are connected
' for A.    The results are sent to a text box called
' CommConn.
If Obj.CommStarted = False Then
      CommConn.Text = "No"
Else
      CommConn.Text = "Yes"
End If
```

APIEnabled

This indicates whether the emulator is API-enabled. A connection may be enabled or disabled depending on the state of its API settings (in a Personal Communications window, choose File ->API Settings). The value is True if the emulator is enabled; otherwise, it is False. APIEnabled is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is API enabled.
' The results are sent to a text box called Result.
If Obj.APIEnabled = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

Ready

This indicates whether the emulator window is started, API enabled, and connected. This property checks for all three properties. The value is True if the emulator is ready; otherwise, it is False. Ready is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is ready.
' The results are sent to a text box called Result.
If Obj.Ready = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

autECLWinMetrics Methods

The following section describes the methods that are valid for the autECLWinMetrics object.

void SetConnectionByName(String Name
void SetConnectionByHandle(Long Handle)
void GetWindowRect(Long Left, Long Top, Long Right, Long Bottom)
void SetWindowRect(Long Left, Long Top, Long Right, Long Bottom)
void StartCommunications()
void StopCommunications()

SetConnectionByName

This method uses the connection name to set the connection for a newly created autECLWinMetrics object. In Personal Communications this connection name is the short ID (character A-Z). There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time.

Note: Do not call this if using the autECLWinMetrics object in autECLSession.

Prototype

void SetConnectionByName(String Name)

Parameters

String Name One-character string short name of the connection (A-Z).

Return Value

None

Example

The following example shows how to use the connection name to set the connection for a newly created autECLWinMetrics object.

```
DIM    autECLWinObj as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the connection
autECLWinObj.SetConnectionByName("A")
' For example, set the host window to minimized
autECLWinObj.Minimized = True
```

SetConnectionByHandle

This method uses the connection handle to set the connection for a newly created autECLWinMetrics object. In Personal Communications this connection handle is a long integer. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time.

Note: Do not call this if using the autECLWinMetrics object in autECLSession.

Prototype
void SetConnectionByHandle(Long Handle)

Parameters
Long Handle Long integer value of the connection to be set for the object.

Return Value
None

Example
The following example shows how to use the connection handle to set the connection for a newly created autECLWinMetrics object.

```
DIM    autECLWinObj as Object
DIM    ConnList as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)
' For example, set the host window to minimized
autECLWinObj.Minimized = True
```

GetWindowRect
The GetWindowRect method returns the bounding points of the emulator window rectangle.

Prototype
void GetWindowRect(Variant Left, Variant Top, Variant Right, Variant Bottom)

Parameters
Variant Left, Top, Right, Bottom Bounding points of the emulator window.

Return Value
None

Example
The following example shows how to return the bounding points of the emulator window rectangle.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Dim   left
Dim   top
Dim   right
Dim   bottom
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)
autECLWinObj.GetWindowRect left, top, right, bottom
```

SetWindowRect

The SetWindowRect method sets the bounding points of the emulator window rectangle.

Prototype

void SetWindowRect(Long Left, Long Top, Long Right, Long Bottom)

Parameters

Long Left, Top, Right, Bottom Bounding points of the emulator window.

Return Value

None

Example

The following example shows how to set the bounding points of the emulator window rectangle.

```
Dim    autECLWinObj as Object
Dim    ConnList as Object
Set autECLWinObj = CreateObject("PCOMM.autECLWinMetrics")
Set ConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection
ConnList.Refresh
autECLWinObj.SetConnectionByHandle(ConnList(1).Handle)
autECLWinObj.SetWindowRect 0, 0, 6081, 6081
```

StartCommunication

The StartCommunication collection element method connects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Connect.

Prototype

void StartCommunication()

Parameters

None

Return Value

None

Example

The following example shows how to connect a PCOMM emulator session to the host.

```
Dim WinObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set WinObj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the session
autECLConnList.Refresh
WinObj.SetConnectionByHandle(autECLConnList(1).Handle)

WinObj.StartCommunication()
```

StopCommunication

The StopCommunication collection element method disconnects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Disconnect.

Prototype

void StopCommunication()

Parameters

None

Return Value

None

Example

The following example shows how to connect a PCOMM emulator session to the host.

```
Dim WinObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set WinObj = CreateObject("PCOMM.autECLWinMetrics")

' Initialize the session
autECLConnList.Refresh
WinObj.SetConnectionByHandle(autECLConnList(1).Handle)

WinObj.StopCommunication()
```

autECLXfer Class

The autECLXfer object provides file transfer services. Its name in the registry is PCOMM.autECLXfer.

You must initially set the connection for the object you create. Use SetConnectionByName or SetConnectionByHandle to initialize your object. The connection may be set only once. After the connection is set, any further calls to the SetConnection methods cause an exception. If you do not set the connection and try to access an autECLXfer property or method, an exception is also raised. The following shows how to create and set the autECLXfer object in Visual Basic.

```
DIM    XferObj as Object

Set XferObj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
XferObj.SetConnectionByName("A")
```

Properties

This section describes the properties for the autECLXfer object.

Type	Name	Attribute
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Boolean	Started	Read-only
Boolean	CommStarted	Read-only
Boolean	APIEnabled	Read-only
Boolean	Ready	Read-only

Name

This property is the connection name string of the connection for which autECLXfer was set. Personal Communications only returns the short character ID (A-Z) in the string. There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time. Name is a String data type and is read-only. The following example shows this property.

```
DIM    Name as String
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
Obj.SetConnectionByName("A")

' Save the name
Name = Obj.Name
```

Handle

This is the handle of the connection for which the autECLXfer object was set. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time. Handle is a Long data type and is read-only. The following example shows this property.

```
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
Obj.SetConnectionByName("A")

' Save the handle
Hand = Obj.Handle
```

ConnType

This is the connection type for which autECLXfer was set. This type may change over time. Conntype is a String data type and is read-only. The following example shows this property.

```
DIM    Type as String
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
Obj.SetConnectionByName("A")
' Save the type
Type = Obj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation

CodePage

This is the code page of the connection for which autECLXfer was set. This code page may change over time. CodePage is a Long data type and is read-only. The following example shows this property.

```
DIM    CodePage as Long
DIM    Obj as Object
Set Obj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
Obj.SetConnectionByName("A")
' Save the code page
CodePage = Obj.CodePage
```

Started

This indicates whether the emulator window is started. The value is True if the window is open; otherwise, it is False. Started is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is started.
' The results are sent to a text box called Result.
If Obj.Started = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

CommStarted

This indicates the status of the connection to the host. The value is True if the host is connected; otherwise, it is False. CommStarted is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if communications are connected
' for A.    The results are sent to a text box called
' CommConn.
If Obj.CommStarted = False Then
      CommConn.Text = "No"
Else
      CommConn.Text = "Yes"
End If
```

APIEnabled

This indicates whether the emulator is API-enabled. A connection may be enabled or disabled depending on the state of its API settings (in a Personal Communications window, choose File -> API Settings). The value is True if the emulator is enabled; otherwise, it is False. APIEnabled is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is API enabled.
' The results are sent to a text box called Result.
If Obj.APIEnabled = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

Ready

This indicates whether the emulator window is started, API enabled, and connected. This property checks for all three properties. The value is True if the emulator is ready; otherwise, it is False. Ready is a Boolean data type and is read-only. The following example shows this property.

```
DIM    Hand as Long
DIM    Obj as Object

Set Obj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
Obj.SetConnectionByName("A")

' This code segment checks to see if A is ready.
' The results are sent to a text box called Result.
If Obj.Ready = False Then
   Result.Text = "No"
Else
   Result.Text = "Yes"
End If
```

autECLXfer Methods

The following section describes the methods that are valid for the autECLXfer object.

void SetConnectionByName(String Name0
void SetConnectionByHandle(Long Handle)
void SendFile(String PCFile, String HostFile, String Options)
void ReceiveFile(String PCFile, String HostFile, String Options)
void StartCommunications()
void StopCommunications()

SetConnectionByName

The SetConnectionByName method uses the connection name to set the connection for a newly created autECLXfer object. In Personal Communications this connection name is the short ID (character A-Z). There can be only one Personal Communications connection open with a given name. For example, there can be only one connection “A” open at a time.

Note: Do not call this if using the autECLXfer object in autECLSession.

Prototype

void SetConnectionByName(String Name)

Parameters

String Name One-character string short name of the connection (A-Z).

Return Value

None

Example

The following example shows how to use the connection name to set the connection for a newly created autECLXfer object.

```
DIM    XferObj as Object

Set XferObj = CreateObject("PCOMM.autECLXfer")

' Initialize the connection
XferObj.SetConnectionByName("A")
```

SetConnectionByHandle

The SetConnectionByHandle method uses the connection handle to set the connection for a newly created autECLXfer object. In Personal Communications this connection handle is a Long integer. There can be only one Personal Communications connection open with a given handle. For example, there can be only one connection “A” open at a time.

Note: Do not call this if using the autECLXfer object in autECLSession.

+

Prototype
void SetConnectionByHandle(Long Handle)

Parameters
Long Handle Long integer value of the connection to be set for the object.

Return Value
None

Example
The following example shows how to use the connection handle to set the connection for a newly created autECLXfer object.

```
DIM    XferObj as Object
DIM    autECLConnList as Object

Set XferObj = CreateObject("PCOMM.autECLXfer")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection with the first connection in the list
autECLConnList.Refresh
XferObj.SetConnectionByHandle(autECLConnList(1).Handle)
```

SendFile

The SendFile method sends a file from the workstation to the host for the connection associated with the autECLXfer object.

Prototype
void SendFile(String PCFile, String HostFile, String Options)

Parameters
String PCFile Name of the file on the workstation.

String HostFile Name of the file on the host.

String Options Host-dependent transfer options. See 248 for more information.

Return Value
None

Usage Notes
File transfer options are host-dependent. The following is a list of some of the valid host options for a VM/CMS host.

ASCII
CRLF
APPEND
LRECL
RECFM
CLEAR/NOCLEAR
PROGRESS
QUIET

Refer to the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for the list of supported hosts and associated file transfer options.

Example

The following example shows how to send a file from the workstation to the host for the connection associated with the autECLXfer object.

```
DIM   XferObj as Object
DIM   autECLConnList as Object
DIM   NumRows as Long

Set XferObj = CreateObject("PCOMM.autECLXfer")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection with the first connection in the autECLConnList
autECLConnList.Refresh
XferObj.SetConnectionByHandle(autECLConnList(1).Handle)

' For example, send the file to VM
XferObj.SendFile "c:\windows\temp\thefile.txt",
                                 "THEFILE TEXT A0",
                                "CRLF  ASCII"
```

ReceiveFile

The ReceiveFile method receives a file from the host to the workstation for the connection associated with the autECLXfer object.

Prototype

void ReceiveFile(String PCFile, String HostFile, String Options)

Parameters

String PCFile Name of the file on the workstation.

String HostFile Name of the file on the host.

String Options Host-dependent transfer options. See 249 for more information.

Return Value

None

Usage Notes

File transfer options are host-dependent. The following is a list of some of the valid host options for a VM/CMS host:

ASCII
CRLF
APPEND
LRECL
RECFM
CLEAR/NOCLEAR
PROGRESS
QUIET

Refer to the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for the list of supported hosts and associated file transfer options.

Example

The following example shows how to receive a file from the host and send it to the workstation for the connection associated with the autECLXfer object.

```
DIM    XferObj as Object
DIM    autECLConnList as Object
DIM    NumRows as Long

Set XferObj = CreateObject("PCOMM.autECLXfer")
Set autECLConnList = CreateObject("PCOMM.autECLConnList")

' Initialize the connection with the first connection in the list
autECLConnList.Refresh
XferObj.SetConnectionByHandle(autECLConnList(1).Handle)
' For example, send the file to VM
XferObj.ReceiveFile "c:\windows\temp\thefile.txt",
                                    "THEFILE TEXT A0",
                                   "CRLF  ASCII"
```

StartCommunication

The StartCommunication collection element method connects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Connect.

Prototype

void StartCommunication()

Parameters

None

Return Value

None

Example

The following example shows how to connect a PCOMM emulator session to the host.

```
Dim XObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set XObj = CreateObject("PCOMM.autECLXfer")

' Initialize the session
autECLConnList.Refresh
XObj.SetConnectionByHandle(autECLConnList(1).Handle)

XObj.StartCommunication()
```

StopCommunication

The StopCommunication collection element method disconnects the PCOMM emulator to the host data stream. This has the same effect as going to the PCOMM emulator Communication menu and choosing Disconnect.

Prototype

void StartCommunication()

Parameters

None

Return Value

None

Example

The following example shows how to connect a PCOMM emulator session to the host.

```
Dim XObj as Object
Dim autECLConnList as Object

Set autECLConnList = CreateObject("PCOMM.autECLConnList")
Set XObj = CreateObject("PCOMM.autECLXfer")

' Initialize the session
autECLConnList.Refresh
XObj.SetConnectionByHandle(autECLConnList(1).Handle)

SessObj.StopCommunication()
```

autECLXfer

Chapter 4. Host Access Class Library LotusScript Extension

The Host Access Class Library LotusScript Extension (ECLLSX) allows you to write LotusScript programs that can query and control Personal Communications connections. The ECLLSX contains several new LotusScript classes that can be used inside LotusScript programs. By running methods on objects created from the new classes, you can access Personal Communications connection information and control the objects that make up a Personal Communications connection.

For example, if you want to automate the task of entering a line of text in a Personal Communications connection you can write a LotusScript program that uses the lsxECLPS class to create an lsxECLPS object associated with the presentation space of a Personal Communications connection. You can then run the SendKeys method on this lsxECLPS object to send a series of keystrokes to the presentation space and the effect is similar to a user typing the keystrokes in that presentation space. The following code fragment shows how this would be done using the ECLLSX classes.

```
'Create an lsxECLPS object associated with Personal
'Communications connection A
dim myPSObj as new lsxECLPS("A")

'Send some keystrokes to the presentation space of
'connection A
myPSObj.Sendkeys("[clear]QUERY FILES[ENTER]")
```

The ECLLSX classes are similar to the ECL C++ classes. Each ECLLSX class begins with lsxECL, for LotusScript Host Access Class Library. The classes are as follows:

- lsxECLConnection, Connection Information, on page 255 provides information about the Personal Communications connection associated with this lsxECLConnection object. In addition to being included in an lsxECLConnList object, an lsxECLConnection object can be created on its own if you only want to query information on a specific Personal Communications connection.
- lsxECLConnList, Connection List, on page 261 provides a list of Personal Communications connections on a system. Each element in an lsxECLConnList is an lsxECLConnection object.
- lsxECLConnMgr, Connection Manager, on page 265 manages Personal Communications connections on a system. Each lsxECLConnMgr object contains an lsxECLConnList object.
- lsxECLField, Field Information, on page 268 provides information on a field in the presentation space of the Personal Communications connection associated with this lsxECLField object.
- lsxECLFieldList, Field List, on page 275 provides a list of the fields in the presentation space of the Personal Communications connection associated with this lsxECLFieldList object. Each element in the list is an lsxECLField object.
- lsxECLOIA, Operator Information Area, on page 279 provides methods to query and manipulate the Operator Information Area of the associated Personal Communications connection. In addition to being contained in an lsxECLSession object, an lsxECLOIA object can be created on its own if you only want to perform OIA related tasks.

© Copyright IBM Corp. 1997

lsxECLPS, Presentation Space, on page 288 provides methods to query and manipulate the Presentation Space of the associated Personal Communications connection. An lsxECLPS object contains an lsxECLFieldList object. In addition to being contained in an lsxECLSession object, an lsxECLPS object can be created on its own if you only want to perform presentation space related tasks.

lsxECLSession, Session, on page 298 provides Personal Communications connection related functionality and information. For convenience, an lsxECLSession object contains lsxECLPS, lsxECLXfer, lsxECLWinMetrics and lsxECLOIA objects for the Personal Communications connection associated with the lsxECLSession object.

lsxECLWinMetrics, Window Metrics, on page 304 provides methods to query the window metrics of the Personal Communications connection associated with this lsxECLWinMetrics object. In addition to being contained in an lsxECLSession object, an lsxECLWinMetrics object can be created on its own if you only want to perform window metrics related queries.

lsxECLXfer, File Transfer, on page 313 provides methods to transfer files between the host and the workstation over the Personal Communications connection associated with this file transfer object. In addition to being contained in an lsxECLSession object, an lsxECLXfer object can be created on its own if you only want to perform file transfer related tasks

In order to use the ECL LotusScript Extension classes in a LotusScript program, you must load the ECL LotusScript Extension. This can be done using the following LotusScript statement:

```
USELSX "*pcslsx"
```

This statement loads the ECL LotusScript Extension and allows you to access the ECL LotusScript Extension classes.

This chapter describes each class' methods and properties in detail.

lsxECLConnection Class

The lsxECLConnection class provides information about a Personal Communications connection.

An lsxECLConnection object is associated with a Personal Communications connection when the lsxECLConnection object is created. You cannot change the connection associated with an lsxECLConnection object. If you want to query information about a different connection, you must create a new lsxECLConnection object associated with that connection.

There are two ways to create an lsxECLConnection object:

1. Create a new lsxECLConnection object by passing a Personal Communications connection name as a parameter on the new statement. A Personal Communications connection name is a single, alphabetic character from A-Z. The following is an example of creating an lsxECLConnection object that is associated with Personal Communications connection A:

   ```
   ' Create an lsxECLConnection object associated with PCOMM connection A
       dim myConnObj as new lsxECLConnection("A")
   ```

2. Create a new lsxECLConnection object by passing a Personal Communications connection handle as a parameter on the new statement. A Personal Communications connection handle is a Long integer. The following is another example of creating an lsxECLConnection object that is associated with Personal Communications connection A:

   ```
   ' Create an IsxECLConnection object using a connection handle
    dim myPSObj as new IsxECPS("A")

   ' Now use the connection handle from the PS object to build a connection object
   dim myConnObj as new IsxECLConnection(myPSObj.Handle)
   ```

Properties

This section describes the properties for the lsxECLConnection class.

Type	Name	Attribute
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Integer	Started	Read-only
Integer	CommStarted	Read-only
Integer	APIEnabled	Read-only
Integer	Ready	Read-only

Name

Name is the connection name of the Personal Communications connection associated with this lsxECLConnection object. The Name property is a String data type and is read-only. Personal Communications connection names are one character in length and from the character set A-Z. The following example shows this property.

```
' Create an lsxECLConnMgr object to get the list of
' connections on the system.
dim myCMgrObj as new lsxECLConnMgr
dim myName as String

' Get the connection name for the first connection
' in the connection list.
myName = myCMgrObj.ConnList(1).Name
```

Handle

Handle is the connection handle of the Personal Communications connection associated with this lsxECLConnection object. The Handle property is a Long data type and is read-only. The following example shows this property.

```
' Create a new lsxECLConnection object associated with connection A
dim myConnObj as new lsxECLConnection("A")
dim myHandle as Long

' Get the connection handle for connection A
myHandle = myConnObj.Handle
```

ConnType

ConnType is the connection type of the connection that is associated with this lsxECLConnection object. The ConnType property is a String data type and is read-only. The following example shows this property.

```
' Create a new lsxECLConnection object associated with connection A
dim myConnObj as new lsxECLConnection("A")
dim myConnType as String

' Get the Connection type for connection A
myConnType = myConnObj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation
UNKNOWN	Unknown

CodePage

CodePage is the code page of the connection associated with this lsxECLConnection object. The CodePage property is a Long data type, is read-only and cannot be changed through this LotusScript interface. However, the code page of a connection may change if the Personal Communications connection is restarted with a new configuration (see “lsxECLConnMgr Class” on page 265 for information about starting a connection). The following example shows this property.

```
' Create a new lsxECLConnection object associated with connection A
dim myConnObj as new lsxECLConnection("A")
dim myCodePage as Long

' Get the CodePage for connection A
myCodePage = myConnObj.CodePage
```

Started

Started is a Boolean flag that indicates whether the connection associated with this lsxECLConnection object is started. The Started property is an integer and is read-only. Started is 1 if the Personal Communications connection has been started; otherwise, it is 0. The following example shows this property.

```
' Create a new lsxECLConnection object associated with connection A
dim myConnObj as new lsxECLConnection("A")

' See if connection is started
if myConnObj.Started then
  call  connection_started
```

CommStarted

CommStarted is a Boolean flag that indicates whether the connection associated with this lsxECLConnection object is connected to the host data stream. The CommStarted property is an integer and is read-only. CommStarted is 1 if there is communication with the host; otherwise, it is 0. The following example shows this property.

```
' Create a new lsxECLConnection object associated with connection A
dim myConnObj as new lsxECLConnection("A")

' See if we are communicating with the host
if myConnObj.CommStarted then
  call  connection_connected
```

APIEnabled

APIEnabled is a Boolean flag that indicates whether the HLLAPI API has been enabled for the Personal Communications connection associated with this lsxECLConnection object. The APIEnabled property is an integer and is read-only. APIEnabled is 1 if the HLLAPI API is available; otherwise, it is 0. The following example shows this property.

```
' Create a new lsxECLConnection object associated with connection A
dim myConnObj as new lsxECLConnection("A")

' See if the HLLAPI API is enabled on this connection
if myConnObj.APIEnabled then
  call  hllapi_available
```

Ready

Ready is a Boolean flag that indicates whether the Personal Communications connection associated with this lsxECLConnection object is ready. The Ready property is a combination of the Started, CommStarted and APIEnabled properties. It is an integer and is read-only. Ready is 1 if the Started, CommStarted and APIEnabled properties are 1; otherwise, it is 0. The following example shows this property.

```
' Create a new lsxECLConnection object associated with connection A
dim myConnObj as new lsxECLConnection("A")

' See if the connection is ready
if myConnObj.Ready then
  call  conn_ready
```

lsxECLConnection Methods

The following section describes the methods that are valid for the lsxECLConnection class.

StartCommunication()
StopCommunication()

StartCommunication

This method connects the ECL Connection to the host data stream. The effect is the same as using the Connect option on the Personal Communications emulator Communication menu.

Prototype
StartCommunication()

Parameters
None

Return Value
None

Example
The following example shows how to connect the ECL Connection to the host data stream.

```
' Create a new lsxECLConnection object for ECL Connection A
dim myConnObj as new lsxECLConnection("A")

' Make sure we have communications with the host
if myConnObj.CommStarted = 0 then
 myConnObj.StartCommunication
```

StopCommunication

This method disconnects the ECL Connection from the host data stream. The effect is the same as using the Disconnect option on the Personal Communications emulator Communication menu.

Prototype
StopCommunication()

Parameters
None

Return Value
None

Example

The following example shows how to disconnect the ECL Connection from the host data stream.

```
' Create a new lsxECLConnection object for ECL Connection A
dim myConnObj as new lsxECLConnection("A")

' Stop communications with the host on this connection
if myConnObj.CommStarted = 1 then
  myConnObj.StopCommunication
```

lsxECLConnList Class

The lsxECLConnList class manages the Personal Communications connections on a system. An lsxECLConnList object contains a list of all the connections that are currently available on the system. Each element of the connection list is an lsxECLConnection object. lsxECLConnection objects can be queried to determine the state of the associated connection. See "lsxECLConnection Class" on page 255 for details on its methods and properties.

An lsxECLConnList object provides a snapshot of the current connections on a system. The Refresh method provides a way to take a new snapshot of the connections on a system. The order of the connections in the lsxECLConnList is undefined and could change as a result of calling the Refresh method.

There are two ways to create an lsxECLConnList object:

1. Create a new lsxECLConnList object by using the new statement. There are no parameters used when creating the lsxECLConnList object. The following is an example of creating an lsxECLConnList object:

   ```
   ' Create an lsxECLConnList object
   dim myCListObj as new lsxECLConnList
   ```

2. Create an lsxECLConnMgr object and an lsxECLConnList object is automatically created. Access the lsxECLConnList attribute of the lsxECLConnMgr object to get to the lsxECLConnList object contained in the lsxECLConnMgr object. The following is an example of accessing the lsxECLConnList object contained in an lsxECLConnMgr object:

   ```
   dim myCMgrObj as new lsxECLConnMgr
   dim myCListObj as lsxECLConnList

   ' Get the lsxECLConnList object from inside the lsxECLConnMgr
   set myCListObj = myCMgrObj.lsxECLConnList
   ```

Properties

This section describes the properties of the lsxECLConnList class.

Type	Name	Attributes
Long	Count	Number of connections in the connection list

Count

Count is the number of connections present in the lsxECLConnList. The Count property is a Long data type and is read-only. The following example shows this property.

```
dim myCMgrObj as new lsxECLConnMgr
dim myCListObj as lsxECLConnList
Set myCListObj = myCMgrObj.lsxECLConnList

dim numConns as Long

' Get a current snapshot of connections on the system
myCListObj.Refresh

' Get number of connections
numConns = myCListObj.Count
```

lsxECLConnList Methods

The following section describes the methods that are valid for the lsxECLConnList class.

Refresh()
FindConnectionByHandle(Long Handle)
FindConnectionByName(String Name)

Refresh

This method gets a list of the connections available on a system.

Prototype
Refresh()

Parameters
None

Return Value
None

Example
The following example shows how to use the Refresh method to get a current list of connections.

```
'Create a new lsxConnMgr
dim myCMgrObj as new lsxECLConnMgr

'Get the lsxConnList contained in the lsxConnMgr
dim myCListObj as lsxECLConnList
set myCListObj = myCMgrObj.lsxECLConnList

later...

'Refresh the list of connections found in lsxECLConnList
myCListObj.Refresh
```

FindConnectionByHandle

This method finds the connection identified by the **Handle** parameter in the lsxECLConnList list of connections.

Prototype
FindConnectionByHandle(Long Handle)

Parameters
Long Handle The connection handle of the target connection.

Return Value
lsxECLConnection The lsxECLConnection object corresponding to the target connection.

Example

The following example shows how to find the connection identified by the **Handle** parameter.

```
dim myConnObj as lsxECLConnection

'Create a new lsxECLConnList object
dim myCListObj as new lsxECLConnList

'Create a new lsxECLPS associated with connection A
dim myPSObj as new lsxECLPS("A")

'Get the lsxECLConnection object for connection A
set myConnObj = myCListObj.FindConnectionByHandle(myPSObj.Handle)
```

FindConnectionByName

This method finds a connection identified by the **Name** parameter in the lsxECLConnList list of connections.

Prototype

FindConnectionByName(String Name)

Parameters

String Name The connection name of the target connection.

Return Value

Long Handle The connection handle of the target connection.

Example

The following example shows how to fine a connection identified by the **Name** parameter.

```
dim myConnObj as lsxECLConnection

'Create a new lsxECLConnList object
dim myCListObj as new lsxECLConnList

'Get the lsxECLConnection object for connection A
set myConnObj = myCListObj.FindConnectionByName("A")
```

lsxECLConnMgr Class

The lsxECLConnMgr class manages Personal Communications connections on a system. It contains methods relating to the management of connections such as starting, stopping and querying connections. It also contains an lsxECLConnList object that is a static list of the connections available when the list was created (see “lsxECLConnList Class” on page 261 for more details on the lsxECLConnList class).

To create an lsxECLConnMgr object, use the new statement. There are no parameters used when creating the lsxECLConnMgr object. The following is an example of creating an lsxECLConnMgr object:

```
'Create an lsxECLConnMgr object
dim myCMgrObj as new lsxECLConnMgr
```

Properties

This section describes the properties of the lsxECLConnMgr class.

Type	Name	Attributes
lsxECLConnList	lsxECLConnList	Read-only

lsxECLConnList

The lsxECLConnMgr object contains an lsxECLConnList object. See “lsxECLConnList Class” on page 261 for details on the lsxECLConnList methods and properties. The following example shows this object.

```
' Create a new Connection manager
dim myCMgrObj as new lsxECLConnMgr

dim NumConns as Long

' Get the number of connections currently available on the system
NumConns = myCMgrObj.lsxECLConnList.Count
```

lsxECLConnMgr Methods

The following section explains the methods that are valid for the lsxECLConnMgr class.

StartConnection(String ConfigParms)
StopConnection(Long Handle, String StopParms)

StartConnection

This method starts a new Personal Communications emulator connection. The **ConfigParms** parameter contains Personal Communications connection startup information (see 266 for an explanation of the startup information).

Prototype

StartConnection(String ConfigParms)

Parameters

String ConfigParms Personal Communications connection startup information.

Return Value

None

Example

The following example shows how to start a new Personal Communications emulator connection.

```
' Create a connection manager
dim myCMgrObj as new lsxECLConnMgr

' Start a new PCOMM connection
myCMgrObj.StartConnection("profile=coax Name=e")
```

Usage Notes

The connection configuration string is implementation-specific. Different implementations of the lsxECLConnMgr class may require different formats or information in the configuration string. The new connection is started upon return from this call, but it may or may not be connected to the host.

For Personal Communications, the configuration string has the following format:

PROFILE=[']<filename>['] [NAME=<c>] [WINSTATE=<MAX|MIN|RESTORE|HIDE>]

Optional parameters are enclosed in square brackets []. The parameters are separated by at least one blank. Parameters may be in upper, lower, or mixed case and may appear in any order. The meaning of each parameter is as follows:

PROFILE=<filename>: Names the Personal Communications workstation profile (.WS file), which contains the configuration information. This parameter is not optional; a profile name must be supplied. If the file name contains blanks the name must be enclosed in single quotation marks. The <filename> value may be either the profile name with no extension, the profile name with the .WS extension, or the fully qualified profile name path.

NAME=<c> specifies the short ID of the new connection. This value must be a single, alphabetic character (A-Z). If this value is not specified, the next

available connection ID is assigned automatically. If a connection already exists with the specified ID a connection not Open error is thrown.

WINSTATE=<MAX|MIN|RESTORE|HIDE> specifies the initial state of the emulator window. The default if this parameter is not specified is RESTORE.

StopConnection

This method stops the Personal Communications connection identified by the **Handle** parameter. The **StopParms** parameters are additional Personal Communications stop connection parameters. See 267 for an explanation of the valid values of StopParms.

Prototype

StopConnection(Long Handle, String StopParms)

Parameters

Long Handle Connection handle of the connection to be stopped.

String StopParms Personal Communications connection stop parameters.

Return Value

None

Example

The following example shows how to stop the Personal Communications connection identified by the **Handle** parameter.

```
' Create a new connection manager
dim myCMgrObj as new lsxECLConnMgr

' Stop the first connection found in the list
myCMgrObj.StopSession(myCMgrObj.lsxECLConnList(1).Handle,
                      "saveprofile=no")
```

Usage Notes

The connection stop parameter string is implementation-specific. Different implementations of the lsxECLConnMgr class may require a different format and contents of the parameter string. For Personal Communications the string has the following format:

[SAVEPROFILE=<YES|NO|DEFAULT>]

Optional parameters are enclosed in square brackets []. The parameters are separated by at least one blank. Parameters may be in upper, lower, or mixed case and may appear in any order. The meaning of each parameter is as follows:

SAVEPROFILE=<YES|NO|DEFAULT> controls the saving of the current connection configuration back to the workstation profile (.WS file). This causes the profile to be updated with any configuration changes you may have made during the connection. If NO is specified, the connection is stopped and the profile is not updated. If YES is specified, the connection is stopped and the profile is updated with the current (possibly changed) configuration. If DEFAULT is specified, the update option is controlled by the File->Save On Exit emulator menu option. If this parameter is not specified, DEFAULT is used.

lsxECLField Class

lsxECLField contains information for a given field from an lsxECLFieldList object residing in an lsxECLPS object. The only way to obtain an lsxECLField object is to access it through the lsxECLFieldList object.

Properties

This section describes the properties for the lsxECLField class.

Type	Name	Attributes
Long	StartRow	Read-only
Long	StartCol	Read-only
Long	EndRow	Read-only
Long	EndCol	Read-only
Long	Length	Read-only
Integer	Modified	Read-only
Integer	Protected	Read-only
Integer	Numeric	Read-only
Integer	HighIntensity	Read-only
Integer	PenDetectable	Read-only
Integer	Display	Read-only

StartRow

StartRow is the row of the first character of the field. The StartRow property is a Long data type and is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim StartRow as Long

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Get the starting row of the first field in the list
   StartRow = myPSObj.lsxECLFieldList(1).StartRow
Endif
```

StartCol

StartCol is the column of the first character of the field. The StartCol property is a Long data type and is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim StartCol as Long

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Get the starting column of the first field in the list
   StartCol = myPSObj.lsxECLFieldList(1).StartCol
Endif
```

EndRow

EndRow is the row of the last character of the field. The EndRow property is a Long data type and is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim EndRow as Long

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Get the ending row of the first field in the list
   EndRow = myPSObj.lsxECLFieldList(1).EndRow
Endif
```

EndCol

EndCol is the column of the last character of the field. The EndCol property is a Long data type and is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim EndCol as Long

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Get the ending column of the first field in the list
   EndCol = myPSObj.lsxECLFieldList(1).EndCol
Endif
```

Length

Length is the length of the field. The Length property is a Long data type and is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim length as Long

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Get the length of the first field in the list
   length = myPSObj.lsxECLFieldList(1).Length
Endif
```

Modified

Modified is a Boolean flag that indicates whether this field has been modified. A value of 1 means the field has been modified; otherwise, the value is 0. This property is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

 ' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
if (myPSObj.lsxECLFieldList.Count) then
' Check if the first field in the list has been modified
   if (myPSObj.lsxECLFieldList(1).Modified) then
      call  field_modified
  endif
endif
```

Protected

This is a Boolean flag that indicates whether the field has a protected attribute. A value of 1 means the field has the protected attribute; otherwise, the value is 0. This property is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
if (myPSObj.lsxECLFieldList.Count) then
' Check if the first field in the list is protected
   if (myPSObj.lsxECLFieldList(1).Protected) then
      call  field_protected
  endif
endif
```

Numeric

This is a Boolean flag that indicates whether the field has the numeric-only input attribute. A value of 1 means the field has the numeric-only attribute; otherwise, the value is 0. This property is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
if (myPSObj.lsxECLFieldList.Count) then
' Check if the first field has the numeric only attribute
   if (myPSObj.lsxECLFieldList(1).Numeric) then
      call  numeric_field
  endif
endif
```

HighIntensity

This is a Boolean flag that indicates whether the field has the high intensity attribute. A value of 1 means the field has the high intensity attribute; otherwise, the value is 0. This property is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Check if the first field has the high intensity attribute
   if (myPSObj.lsxECLFieldList(1).HighIntensity) then
      call  high_intensity_field
  endif
Endif
```

PenDetectable

This is a Boolean flag that indicates whether this field has the pen detectable attribute. A value of 1 means the field does have the pen detectable attribute; otherwise, the value is 0. This property is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Check if the first field is pen detectable
   if (myPSObj.lsxECLFieldList(1).PenDetectable) then
      call  field_pen_detectable
  endif
Endif
```

Display

This is a Boolean flag that indicates whether this field has the display attribute. A value of 1 means that the field has the display attribute; otherwise, the value is 0. This property is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Check if the first field has the display attribute
    if (myPSObj.lsxECLFieldList(1).Display) then
       call  display_field
   endif
Endif
```

lsxECLField Methods

The following section describes the methods that are valid for the lsxECLField class.

GetText()
SetText(String Text)

GetText

This method retrieves the characters of the field from the text plane.

Prototype
GetText()

Parameters
None

Return Value
String A string of characters from the text plane.

Example
The following example shows how to retrieve the characters of the field:

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim fieldData as String

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Get the characters from the first field's text plane
   fieldData = myPSObj.lsxECLFieldList(1).GetText()
Endif
```

SetText

This method sends a string of characters to the field. The **Text** parameter is a String data type. If the text exceeds the length of the field, the text is truncated.

Prototype
SetText(String Text)

Parameters
String Text String of characters to send to the field.

Return Value
None

Example

The following example shows how to send a string of characters to the field.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

 ' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh
If (myPSObj.lsxECLFieldList.Count) Then
' Send a string of characters to the first field
   myPSObj.lsxECLFieldList(1).SetText("This is a test")
Endif
```

lsxECLFieldList Class

The lsxECLFieldList class performs operations on fields in a connection's presentation space. An lsxECLFieldList object is contained in an lsxECLPS object and can only be accessed through an existing lsxECLPS object. See "lsxECLPS Class" on page 288 for more information on lsxECLPS objects.

An lsxECLFieldList object provides a static snapshot of what the presentation space contained when the Refresh method was called. The lsxECLFieldList class contains a list of all the fields in a given presentation space. Each element of the collection is an lsxECLField object. See "lsxECLField Class" on page 268 for more information about the lsxECLField objects.

Properties

This section describes the properties of the lsxECLFieldList class.

Type	Name	Attributes
Long	Count	Read-only

Count

Count is the number of fields in the lsxECLFieldList list. This value could change after each call to the Refresh method. The Count property is a Long data type and is read-only. The following example shows this property.

```
' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim numFields as Long

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh

' Get the field that contains row 2, column 1
numFields = myPSObj.lsxECLFieldList.Count
lsxECLFieldList
```

lsxECLFieldList Methods

The following section describes the methods that are valid for the lsxECLFieldList class.

Refresh()
FindFieldByRowCol(Long row, Long col)
FindFieldByText(String Text, [optional] Long dir, [optional] Long row, [optional] Long col)

Refresh

This method refreshes the list of lsxECLField objects contained in the lsxECLFieldList object.

Prototype
Refresh()

Parameters
None

Return Value
None

Example
The following example shows how to refresh the list of lsxECLField objects contained in the lsxECLFieldList object.

```
' Create a new connection manager
dim myCMgr as new lsxECLConnMgr

dim myPSObj as lsxECLPS
set myPSObj = myCMgr.lsxECLConnList(1).Handle

dim numFields as Long

' Build the field list and get the number of fields
myPSObj.lsxECLFieldList.Refresh
numFields = myPSObj.lsxECLFieldList.Count
```

FindFieldByRowCol

This method finds an lsxECLField object in the lsxECLFieldList that contains the position indicated by the **row** and **col** parameters, which is a position in the presentation space. See “lsxECLField Class” on page 268 for the methods and properties of the lsxECLField object.

Prototype
FindFieldByRowCol(Long row, Long col)

Parameters
Long row Row position in the presentation space.

Long col Column position in the presentation space.

Return Value
ECLField ECLField object.

Example
The following example shows how to find an lsxECLField object in the lsxECLFieldList that contains the position indicated by the **row** and **col** parameters.

```
dim myFInfoObj as lsxECLField

' Create a new PS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh

' Get the field that contains row 2, column 1
myFInfoObj = myPSObj.lsxECLFieldList.FindFieldByRowCol(2,1)
```

FindFieldByText

This method finds the lsxECLField object in the lsxECLFieldList that contains the location of the string provided in the Text parameter. The search starts at the location indicated by the row and col parameters. If the row and col parameters are not specified, the search starts at the beginning the presentation space. The **row** and **col** parameters must both be specified or omitted. The optional **dir** parameter indicates the direction to search.

Prototype
FindFieldByText(String Text, [optional] dir, [optional Long row, [optional] Long col)

Parameters
String Text Target string to search for in the presentation space.

Long dir Direction in which to search. Valid values are **1** for Search Forward and **2** for Search Backward. The default is 1, Search Forward.

Long row Target row in the presentation space. This parameter is optional. If not specified, the search starts at the beginning of the presentation space. If row is specified, col must also be specified.

Long col Target column in the presentation space. This parameter is optional. If it is not specified, the search starts at the beginning of the presentation space. If col is specified, row must also be specified.

Return Value
lsxECLField An lsxECLField object.

Example
The following example shows how to search for the lsxECLField object that contains a specified string.

```
' Create an lsxECLPS object associated with ECL Connection A
dim myPSObj as new lsxECLPS("A")

dim myFieldObj as lsxECLField

' Refresh the list of fields
myPSObj.lsxECLFieldList.Refresh

' Search for the field containing the specified string.
' The search direction defaults to forward and the search
' will start from the beginning of the presentation space.
set myFieldObj = myPSObj.lsxECLFieldList.FindFieldByText("Target Text")
```

lsxECLOIA Class

The lsxECLOIA class provides status information from a connection's operator information area.

The lsxECLOIA object is associated with a Personal Communications connection when the lsxECLOIA object is created. You cannot change the connection that is associated with an lsxECLOIA object. If you want to query the OIA of a different connection, you must create a new lsxECLOIA object associated with that connection.

There are three ways to create an lsxECLOIA object:

1. Create a new lsxECLOIA object by passing a Personal Communications connection name as a parameter on the new statement. A Personal Communications connection name is a single, alphabetic character from A-Z. The following is an example of creating an lsxECLOIA object that is associated with Personal Communications connection A:

   ```
   ' Create an lsxECLOIA object associated with PCOMM connection A
   dim myOIAObj as new lsxECLOIA("A")
   ```

2. Create a new lsxECLOIA object by passing a Personal Communications connection handle as a parameter on the new statement. A Personal Communications connection handle is a Long integer and is usually obtained by querying the lsxECLConnection object corresponding to the target Personal Communications connection (see "lsxECLConnMgr Class" on page 265, "lsxECLConnList Class" on page 261 and "lsxECLConnection Class" on page 255 for more information on the properties and methods of those objects). The following is an example of creating an lsxECLOIA object using a Personal Communications connection handle:

   ```
   dim myOIAObj as lsxECLOIA
   dim myConnObj as new lsxECLConnection

   ' Create a new lsxECLOIA object using a connection handle
   set myOIAObj = new lsxECLOIA(myConnObj.Handle)
   ```

3. Create an lsxECLSession object to create an lsxECLOIA object. After creating the lsxECLSession object, access its lsxECLOIA attribute to get access to the lsxECLOIA object contained in the lsxECLSession object. The following is an example of accessing the lsxECLOIA object contained in an lsxECLSession object:

   ```
   dim myOIAObj as lsxECLOIA
   ' Create a new lsxECLSession object associated with connection A
   dim mySessObj as new lsxECLSession("A")

   ' Get the lsxECLOIA object from the lsxECLSession object
   set myOIAObj = mySessObj.lsxECLOIA
   ```

Properties

This section describes the properties for the lsxECLOIA class.

Type	Name	Attributes
Integer	Alphanumeric	Read-only
Integer	APL	Read-only
Integer	Katakana	Read-only
Integer	Hiragana	Read-only
Integer	DBCS	Read-only
Integer	UpperShift	Read-only
Integer	Numeric	Read-only
Integer	CapsLock	Read-only
Integer	InsertMode	Read-only
Integer	CommErrorCondition	Read-only
Integer	MessageWaiting	Read-only
Integer	InputInhibited	Read-only
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Integer	Started	Read-only
Integer	CommStarted	Read-only
Integer	APIEnabled	Read-only
Integer	Ready	Read-only

Alphanumeric

This property queries the connection's operator information area to determine if the field at the cursor position is alphanumeric. The Alphanumeric property is set to 1 if the field is alphanumeric; otherwise, it is set to 0. Alphanumeric is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if the field is alphanumeric
if myOIAObj.Alphanumeric then
  call  abc
```

APL

This property queries the connection's operator information area to determine if the keyboard is in APL mode. The APL property is set to 1 if the keyboard is in APL mode; otherwise, it is set to 0. APL is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if the keyboard is in APL mode
if myOIAObj.APL then
  call  abc
```

Katakana

This property queries the connection's operator information area to determine if Katakana characters are enabled. The Katakana property is set to 1 if Katakana characters are enabled; otherwise, it is set to 0. Katakana is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if Katakana characters are available
if myOIAObj.Katakana then
  call  abc
```

Hiragana

This property queries the connection's operator information area to determine if Hiragana characters are enabled. The Hiragana property is set to 1 if Hiragana characters are enabled; otherwise, it is set to 0. Hiragana is an Integer data type and is read-only. The following example shows this property.

```
 ' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if Hiragana characters are available
if myOIAObj.Hiragana then
  call  abc
```

DBCS

This property queries the connection's operator information area to determine if the field at the cursor position is DBCS. The DBCS property is set to 1 if the field is DBCS; otherwise, it is set to 0. DBCS is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if DBCS is available
if myOIAObj.DBCS then
  call  abc
```

UpperShiftt

This property queries the connection's operator information area to determine if the keyboard is in uppershift mode. The UpperShift property is set to 1 if the keyboard is in uppershift mode; otherwise, it is set to 0. UpperShift is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if the keyboard is in uppershift mode
if myOIAObj.UpperShift then
  call  abc
```

Numeric

This property queries the connection's operator information area to determine if the field at the cursor postion is numeric. The Numeric property is set to 1 if the field is numeric; otherwise, it is set to 0. Numeric is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if the field is numeric
if myOIAObj.Numeric then
  call  abc
```

CapsLock

This property queries the connection's operator information area to determine if the keyboard is in capslock mode. The CapsLock property is set to 1 if the keyboard is in capslock mode, otherwise it is set to 0. CapsLock is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if the keyboard is in capslock mode
if myOIAObj.CapsLock then
  call  abc
```

InsertMode

This property queries the connection's operator information area to determine if the keyboard is in insert mode. The InsertMode property is set to 1 if the keyboard is in insert mode; otherwise, it is set to 0. InsertMode is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' Check if the keyboard is in insert mode
if myOIAObj.InsertMode then
  call  abc
```

CommErrorReminder

This property queries the connection's operator information area to determine if a communications error reminder condition exists. The CommErrorReminder property is set to 1 if a communications error reminder condition exists; otherwise, it is set to 0. CommErrorReminder is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' See if we have a communications error reminder
' condition on connection A
if myOIAObj.CommErrorReminder then
  call  abc
```

MessageWaiting

This property queries the connection's operator information area to determine if the message waiting indicator is on. The MessageWaiting property is set to 1 if the message waiting indicator is on; otherwise, it is set to 0. MessageWaiting is an Integer data type and is read-only. The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
' Assume connection A is a 5250 connection
dim myOIAObj as new lsxECLOIA("A")

' See if we have a message waiting on connection A
if myOIAObj.MessageWaiting then
  call  abc
```

The message waiting indicator is only used in connections of SessionType "DISP5250". For other connection types, the MessageWaiting property is always set to 0.

InputInhibited

This property queries whether the host is ready for input. InputInhibited is an Integer data type and is read-only. The following table shows valid values for InputInhibited.

Value	Meaning
0	Not Inhibited
1	System Wait
2	Communication Check
3	Program Check
4	Machine Check
5	Other Inhibit

The following example shows this property.

```
' Create a new lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' See if the host is ready for input
if myOIAObj.InputInhibited = 0 then
   ' Okay to send text
  call  sendtext
```

Name

Name is the connection name of the Personal Communications connection associated with this lsxECLOIA object. The Name property is a String data type and is read-only. Personal Communications connection names are one character in length and from the character set A-Z. The following example shows this property.

```
' Create an lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

dim myName as String

' Get our connection name
myName = myOIAObj.Name
```

Handle

Handle is the connection handle of the Personal Communications connection associated with this lsxECLOIA object. The Handle property is a Long data type and is read-only. The following example shows this property.

```
' Create an lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

dim myHandle as Long

' Get our connection handle
myHandle = myOIAObj.Handle
```

ConnType

ConnType is the connection type of the connection that is associated with this lsxECLOIA object. The ConnType property is a String data type and is read-only. See Usage Notes for the list of possible connection type values. The following example shows this property.

```
' Create an lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

dim myConnType as String

' Get the connection type for connection A
myConnType = myOIAObj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation
UNKNOWN	Unknown

CodePage

CodePage is the code page of the connection associated with this lsxECLOIA object. The CodePage property is a Long data type, is read-only and cannot be changed through this LotusScript interface. However, the code page of a connection may change if the Personal Communications connection is restarted with a new configuration (see “lsxECLConnMgr Class” on page 265 for information about starting a connection). The following example shows this property.

```
' Create an lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

dim myCodePage as Long

' Get the code page for connection A
myCodePage = myOIAObj.CodePage
```

Started

Started is a Boolean flag that indicates whether the connection associated with this lsxECLOIA object is started (for example, still running as a Personal Communications connection). The Started property is an integer and is read-only. Started is 1 if the Personal Communications connection has been started; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' See if our connection is started
if myOIAObj.Started then
  call  connection_started
```

CommStarted

CommStarted is a Boolean flag that indicates whether the connection associated with this lsxECLOIA object is connected to the host data stream. The CommStarted property is an integer and is read-only. CommStarted is 1 if there is communication with the host; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' See if we are communicating with the host
if myOIAObj.CommStarted then
  call  communications_started
```

APIEnabled

APIEnabled is a Boolean flag that indicates whether the HLLAPI API has been enabled for the Personal Communications connection associated with this lsxECLOIA object. The APIEnabled property is an integer and is read-only. APIEnabled is 1 if the HLLAPI API is available; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' See if the HLLAPI API is enabled on this connection
if myOIAObj.APIEnabled then
  call  hllapi_available
```

Ready

Ready is a Boolean flag that indicates whether the Personal Communications connection associated with this lsxECLOIA object is ready. The Ready property is a combination of the Started, CommStarted and APIEnabled properties. It is an integer and is read-only. Ready is 1 if the Started, CommStarted and APIEnabled properties are 1; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLOIA object associated with connection A
dim myOIAObj as new lsxECLOIA("A")

' See if our connection is ready
if myOIAObj.Ready then
   call  conn_ready
```

lsxECLOIA Methods

There are no methods that are valid for the lsxECLOIA class.

lsxECLPS Class

The lsxECLPS class performs operations on a connection's presentation space.

The lsxECLPS object is associated with a Personal Communications connection when the lsxECLPS object is created. You cannot change the connection that is associated with an lsxECLPS object. If you want to manipulate the presentation space of a different connection, you must create a new lsxECLPS object associated with that connection.

There are three ways to create an lsxECLPS object:

1. Create a new lsxECLPS object by passing a Personal Communications connection name as a parameter on the new statement. A Personal Communications connection name is a single, alphabetic character from A-Z. The following is an example of creating an lsxECLPS object that is associated with Personal Communications connection A:

   ```
   ' Create an lsxECLPS object associated with PCOMM connection A
   dim myPSObj as new lsxECLPS("A")
   ```

2. Create a new lsxECLPS object by passing a Personal Communications connection handle as a parameter on the new statement. A Personal Communications connection handle is a long integer and is usually obtained by querying the lsxECLConnection object corresponding to the target Personal Communications connection (see "lsxECLConnMgr Class" on page 265 and "lsxECLConnection Class" on page 255 for more information on the properties and methods of those objects). The following is an example of creating an lsxECLPS object using a Personal Communications connection handle:

   ```
   dim myPSObj as lsxECLPS
   dim myCMgrObj as new lsxECLConnMgr

   ' Create a new lsxECLPS object associated with the first PCOMM connection
   ' found in lsxECLConnList
   set myPSObj = new lsxECLPS(myCMgrObj.lsxECLConnList(1).Handle)
   ```

3. Create an lsxECLSession object and an lsxECLPS object is automatically created. Access the lsxECLPS attribute to get to the lsxECLPS object contained in the lsxECLSession object. The following is an example of accessing the lsxECLPS object contained in an lsxECLSession object:

   ```
   dim myPSObj as lsxECLPS
   dim mySessionObj as lsxECLSession

   ' Create a new lsxECLSession object associated with PCOMM connection A
   set mySessionObj = new lsxECLSession("A")
   ' Get the lsxECLPS object from the lsxECLSession object
   set myPSObj = mySessionObj.lsxECLPS
   ```

Note: In the presentation space, the first row coordinate is row 1 and the first column coodinate is column 1. Therefore, the top, left position has a coordinate of row 1, column 1.

Properties

This section describes the properties of the lsxECLPS class

Type	Name	Attributes
Long	NumRows	Read-only
Long	NumCols	Read-only
Long	CursorPosRow	Read-only
Long	CursorPosCol	Read-only
lsxECLFieldList	lsxECLFieldList	Read-only
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Integer	Started	Read-only
Integer	CommStarted	Read-only
Integer	APIEnabled	Read-only
Integer	Ready	Read-only

NumRows

NumRows is the number of rows in this connection's presentation space. The NumRows property is a Long data type and is read-only. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSOBj as new lsxECLPS("A")

dim Rows as Long

' Get the number of rows in our presentation space
Rows = myPSObj.NumRows
```

NumCols

NumCols is the number of columns in this connection's presentation space. The NumCols property is a Long data type and is read-only. The following example shows this property.

```
:

' Create an lsxECLPS object associated with connection A
dim myPSOBj as new lsxECLPS("A")

dim Cols as Long

' Get the number of columns in our presentation space
Cols = myPSObj.NumCols
```

CursorPosRow

CursorPosRow is the row of the current cursor position in this connection's presentation space. The CursorPosRow is a Long data type and is read-only. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSOBj as new lsxECLPS("A")

dim CursorRow as Long

' Get the row location of the cursor in our presentation space
CursorRow = myPSObj.CursorPosRow
```

CursorPosCol

CursorPosCol is the column of the current cursor position in this connection's presentation space. The CursorPosCol is a Long data type and is read-only. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSOBj as new lsxECLPS("A")

dim CursorCol as Long

' Get the cursor column location in our presentation space
CusorCol = myPSObj.CursorPosCol
```

lsxECLFieldList

The lsxECLPS object contains an lsxECLFieldList object. See "lsxECLFieldList Class" on page 275 for details on the lsxECLFieldList methods and properties. The following example shows this object.

```
' Create an lsxECLPS object associated with PCOM connection A
dim myPSOBj as new lsxECLPS("A")

dim numFields as Long

' Get the number of fields in the presentation space
numFields = myPSObj.lsxECLFieldList.Count
```

Name

Name is the connection name of the Personal Communications connection associated with this lsxECLPS object. The Name property is a String data type and is read-only. Personal Communications connection names are one character in length and from the set of A-Z. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSOBj as new lsxECLPS("A")

dim myName as String

' Get our connection name
myName = myPSObj.Name
```

Handle

Handle is the connection handle of the Personal Communications connection associated with this lsxECLPS object. Handle is a Long data type and is read-only. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSOBj as new lsxECLPS("A")

dim myHandle as Long

' Get our connection handle
myHandle = myPSObj.Handle
```

ConnType

ConnType is the connection type of the connection that is associated with this lsxECLPS object. The ConnType is a String data type and is read-only. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSOBj as new lsxECLPS("A")

dim myConnType as String

' Get the connection type for connection A
myConnType = myPSObj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation
UNKNOWN	Unknown

CodePage

CodePage is the code page of the connection associated with this lsxECLPS object. The CodePage property is a Long data type, is read-only and cannot be changed through this LotusScript interface. However, the code page of a connection may change if the Personal Communications connection is restarted with a new configuration (see “lsxECLConnMgr Class” on page 265 for information about starting a connection). The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim myCodePage as Long

' Get the code page for connection A
myCodePage = myPSObj.CodePage
```

Started

Started is a Boolean flag that indicates whether the connection associated with this lsxECLPS object is started (for example, still running as a Personal Communications connection). The Started property is an integer and is read-only. Started is 1 if the Personal Communications connection has been started; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' See if our connection is started
if myPSObj.Started then
  call  connection_started
```

CommStarted

CommStarted is a Boolean flag that indicates whether the connection associated with this lsxECLPS object is connected to the host data stream. CommStarted is an integer and is read-only. CommStarted is 1 if there is communication with the host; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' See if we are communicating with the host
if myPSObj.CommStarted then
  call  communications_started
```

APIEnabled

APIEnabled is a Boolean flag that indicates whether the HLLAPI API has been enabled for the Personal Communications connection associated with this lsxECLPS object. APIEnabled is an integer and is read-only. APIEnabled is 1 if the HLLAPI API is available; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' See if the HLLAPI API is enabled on this connection
if myPSObj.APIEnabled then
  call  hllapi_available
```

Ready

The Ready property is a Boolean flag that indicates whether the Personal Communications connection associated with this lsxECLPS object is ready. The Ready property is a combination of the Started, CommStarted and APIEnabled properties. It is an integer and is read-only. Ready is 1 if the Started, CommStarted and APIEnabled properties are 1; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLPS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' See if our connection is ready
if myPSObj.Ready then
  call  conn_ready
```

lsxECLPS Methods

The following section describes the methods that are valid for the lsxECLPS class.

SetCursorPos(Long row, Long col)
SendKeys(String text, [optional] Long row, [optional] Long col)
SearchText(String text, [optional] Long Dir, [optional] Long row, [optional] Long col)
GetText(Long row, Long col, Long len)
SetText(String Text, [optional] Long row, [optional] Long col)
GetTextRect(Long startrow, Long startcol, Long endrow, Long endcol)

SetCursorPos

This method sets the position of the cursor in the presentation space of the connection associated with this lsxECLPS object. The cursor is set to the position indicated by the **row** and **col** parameters.

Prototype

SetCursorPos(Long row, Long col)

Parameters

Long row Target row for the cursor.

Long col Target column for the cursor.

Return Value

None

Example

The following example shows how to set the position of the cursor in the presentation space of the connection associated with this lsxECLPS object.

```
' Create an lsxECLPS object associated with connection A
dim myPSObj as new lsxECLPS("A")

' Set the cursor location in the presentation space
myPSObj.SetCursorPos(3,1)
```

SendKeys

This method sends a string of keystrokes to the presentation space of the connection associated with this lsxECLPS object. The string is positioned in the presentation space at the position indicated by the **row** and **col** parameters. The **row** and **col** parameters must be specified together. If the **row** and **col** parameters are not specified, the string is sent to the current cursor position.

Prototype

SendKeys(String text, [optional] Long row, [optional] Long col)

Parameters

String text String of keys to send to the presentation space.

Long row Target row within the presentation space. This parameter is optional. If the parameter is not specified, the location defaults to the current cursor row position. If row is specified, col must also be specified.

Long col Target column within the presentation space. This parameter is optional. If the parameter is not specified, the location defaults to the current cursor column position. If col is specified, row must also be specified

Return Value

None

Example

The following example shows how to send a string of keystrokes to the presentation space of the connection associated with this lsxECLPS object.

```
' Create an lsxECLPS object associated with connection A
dim PSObj as new lsxECLPS("A")

' Send a string of keystrokes to the cursor location in the presentation space
PSObj.SendKeys("[clear]QUERY DISK[ENTER]")

' Send a string of keystrokes to a specific location in the presentation space
PSObj.SendKeys("[clear]QUERY DISK[ENTER]", 23, 1)
```

Usage Notes

This method allows you to send mnemonic keystrokes to the presentation space. See Appendix A, “Sendkeys Mnemonic Keywords” on page 319 for a list of these keystrokes.

SearchText

This method searches for the first occurrence of a text string in the presentation space of the connection associated with this lsxECLPS object. This method returns a 1 if text is found; otherwise it returns a 0. The search begins from the position specified by the **row** and **col** parameters. The **row** and **col** parameters must be specified together. If the **row** and **col** parameters are not specified, the search begins at the beginning of the presentation space for a search forward or the end of the presentation space for a search backward. The search direction can either be forward or backward, and can be specified using the **dir** parameter. If **dir** is not specified, the default is forward.

Prototype

SearchText(String text, [optional] Long dir, [optional] Long row, [optional] Long col)

Parameters

String text Target text string.

Long dir Search direction. Must be **1** (Search forward) or **2** (Search Backward). This parameter is optional. If the parameter is not specified, the default is forward.

Long row Row position at which to start the search in the presentation space. The row of the located text is returned if the search is successful. This parameter is optional. If row is specified, col must also be specified.

Long col Column position at which to start the search in the presentation space. The column of the located text is returned if the search is successful. This parameter is optional. If col is specified, row must also be specified.

Return Value

Integer 1 if text found; 0 if text is not found.

Example

The following example shows how to search for the first occurrence of a text string in the presentation space of the connection associated with this lsxECLPS object.

```
' Create an lsxECLPS object associated with connection A
dim PSObj as new lsxECLPS("A")
dim tRow as Long
dim tCol as Long

tRow = 1
tCol = 1

' Search for a string in presentation space starting from the
' beginning of the presentation space.
if PSObj.SearchText("Alex",1) then
    call  found...

' Search for a string in presentation space starting from
' a specific location, the search direction is forward.
if PSObj.SearchText("ALEX", 1, tRow, tCol) then
  call  found...
```

GetText

This method retrieves a text string from the presentation space of the connection associated with this lsxECLPS object. The method returns a string starting at the position indicated by the **row** and **col** parameters for the length (**len**) parameter. If the **row**, **col** and **len** parameters are not specified, the entire presentation space is returned.

Prototype

GetText([optional] Long row, [optional] Long col, [optional] Long len)

Parameters

Long row Target row in the presentation space. This parameter is optional. If it is not specified, the entire presentation space is returned.

Long col Target column in the presentation space. This parameter is optional. If it is not specified, the entire presentation space is returned.

Long len Length of text to retrieve from the presentation space. This parameter is optional. If it is not specified, the entire presentation space is returned.

Return Value

String Text retrieved from the presentation space.

Example

The following example shows how to retrieve a text string from the presentation space of the connection associated with this lsxECLPS object.

```
' Create an lsxECLPS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim scrnText as String

' Get all the text from the text plane.
scrnText = myPSObj.GetText()

' Get 10 characters from the text plane starting
' at row 3, column 1
scrnText = myPSObj.GetText(3,1,10)
```

SetText

This method copies a text string to the presentation space of the connection associated with this lsxECLPS object. The string is copied to the location indicated by the row and col parameters. If the **row** and **col** parameters are not specified, the string is copied to the presentation space at the current cursor location. The **row** and **col** parameters must both be specified or omitted.

Prototype

SetText(String Text, [optional] Long row, [optional] Long col)

Parameters

String Text String to copy to the presentation space.

Long row Target row in the presentation space. This parameter is optional. If it is not specified, the current row position of the cursor is used. If row is specified, col must also be specified.

Long col Target column in the presentation space. This parameter is optional. If it is not specified, the current col position of the cursor is used. If col is specified, row must also be specified.

Return Value

None

Example

The following example shows how to copy a text string to the presentation space of the connection associated with an lsxECLPS object.

```
' Create an lsxECLPS object associated with ECL Connection A
dim myPSObj as new lsxECLPS("A")


' Copy a string to the current cursor position in the Presentation
' Space of ECL Connection A
myPSObj.SetText("Text to copy to PS")

' Copy a string to a specific location in the Presentation Space
' of ECL Connection A
myPSObj.SetText("Text to copy to PS", 23, 1)
```

GetTextRect

This method retrieves a text string from a rectangular area in the presentation space of the connection associated with this lsxECLPS object and returns a String data type. The rectangle is identified by the **startrow**, **startcol**, **endrow** and **endcol** parameters. No text wrapping is done during the text string retrieval; only the text within the designated rectangle is retrieved.

Prototype

GetTextRect(Long startrow, Long startcol, Long endrow, Long endcol)

Parameters

Long startrow Upper left row position of the rectangle in the presentation space.

Long startcol Upper left column position of the rectangle in the presentation space.

Long endrow Lower right row position of the rectangle in the presentation space.

Long endcol Lower right column position of the rectangle in the presentation space.

Return Value

String Text string retrieved from the presentation space.

Example

The following example shows how to retrieve a text string from a rectangular area in the presentation space of the connection associated with this lsxECLPS object and return a String data type.

```
' Create an lsxECLPS object associated with connection A
dim myPSObj as new lsxECLPS("A")

dim scrnText as String

' Get text from rectangle on the text plane
scrnText = myPSObj.GetTextRect(3,1,5,10)
```

lsxECLSession Class

The lsxECLSession class provides information about a host-connected connection. The lsxECLSession class also contains several other objects that correspond to the various pieces of a host-connected connection.

An lsxECLSession object is associated with a Personal Communications connection when the lsxECLSession object is created. You cannot change the connection that is associated with an lsxECLSession object. If you want to manage a different connection, you must create a new lsxECLSession object associated with that connection.

There are two ways to create an lsxECLSession object:

1. Create a new lsxECLSession object by passing a Personal Communications connection name as a parameter on the new statement. A Personal Communications connection name is a single, alphabetic character from A-Z. The following shows how to create an lsxECLSession object that is associated with Personal Communications connection A:

   ```
   ' Create an lsxECLSession object associated with PCOMM connection A
    dim mySessObj as new lsxECLSession("A")
   ```

2. Create a new lsxECLSession object by passing a Personal Communications connection handle as a parameter on the new statement. A Personal Communications connection handle is a Long integer, and is usually obtained by querying the lsxECLConnection object corresponding to the target Personal Communications connection (see “lsxECLConnMgr Class” on page 265 and “lsxECLConnection Class” on page 255 for more information on the properties and methods of those objects). The following example shows how to create an lsxECLSession object using a Personal Communications connection handle:

   ```
   dim mySessObj as lsxECLSession
   dim myConnObj as new lsxECLConnection

   ' Create a new lsxECLSession object using a connection handle
   set mySessObj = new lsxECLSession(myConnObj.Handle)
   ```

When an lsxECLSession object is created, contained lsxECLSession, lsxECLOIA, lsxECLXfer, and lsxECLWinMetrics objects are also created. Refer to them as you would any other property. The following is an example of accessing the lsxECLWinMetrics object within an lsxECLSession object:

```
' Set the host window to minimized
  mySessObj.lsxECLWinMetrics.Minimized = 1
```

Properties

This section describes the properties of the lsxECLSession class.

Type	Name	Attributes
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Integer	Started	Read-only
Integer	CommStarted	Read-only
Integer	APIEnabled	Read-only
Integer	Ready	Read-only
lsxECLPS	lsxECLPS	Read-only
lsxECLOIA	lsxECLOIA	Read-only
lsxECLXfer	lsxECLXfer	Read-only
lsxECLWinMetrics	lsxECLWinMetrics	Read-only

Name

Name is the connection name of the Personal Communications connection associated with this lsxECLSession object. The Name property is a String data type and is read-only. Personal Communications connection names are one character in length and from the character set A-Z. The following example shows this property.

```
' Create an lsxECLSession object associated with connection A
dim mySessObj as new lsxECLSession("A")

dim myName as String

' Get our connection name
myName = mySessObj.Name
```

Handle

Handle is the connection handle of the Personal Communications connection associated with this lsxECLSession object. The Handle property is a Long data type and is read-only. The following example shows this property.

```
' Create an lsxECLSession object associated with connection A
dim mySessObj as new lsxECLSession("A")

dim myHandle as Long

' Get our connection handle
myHandle = mySessObj.Handle
```

ConnType

ConnType is the connection type of the connection that is associated with this lsxECLSession object. The ConnType property is a String data type and is read-only. The following example shows this property.

```
' Create an lsxECLSession object associated with connection A
dim mySessObj as new lsxECLSession("A")

dim myConnType as String

' Get the connection type for connection A
myConnType = mySessObj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5270	5250 printer
ASCII	VT emulation
UNKNOWN	Unknown

CodePage

CodePage is the code page of the connection associated with this lsxECLSession object. The CodePage property is a Long data type, is read-only and cannot be changed through this LotusScript interface. However, the code page of a connection may change if the Personal Communications connection is restarted with a new configuration (see “lsxECLConnMgr Class” on page 265 for information about starting a connection). The following example shows this property.

```
' Create an lsxECLSession object associated with connection A
dim mySessObj as new lsxECLSession("A")

dim myCodePage as Long

' Get the code page for connection A
myCodePage = mySessObj.CodePage
```

Started

Started is a Boolean flag that indicates whether the connection associated with this lsxECLSession object is started (for example, still running as a Personal Communications connection). The Started property is an integer and is read-only. Started is 1 if the Personal Communications connection has been started; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLSession object associated with connection A
dim mySessObj as new lsxECLSession("A")

' See if our connection is started
if mySessObj.Started then
   call  connection_started
```

CommStarted

CommStarted is a Boolean flag that indicates whether the connection associated with this lsxECLSession object is connected to the host data stream. The CommStarted property is an integer and is read-only. CommStarted is 1 if there is communication with the host; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLSession object associated with connection A
dim mySessObj as new lsxECLSession("A")

' See if we are communicating with the host
if mySessObj.CommStarted then
  call  communications_started
```

APIEnabled

APIEnabled is a Boolean flag that indicates whether the HLLAPI API has been enabled for the Personal Communications connection associated with this lsxECLSession object. The APIEnabled property is an integer and is read-only. APIEnabled is 1 if the HLLAPI API is available; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLSession object associated with connection A
dim mySessObj as new lsxECLSession("A")

' See if the HLLAPI API is enabled on this connection
if mySessObj.APIEnabled then
  call  hllapi_available
```

Ready

Ready is a Boolean flag that indicates whether the Personal Communications connection associated with this lsxECLSession object is ready. The Ready property is a combination of the Started, CommStarted and APIEnabled properties. It is an integer and is read-only. Ready is 1 if the Started, CommStarted and APIEnabled properties are 1; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLSession object associated with connection A
dim mySessObj as new lsxECLSession("A")

' See if our connection is ready
if mySessObj.Ready then
  call  conn_ready
```

lsxECLPS

This is the lsxECLPS object contained within this lsxECLSession object. Refer to “lsxECLPS Class” on page 288 for a list of the properties and methods of this object. The following example shows this object.

```
' Connect to connection A
dim mySessObj as new lsxECLSession("A")

dim PSSize as Long

' Get the PS size from the contained lsxECLPS object
PSSize = mySessObj.lsxECLPS.Size
```

lsxECLOIA

This is the lsxECLOIA object contained within this lsxECLSession object. Refer to “lsxECLOIA Class” on page 279 for a list of the properties and methods of this object. The following example shows this object.

```
' Connect to connection A
dim mySessObj as new lsxECLSession("A")

' Check whether we have DBCS on this connection by querying
' the contained lsxECLOIA object.
if mySessObj.lsxECLOIA.DBCS then
  call  dbcs_enabled
```

lsxECLXfer

This is the lsxECLXfer object contained within this lsxECLSession object. Refer to “lsxECLXfer Class” on page 313 for a list of the properties and methods of this object. The following example shows this object.

```
' Connect to connection A
dim mySessObj as new lsxECLSession("A")

' Transfer a file to the host using the contained lsxECLXfer object
mySessObj.lsxECLXfer.Sendfile "c:\temp\filename.txt",
                        "filename  text  a0",
                        "CRLF  ASCII"
```

lsxECLWinMetrics

This is the lsxECLWinMetrics object contained within this lsxECLSession object. Refer to “lsxECLWinMetrics Class” on page 304 for a list of the properties and methods of this object. The following example shows this object.

```
' Connect to connection A
dim mySessObj as new lsxECLSession("A")

' Minimize the host window
mySessObj.lsxECLWinMetrics.Minimized = 1
```

lsxECLSession Methods

There are no methods that are valid for the lsxECLSession class.

lsxECLWinMetrics Class

The lsxECLWinMetrics class performs operations on a connection window. It allows you to perform window rectangle and position manipulation (for example, SetWindowRect, Ypos or Width), as well as window state manipulation (for example, Visible or Restored).

The lsxECLWinMetrics object is associated with a Personal Communications connection when the lsxECLWinMetrics object is created. You cannot change the connection that is associated with an lsxECLWinMetrics object. If you want to manipulate the window of a different connection, you must create a new lsxECLWinMetrics object associated with that connection.

There are three ways to create an lsxECLWinMetrics object:

1. Create a new lsxECLWinMetrics object by passing a Personal Communications connection name as a parameter on the new statement. A Personal Communications connection name is a single, alphabetic character from A-Z. The following is an example of creating an lsxECLWinMetrics object that is associated with Personal Communications connection A:

   ```
   ' Create an lsxECLWinMetrics object associated with PCOMM connection A
   dim myWMetObj as new lsxECLWinMetrics("A")
   ```

2. Create a new lsxECLWinMetrics object by passing a Personal Communications connection handle as a parameter on the new statement. A Personal Communications connection handle is a long integer and is usually obtained by querying the lsxECLConnection object corresponding to the target Personal Communications connection (see “lsxECLConnMgr Class” on page 265, “lsxECLConnList Class” on page 261 and “lsxECLConnection Class” on page 255 for more information on the properties and methods of those objects). The following is an example of creating an lsxECLWinMetrics object using a Personal Communications connection handle:

   ```
   dim myWMetObj as lsxECLWinMetrics
   dim myConnObj as new lsxECLConnection

   ' Create a new lsxECLWinMetrics object using a connection handle
     set myWMetObj = new
                         lsxECLWinMetrics(myConnObj.Handle)
   ```

3. Create an lsxECLSession object and an lsxECLWinMetrics object is automatically created. Access the lsxECLWinMetrics attribute to get to the lsxECLWinMetrics object contained in the lsxECLSession object. The following is an example of accessing the lsxECLWinMetrics object contained in an lsxECLSession object:

   ```
   dim myWMetObj as lsxECLWinMetrics
   dim mySessObj as lsxECLSession

   ' Create a new lsxECLSession object associated with PCOMM connection A
   set mySessObj = new lsxECLSession("A")
   ' Get the lsxECLWinMetrics object from the lsxECLSession object
   set myWMetObj = mySessObj.lsxECLWinMetrics
   ```

Properties

This section describes the properties for the lsxECLWinMetrics class.

Type	Name	Attributes
String	WindowTitle	Read-Write
Long	Xpos	Read-Write
Long	Ypos	Read-Write
Long	Width	Read-Write
Long	Height	Read-Write
Integer	Visible	Read-Write
Integer	Active	Read-Write
Integer	Minimized	Read-Write
Integer	Maximized	Read-Write
Integer	Restored	Read-Write
String	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Integer	Started	Read-only
Integer	CommStarted	Read-only
Integer	APIEnabled	Read-only
Integer	Ready	Read-only

WindowTitle

This is the title that is currently in the title bar for the connection associated with the lsxECLWinMetrics object. The WindowTitle property is a String data type and is read/write enabled.

Note: If Window Title is set to blank, the window title of the connection is restored to its original setting.

The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

 ' Set the window title
myWMetObj.WindowTitle = "Main Office"
```

Xpos

This is the *x* coordinate of the upper left point of the connection's window rectangle. The Xpos property is a Long data type and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. The following example shows this property.

```
 ' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

' Set the Xpos of the connection window
myWMetObj.Xpos = 0
```

Ypos

This is the *y* coordinate of the upper left point of the connection's window rectangle. The Ypos property is a Long data type and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. The following example shows this property.

```
 ' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

' Set the Ypos of the connection window
myWMetObj.Ypos = 0
```

Width

This is the width of the connection's window rectangle. The Width property is a Long data type and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. The following example shows this property.

```
 ' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

 ' Set the width of the connection window
myWMetObj.Width = 6081
```

Height

This is the height of the connection's window rectangle. The Height property is a Long data type and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. The following example shows this property.

```
 ' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

 ' Set the height of the connection window
myWMetObj.Height = 6081
```

Visible

This is a Boolean value that indicates whether the connection's window is visible. The Visible property is an integer and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. If the connection's window is visible, the Visible property has a value of 1; otherwise, it has a value of 0. The following example shows this property.

```
 ' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

' Make sure our window is visible
if myWMetObj.Visible = 0 then
   myWMetObj.Visible = 1
```

Active

This is a Boolean property that indicates whether the connection's window has the focus. The Active property is an integer and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. If the window has the focus, Active is set to 1; otherwise, is set to 0. The following example shows this property.

```
 ' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

' Make sure our window has the focus
if myWMetObj.Active = 0 then
   myWMetObj.Active = 1
```

Minimized

Minimized is a Boolean property that indicates whether the connection's window is minimized. The Minimized property is an integer and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. If the connection's window is minimized, the Minimized property is set to 1; otherwise, it is set to 0. The following example shows this property.

```
 ' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

' Make sure our window isn't minimized
if myWMetObj.Minimized then
   myWMetObj.Minimized = 0
```

Maximized

Maximized is a Boolean property that indicates whether the connection's window is maximized. The Maximized property is an integer and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. If the connection's window is maximized, the Maximized property is set to 1; otherwise, it is set to 0. The following example shows this property.

```
 ' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

' Make sure our window is maximized
if myWMetObj.Maximized = 0 then
   myWMetObj.Maximized = 1
```

Restored

This is a Boolean property that indicates whether the connection's window is in a restored state. The Restored property is an integer and is read/write enabled. However, if the connection to which you are attached is an inplace, embedded object, this property is read-only. If the connection's window is in a restored state, the Restored property is set to 1; otherwise, it is set to 0. The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as lsxECLWinMetrics("A")

' Make sure we're in a restored state
if myWMetObj.Restored = 0 then
   myWMetObj.Restored = 1
```

Name

Name is the connection name of the Personal Communications connection associated with this lsxECLWinMetrics object. The Name property is a String data type and is read-only. Personal Communications connection names are one character in length and from the character set A-Z. The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

dim myName as String

' Get our connection name
myName = myWMetObj.Name
```

Handle

Handle is the connection handle of the Personal Communications connection associated with this lsxECLWinMetrics object. The Handle property is a Long data type and is read-only. The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

dim myHandle as Long

' Get our connection handle
myHandle = myWMetObj.Handle
```

ConnType

The connection type of the connection that is associated with this lsxECLWinMetrics object. The ConnType property is a String data type and is read-only. The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

dim myConnType as String

' Get the connection type for connection A
myConnType = myWMetObj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5250	5250 display
PRNT3270	3270 printer
PRNT5250	5250 printer
ASCII	VT emulation
UNKNOWN	Unknown

CodePage

CodePage is the code page of the connection associated with this lsxECLWinMetrics object. The CodePage property is a Long data type, is read-only and cannot be changed through this LotusScript interface. However, the code page of a connection may change if the Personal Communications connection is restarted with a new configuration (see “lsxECLConnMgr Class” on page 265 for information about starting a connection). The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

dim myCodePage as Long

' Get the code page for connection A
myCodePage = myWMetObj.CodePage
```

Started

Started is a Boolean flag that indicates whether the connection associated with this lsxECLWinMetrics object is started (for example, still running as a Personal Communications connection). This property is an integer and is read-only. The Started property is 1 if the Personal Communications connection has been started; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

' See if our connection is started
if myWMetObj.Started then
  call  connection_started
```

CommStarted

CommStarted is a Boolean flag that indicates whether the connection associated with this lsxECLWinMetrics object is connected to the host data stream. The CommStarted property is an integer and is read-only. CommStarted is 1 if there is communication with the host; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

' See if we are communicating with the host
if myWMetObj.CommStarted then
  call  communications_started
```

APIEnabled

APIEnabled is a Boolean flag that indicates whether the HLLAPI API has been enabled for the Personal Communications connection associated with this lsxECLWinMetrics object. The APIEnabled property is an integer and is read-only. APIEnabled is 1 if the HLLAPI API is available; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

' See if the HLLAPI API is enabled on this connection
if myWMetObj.APIEnabled then
  call  hllapi_available
```

Ready

Ready is a Boolean flag that indicates whether the Personal Communications connection associated with this lsxECLWinMetrics object is ready. The Ready property is a combination of the Started, CommStarted and APIEnabled properties. It is an integer and is read-only. Ready is 1 if the Started, CommStarted and APIEnabled properties are 1; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

' See if our connection is ready
if myWMetObj.Ready then
  call  conn_ready
```

lsxECLWinMetrics Methods

The following section describes the methods that are valid for lsxECLWinMetrics.

void SetWindowRect(Long left, Long top, Long right, Long bottom)
void GetWindowRect(Long left, Long top, Long right, Long bottom)

GetWindowRect

This method returns the coordinates of the top, bottom, left and right sides of the window rectangle associated with this connection. The supplied parameters are set to the coordinates of the window rectangle.

Prototype

GetWindowRect(Long left, Long top, Long right, Long bottom)

Parameters

Long left The coordinate of the left side of the window rectangle.

Long top The coordinate of the top of the window rectangle.

Long right The coordinate of the right side of the window rectangle.

Long bottom The coordinate of the bottom of the window rectangle.

Return Value

None

Example

The following example shows how to return the coordinates of the top, bottom, left and right sides of the window rectangle associated with this connection.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

' Create some variables to hold our window coordinates
dim left as Long
dim top as Long
dim right as Long
dim bottom as Long

 ' Get the window coordinates
myWMetObj.GetWindowRect left, top, right, bottom
```

SetWindowRect

This method sets the coordinates of the top, bottom, left and right sides of the window rectangle associated with this connection.

Prototype

SetWindowRect(Long left, Long top, Long right, Long bottom)

Parameters

Long left The new coordinate of the left side of the window rectangle.

Long top Thenew coordinate of the top of the window rectangle.

Long right The new coordinate of the right side of the window rectangle.

Long bottom The new coordinate of the bottom of the window rectangle.

Return Value

None

Example

The following example shows how to set the coordinates of the top, bottom, left and right sides of the window rectangle associated with this connection.

```
' Create an lsxECLWinMetrics object associated with connection A
dim myWMetObj as new lsxECLWinMetrics("A")

' Set the window coordinates
myWMetObj.SetWindowRect 0, 0, 6081, 6081
```

lsxECLXfer Class

The lsxECLXfer Class provides file transfer services between a host and a client. The transfer is done through a Personal Communications connection and therefore, the lsxECLXfer object must be associated with a Personal Communications connection.

The lsxECLXfer object is associated with a Personal Communications connection when the lsxECLXfer object is created. You cannot change the connection that is associated with an lsxECLXfer object. If you want to transfer files on a different connection, you must create a new lsxECLXfer object associated with that connection.

There are three ways to create an lsxECLXfer object:

1. Create a new lsxECLXfer object by passing a Personal Communications connection name as a parameter on the new statement. A Personal Communications connection name is a single, alphabetic character from A-Z. The following is an example of creating an lsxECLXfer object that is associated with Personal Communications connection A:

   ```
   ' Create an lsxECLXfer object associated with PCOMM connection A
   dim myXferObj as new lsxECLXfer("A")
   ```

2. Create a new lsxECLXfer object by passing a Personal Communications connection handle as a parameter on the new statement. A Personal Communications connection handle is a long integer and is usually obtained by querying the lsxECLConnection object corresponding to the target Personal Communications connection (see “lsxECLConnMgr Class” on page 265, “lsxECLConnList Class” on page 261 and “lsxECLConnection Class” on page 255 for more information on the properties and methods of those objects). The following is an example of creating an lsxECLXfer object using a Personal Communications connection handle:

   ```
   dim myXferObj as lsxECLXfer
   dim myConnObj as new lsxECLConnection

   ' Create a new lsxECLXfer object using the connection handle
   set myXferObj = new lsxECLXfer(myConnObj.Handle)
   ```

3. Create an lsxECLSession object and an lsxECLXfer object is automatically created. Access the lsxECLXfer attribute to get to the lsxECLXfer object contained in the lsxECLSession object. The following is an example of how to access the lsxECLXfer object contained in an lsxECLSession object:

   ```
   dim myXferObj as lsxECLXfer
   dim lsxECLSessionObj as lsxECLSession

   ' Create a new lsxECLSession object associated with PCOMM connection A
   set lsxECLSessionObj = new lsxECLSession("A")
   ' Get the lsxECLXfer object from the lsxECLSession object
   set myXferObj = lsxECLSessionObj.lsxECLXfer
   ```

Properties

This section describes the properties of the lsxECLXfer class.

Type	Name	Attribute
Long	Name	Read-only
Long	Handle	Read-only
String	ConnType	Read-only
Long	CodePage	Read-only
Integer	Started	Read-only
Integer	CommStarted	Read-only
Integer	APIEnabled	Read-only
Integer	Ready	Read-only

Name

Name is the connection name of the Personal Communications connection associated with this lsxECLXfer object. The Name property is a String data type and is read-only. Personal Communications connection names are one character in length and from the character set A-Z. The following example shows this property.

```
' Create an lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

dim myName as String

' Get our connection name
myName = myXferObj.Name
```

Handle

Handle is the connection handle of the Personal Communications connection associated with this lsxECLXfer object. The Handle property is a Long data type and is read-only. The following example shows this property.

```
' Create an lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

dim myHandle as Long

' Get our connection handle
myHandle = myXferObj.Handle
```

ConnType

ConnType is the connection type of the connection that is associated with this lsxECLXfer object. The ConnType property is a String data type and is read-only. The following example shows this property.

```
' Create an lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

dim myConnType as String

' Get the connection type for connection A
myConnType = myXferObj.ConnType
```

Connection types for the ConnType property are:

String Returned	Meaning
DISP3270	3270 display
DISP5270	5250 display
PRNT3270	3270 printer
PRNT5270	5250 printer
ASCII	VT emulation
UNKNOWN	Unknown

CodePage

CodePage is the code page of the connection associated with this lsxECLXfer object. The CodePage property is a Long data type, is read-only and cannot be changed through this LotusScript interface. However, the code page of a connection may change if the Personal Communications connection is restarted with a new configuration (see “lsxECLConnMgr Class” on page 265 for information about starting a connection). The following example shows this property.

```
' Create an lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

dim myCodePage as Long

' Get the code page for connection A
myCodePage = myXferObj.CodePage
```

Started

Started is a Boolean flag that indicates whether the connection associated with this lsxECLXfer object is started (for example, still running as a Personal Communications connection). The Started property is an integer and is read-only. Started is 1 if the Personal Communications connection has been started; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

' See if our connection is started
if myXferObj.Started then
   call  connection_started
```

CommStarted

CommStarted is a Boolean flag that indicates whether the connection associated with this lsxECLXfer object is connected to the host data stream. The CommStarted property is an integer and is read-only. CommStarted is 1 if there is communication with the host; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

' See if we are communicating with the host
if myXferObj.CommStarted then
  call  communications_started
```

APIEnabled

APIEnabled is a Boolean flag that indicates whether the HLLAPI API has been enabled for the Personal Communications connection associated with this lsxECLXfer object. The APIEnabled property is an integer and is read-only. APIEnabled is 1 if the HLLAPI API is available; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

' See if the HLLAPI API is enabled on this connection
if myXferObj.APIEnabled then
  call  hllapi_available
```

Ready

Ready is a Boolean flag that indicates whether the Personal Communications connection associated with this lsxECLXfer object is ready. The Ready property is a combination of the Started, CommStarted and APIEnabled properties. It is an integer and is read-only. Ready is 1 if the Started, CommStarted and APIEnabled properties are 1; otherwise, it is 0. The following example shows this property.

```
' Create an lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

' See if our connection is ready
if myXferObj.Ready then
  call  conn_ready
```

lsxECLXfer Methods

The following section describes the methods that are valid for the lsxECLXfer class.

SendFile (String PCFile, String HostFile, String Options)
ReceiveFile (String PCFile, String HostFile, String Options)

SendFile

This method sends a file from the workstation to the host.

Prototype

SendFile(String PCFile, String HostFile, String Options)

Parameters

String PCFile Name of the source file on the workstation.

String HostFile Name of the target file on the host.

String Options File transfer options (see 317).

Return Value

None

Example

The following example shows how to send a file from the workstation to the host.

```
' Create a new lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

' Send a file from my PC to the host on connection A,
' Assume the host is a VM/CMS host
myXferObj.SendFile "c:\windows\temp\thefile.txt",
                        "THEFILE TEXT A",
                       "CRLF  ASCII"
```

Usage Notes

File transfer options are host-dependent. The following is a list of some of the valid host options for a VM/CMS host:

ASCII
JISCII
CRLF
APPEND
TIME n
CLEAR
NOCLEAR
PROGRESS
QUIET

Refer to the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for the list of supported hosts and associated file transfer options.

ReceiveFile

This method receives a file from the host to the workstation.

Prototype

ReceiveFile(String PCFile, String HostFile, String Options)

Parameters

String PCFile Name of the file on the workstation.

String HostFile Name of the file on the host.

String Options File transfer options (see 318).

Return Value

None

Example

The following example shows how to receive a file from the host to the workstation.

```
' Create a new lsxECLXfer object associated with connection A
dim myXferObj as new lsxECLXfer("A")

' Receive a file from host connection A onto my workstation,
' Assume the host is a VM/CMS host
myXferObj.ReceiveFile "c:\windows\temp\thefile.txt",
                        "THEFILE TEXT A0",
                       "CRLF  ASCII"
```

Usage Notes

File transfer options are host-dependent. For example, a list of some of the valid host options for a VM/CMS host are:

ASCII
JISCII
CRLF
APPEND
TIME n
CLEAR
NOCLEAR
PROGRESS
QUIET

Refer to the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for the list of supported hosts and associated file transfer options.

Appendix A. Sendkeys Mnemonic Keywords

This chapter contains the mnemonic keywords for the Sendkeys method. Refer to the *IBM eNetwork Personal Communications Version 4.2 for Windows 95 and Windows NT Emulator Programming* manual for additional details regarding the meanings of the keywords.

Table 1 (Page 1 of 3). Mnemonic Keywords for the Sendkey Method

Keyword	**Description**
[backtab]	Back tab
[clear]	Clear screen
[delete]	Delete
[enter]	Enter
[eraseeof]	Erase end of file
[help]	Help
[insert]	Insert
[jump]	Jump
[left]	Left
[newline]	New line
[space]	Space
[print]	Print
[reset]	Reset
[tab]	Tab
[up]	Up
[Down]	Down
[dbcs]	DBCS
[capslock]	CapsLock
[right]	Right
[home]	Home
[pf1]	PF2
[pf2]	PF2
[pf3]	PF3
[pf4]	PF4
[pf5]	PF5
[pf6]	PF6
[pf7]	PF7
[pf8]	PF8
[pf9]	PF9
[pf10]	PF10
[pf11]	PF11
[pf12]	PF12

© Copyright IBM Corp. 1997

Table 1 (Page 2 of 3). Mnemonic Keywords for the Sendkey Method	
Keyword	**Description**
[pf13]	PF13
[pf14]	PF14
[pf15]	PF15
[pf16]	PF16
[pf17]	PF17
[pf18]	PF18
[pf19]	PF19
[pf20]	PF20
[pf21]	PF21
[pf22]	PF22
[pf23]	PF23
[pf24]	PF24
[eof]	End of file
[scrlock]	Scroll Lock
[numlock]	Num Lock
[pageup]	Page Up
[pagedn]	Page Down
[pa1]	PA 1
[pa2]	PA 2
[pa3]	PA 3
[test]	Test
[worddel]	Word Delete
[fldext]	Field Exit
[erinp]	Erase Input
[sysreq]	System Request
[instog]	Insert Toggle
[crsel]	Cursor Select
[fastleft]	Cursor Left Fast
[attn]	Attention
[devcance]	Device Cancel
[printps]	Print Presentation Space
[fastup]	Cursor Up Fast
[fastdown]	Cursor Down Fast
[hex]	Hex
[fastright]	Cursor Right Fast
[revvideo]	Reverse Video
[underscr]	Underscore
[rstvideo]	Reset Reverse Video

Table 1 (Page 3 of 3). Mnemonic Keywords for the Sendkey Method

Keyword	Description
[red]	Red
[pink]	Pink
[green]	Green
[yellow]	Yellow
[blue]	Blue
[turq]	Turquoise
[white]	White
[rstcolor]	Reset Host Color
[printpc]	Print (PC)
[wordright]	Forward Word Tab
[wordleft]	Backward Word Tab
[field-]	Field -
[field+]	Field +
[rcdbacksp]	Record Backspace
[printhost]	Print Presentation Space on Host
[dup]	Dup
[fieldmark]	Field Mark
[dispsosi]	Display SO/SI
[gensosi]	Generate SO/SI
[dispattr]	Display Attribute
[fwdchar]	Forward Character
[splitbar]	Split Vertical Bar
[altcsr]	Alternate Cursor
[backspace]	Backspace
[null]	Null

Appendix B. ECL Planes — Format and Content

This appendix describes the format and contents of the different data planes in the ECL presentation space model. Each plane represents a distinct aspect of the host presentation space, such as its character contents, color specifications, field attributes, etc. The ECL::GetScreen methods and others return data from the different presentation space planes.

Each plane contains one byte per host presentation space character position. Each plane is described in the following sections in terms of its logical contents and data format. The plane types are enumerated in the ECLPS.HPP header file.

TextPlane

The text plane represents the visible characters of the presentation space. Non-display fields are shown in the text plane. The byte value of each element of the text plane corresponds to the ASCII value of the displayed character. The text plane does not contain any binary zero (null) character values. Any null characters in the presentation space (such as null-padded input fields) are represented as ASCII blank (0x20) characters.

FieldPlane

The field plane represents the field positions and their attributes in the presentation space. This plane is meaningful only for field-formatted presentation spaces. (For example, VT connections are not formatted).

This plane is a sparse-array of field attribute values. All values in this plane are binary zero except for where field attribute characters are present in the presentation space. At those positions, the values are the attributes of the field which starts at that location. The length of a field is the linear distance between the field attribute position and the next field attribute in the presentation space, not including the attribute position itself.

The value of the field attribute positions are as shown in the following tables.

Note: Attribute values are different for different types of connections.

© Copyright IBM Corp. 1997

Table 2. 3270 Field Attributes		
Bit Position (0 is least significant bit)	**Meaning**	
7	Always "1"	
6	Always "1"	
5	**0**	Unprotected
	1	Protected
4	**0**	Alphanumeric data
	1	Numeric data only
3, 2	**0, 0**	Normal intensity, not pen detectable
	0, 1	Normal intensity, pen detectable
	1, 0	High intensity, pen detectable
	1, 1	Non-display, not pen detectable
1	Reserved	
0	**0**	Field has not been modified
	1	Unprotected field has been modified

Table 3. 5250 Field Attributes		
Bit Position (0 is least significant bit)	**Meaning**	
7	Always "1"	
6	**0**	Nondisplay
	1	Display
5	**0**	Unprotected
	1	Protected
4	**0**	Normal intensity
	1	High intensity
3, 2, 1	**0, 0, 0**	Alphanumeric data
	0, 0, 1	Alpha only
	0, 1, 0	Numeric shift
	0, 1, 1	Numeric data plus numeric specials
	1, 0, 1	Numeric only
	1, 1, 0	Magnetic stripe reading device data only
	1, 1, 1	Signed numeric only
0	**0**	Field has not been modified
	1	Unprotected field has been modified

ColorPlane

The color plane contains color information for each character of the presentation space. The foreground and background color of each character is represented as it is specified in the host data stream. The colors in the color plane are not modified by any color display mapping of the emulator window. Each byte of the color plane contains the following color information.

Table 4. Color Plane Information

Bit Position (0 is least significant bit)	Meaning
7 - 4	**Background character color** **0x0** Blank **0x1** Blue **0x2** Green **0x3** Cyan **0x4** Red **0x5** Magenta **0x6** Brown (3270), Yellow (5250) **0x7** White
6	**Foreground character color** **0x0** Blank **0x1** Blue **0x2** Green **0x3** Cyan **0x4** Red **0x5** Magenta **0x6** Brown (3270), Yellow (5250) **0x7** White (normal intensity) **0x8** Gray **0x9** Light blue **0xA** Light green **0xB** Light cyan **0xC** Light red **0xD** Light magenta **0xE** Yellow **0xF** White (high intensity)

ExtendedFieldPlane

This plane contains extended character attribute data.

This plane is a sparse-array of extended character attribute values. All values in the array are binary zero except for character in the presentation space for which the host has specified extended character attributes. The meaning of the extended character attribute values are as follows.

Table 5. 3270 Extended Character Attributes

Bit Position (0 is least significant bit)	Meaning	
7, 6	**Character highlighting**	
	0, 0	Normal
	0, 1	Blink
	1, 0	Reverse video
	1, 1	Underline
5, 4, 3	**Character color**	
	0, 0, 0	Default
	0, 0, 1	Blue
	0, 1, 0	Red
	0, 1, 1	Pink
	1, 0, 0	Green
	1, 0, 1	Turquoise
	1, 1, 0	Yellow
	1, 1, 1	White
2, 1	**Character attribute**	
	00	Default
	11	Double byte character
0	Reserved	

Table 6. 5250 Extended Character Attributes

Bit Position (0 is least significant bit)	Meaning	
7	**0**	Normal image
	1	Reverse image
6	**0**	No underline
	1	Underline
5	**0**	No blink
	1	Blink
4	**0**	No column separator
	1	Column separator
3, 2, 1, 0	Reserved	

Appendix C. Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make them 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:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594
USA

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:

Site Counsel
IBM Corporation
P.O. Box 12195
3039 Cornwallis Road
Research Triangle Park, NC
27709-2195
USA

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

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

© Copyright IBM Corp. 1997

Appendix D. Trademarks

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

IBM
Operating System/2
OS/2
Virtual Machine/Enterprise Systems Architecture
VM/ESA
VTAM

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

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

C-bus is a trademark of Corollary, Inc.

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

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

© Copyright IBM Corp. 1997

Index

A

Active 158, 233, 307
Alphanumeric 196, 280
APIEnabled 176, 202, 211, 224, 237, 245, 257, 285, 292, 301, 310, 316
APL 196, 280
autECLConnList 173
  methods 177
    FindConnectionByHandle 177
    FindConnectionByName 178
    Refresh 177
  properties 173
    APIEnabled 176
    CodePage 175
    CommStarted 175
    ConnType 174
    Count 173
    Handle 174
    Name 174
    Ready 176
    Started 175
  usage notes 173
autECLConnList Methods 177
autECLConnMgr 181
  methods 182
    StartConnection 182
    StopConnection 183
  properties 181
    autECLConnList 181
autECLConnMgr Methods 182
autECLFieldList 184
  methods 191
    FindFieldByRowCol 192
    FindFieldByText 192
    GetScreen 193
    Refresh 191
    SetText 194
  properties 184
    Count 184
    Display 190
    EndCol 186
    EndRow 186
    HighIntensity 189
    Length 187
    Modified 187
    Numerics 188
    PenDetectable 189
    Protected 188
    StartCol 185
    StartRow 185
  usage notes 184
autECLFieldList Methods 191
autECLOIA 195
  methods 204
    SetConnectionByHandle 204
    SetConnectionByName 204
  properties 195
    Alphanumeric 196
    APIEnabled 202
    APL 196
    CapsLock 198
    CodePage 201
    CommErrorReminder 199
    CommStarted 202
    ConnType 200
    DBCS 197
    Handle 200
    Hiragana 197
    InputInhibited 199
    InsertMode 198
    Katakana 196
    MessageWaiting 199
    Name 200
    NumLock 198
    Ready 202
    Started 201
    UpperShift 197
  usage notes 195
autECLOIA Methods 204
autECLOIA object 225
autECLPS 207
  methods 213
    GetScreen 216
    GetScreenRect 218
    SearchText 215
    SendKey 215
    SetConnectionByHandle 213
    SetConnectionByName 213
    SetCursorPos 214
    SetScreen 217
  properties 207
    APIEnabled 211
    autECLFieldList 209
    CodePage 210
    CommStarted 211
    ConnType 210
    CursorPosCol 208
    CursorPosRow 208
    Handle 209
    Name 209
    NumCols 208
    NumRows 208
    Ready 212
    Started 211

© Copyright IBM Corp. 1997

autECLPS *(continued)*
usage notes 207
autECLPS Methods 213
autECLPS object 225
autECLSession 221
methods 227
SetConnectionByHandle 227
SetConnectionByName 227
properties 221
APIEnabled 224
autECLOIA object 225
autECLPS object 225
autECLWinMetrics object 226
autECLXfer object 225
CodePage 223
CommStarted 223
ConnType 222
Handle 222
Name 222
Ready 224
Started 223
usage notes 221
autECLSession Methods 227
autECLWinMetrics 230
methods 239
GetWindowRect 240
SetConnectionByHandle 239
SetConnectionByName 239
SetWindowRect 241
properties 230
Active 233
APIEnabled 237
CodePage 236
CommStarted 237
ConnType 236
Handle 236
Height 232
Maximized 234
Minimized 234
Name 235
Ready 238
Restored 235
Started 237
Visible 233
Width 232
WindowTitle 231
Xpos 231
Ypos 231
usage notes 230
autECLWinMetrics Methods 239
autECLWinMetrics object 226
autECLXfer 243
methods 247
ReceiveFile 249
SendFile 248
SetConnectionByHandle 247
SetConnectionByName 247
autECLXfer *(continued)*
properties 243
APIEnabled 245
CodePage 244
CommStarted 245
ConnType 244
Handle 244
Name 243
Ready 246
Started 245
usage notes 243
autECLXfer Methods 247
autECLXfer object 225
automation objects 171
autECLConnList 173
autECLConnMgr 181
autECLFieldList 184
autECLOIA 195
autECLPS 207
autECLSession 221
autECLWinMetrics 230
autECLXfer 243
description 171

B

Building C++ ECL Programs 14
Microsoft Visual C++ 15

C

C++ objects 11
description 11
ECLBase 17
ECLConnection 23
ECLConnList 37
ECLConnMgr 45
ECLConnNotify 53
ECLErr 59
ECLField 63
ECLFieldList 77
ECLKeyNotify 84
ECLOIA 90
ECLPS 101
ECLSession 125
ECLStartNotify 133
ECLUpdateNotify 138
ECLWinMetrics 143
ECLXfer 164
CapsLock 198, 282
CLXfer Constructor 165
CodePage 175, 201, 210, 223, 236, 244, 257, 284, 291, 300, 309, 315
Collection Element Methods 177, 191
ColorPlane 325

CommErrorReminder 199, 282
CommStarted 175, 202, 211, 223, 237, 245, 257, 285, 292, 301, 309, 316
ConnType 174, 200, 210, 222, 236, 244, 256, 284, 291, 299, 308, 314
Convert Pos 21
ConvertHandle2ShortName 18
ConvertPosToCol 120
ConvertPosToRow 120
ConvertPosToRowCol 118
ConvertRowColToPos 119
ConvertShortName2Handle 19
ConvertTypeToString 20
Count 261, 275
CursorPosCol 208, 290
CursorPosRow 208, 290

D

DBCS 197, 281
Derivation 23, 37, 45, 54, 59, 63, 77, 85, 90, 101, 125, 134, 139, 143, 164
Display 190, 271

E

ECL Concepts 2
  Addressing 5
  Connections, Handles and Names 2
  ECL Container Objects 3
  ECL List Objects 4
  Error Handling 5
  Events 4
  Sessions 3
ECL Planes 323
ECLBase 17
ECLBase Methods 18
ECLCommNotify Methods 57
ECLConnection 23
  derivation 23
  methods 24
    ECLConnection Constructor 24
    GetCodePage 32
    GetConnType 27
    GetHandle 26
    GetName 28
    IsAPIEnabled 31
    IsCommStarted 30
    IsReady 32
    IsStarted 29
    StartCommunication 33
    StopCommunication 34, 35
ECLConnection Constructor 24
ECLConnection Destructor 25
ECLConnection Methods 24
ECLConnList 37
  derivation 37
  methods 38
    ECLConnList Constructor 38
    ECLConnList Destructor 39
    FindConnection 41
    GetFirstConnection 39
    GetNextConnection 40
    Refresh 43
  usage notes 37
ECLConnList Constructor 38
ECLConnList Destructor 39
ECLConnList Methods 38
ECLConnMgr 45
  methods
    ECLConnList 47
    RegisterStartEvent 50
    StartConnection 48
    StopConnection 49
    UnregisterStartEvent 51
ECLConnMgr Constructor 46
ECLConnMgr Deconstructor 47
ECLConnMgr Methods 46
ECLConnNotify 53
ECLErr 59
  derivation 59
  methods 60
    const GetMessageNumber 60
    GetMsgText 61
    GetReasonCode 61
ECLErr Methods 60
ECLField 63
  derivation 63
  methods 64
    GetAttribute 75
    GetEnd 67
    GetEndCol 69
    GetEndRow 68
    GetLength 70
    GetScreen 71
    GetStart 64
    GetStartCol 66
    GetStartRow 65
    IsDisplay 74
    IsHighIntensity 74
    IsModified 74
    IsNumerics 74
    IsPenDetectable 74
    IsProtected 74
    SetText 72
ECLField Methods 64
ECLFieldList 77
  derivation 77
  methods 78
    FindField 82
    GetFieldCount 79
    GetFirstField 80

ECLFieldList *(continued)*
methods *(continued)*
GetNextField 81
Refresh 78
properties 77
ECLFieldList Methods 78
ECLKeyNotify 84
ECLKeyNotify Methods 88
ECLOIA 90
derivation 90
methods 91
ECLOIA Constructor 91
IsAlphanumeric 92
IsAPL 93
IsCapsLock 97
IsCommErrorReminder 98
IsDBCS 95
IsHiragana 94
IsInputInhibited 99
IsInsertMode 97
IsKatakana 93
IsMessageWaiting 99
IsNumLock 96
IsUpperShift 95
usage notes 90
ECLOIA Constructor 91
ECLOIA Methods 91
ECLPS 101
derivation 101
methods 102
ConvertPosToRowCol 118
ConvertRowColtoPos 119
ECLFieldList 123
ECLPS 102, 103
GetCursorPos 106
GetScreen 113, 115
GetSize 104
RegisterKeyEvent 121
SearchText 111
SendKeys 110
SetCursorPos 109
UnregisterKeyEvent 123
properties 101
usage notes 101
ECLPS Constructor 102
ECLPS Destructor 103
ECLPS Methods 102
ECLSession 125
derivation 125
methods 126
ECLSession Constructor 126
ECLSession Destructor 127
GetOIA 128
GetPS 127
GetWinMetrics 130
GetXfer 129
RegisterUpdateEvent 131
ECLSession *(continued)*
methods *(continued)*
UnregisterUpdateEvent 132
properties 125
usage notes 125
ECLSession Constructor 126
ECLSession Destructor 127
ECLSession Methods 126
ECLStartNotify 133
ECLStartNotify Methods 136
ECLUpdateNotify 138
ECLUpdateNotify Methods 141
ECLWinMetrics 143
derivation 139, 143
methods 144
Active 158
ECLWinMetrics 144
GetHeight 153
GetWidth 151
GetWindowRect 155
GetWindowTitle 146
GetXpos 148
GetYpos 149
IsMaximized 160
IsMinimized 159
IsRestored 161
IsVisible 157
SetActive 158
SetHeight 154
SetMaximized 161
SetMinimized 160
SetRestored 162
SetVisible 157
SetWidth 152
SetWindowRect 156
SetWindowTitle 147
SetXpos 148
SetYpos 150
properties 143
usage notes 143
ECLWinMetrics Constructor 144
ECLWinMetrics Destructor 145
ECLWinMetrics Methods 144
ECLXfer 164
derivation 164
methods 165
ECLXfer Constructor 165
ECLXfer Destructor 166
ReceiveFile 168
SendFile 166
properties 164
usage notes 164
ECLXfer Destructor 166
ECLXfer Methods 165
ELLHAPI, migrating from 6
Events 9

ELLHAPI, migrating from *(continued)*
Execution/Language Interface 6
Features 6
Presentation Space Models 8
PS Connect/Disconnect, Multithreading 9
SendKey Interface 9
Session IDs 7
EndCol 186, 269
EndRow 186, 269
etConnectionByHandle 239
ExtendedFieldPlane 326

F

FieldPlane 323
FindConnection 41
FindConnectionByHandle 177, 263
FindConnectionByName 178, 264
FindField 82
FindFieldByRowCol 192, 276
FindFieldByText 192, 277

G

GetAttribute 75
GetCodePage 32
GetConnList 47
GetConnType 27
GetCount 42
GetCursorPos 106
GetCursorPosCol 108
GetCursorPosRow 107
GetEnd 67
GetEndCol 69
GetEndRow 68
GetFieldCount 79
GetFirstConnection 39
GetFirstField 80
GetHeight 153
GetLength 70
GetMsgNumber 60
GetMsgText 61
GetName 28
GetNextConnection 40
GetNextField 81
GetOIA 128
GetPS 127
GetReasonCode 61
GetSize 104
GetSizeCols 106
GetSizeRows 105
GetStart 64
GetStartCol 66
GetStartRow 65
GetText 71, 113, 193, 216, 273, 295
GetTextRect 115, 218, 297
GetVersion 18
GetWidth 151
GetWindowRect 155, 240, 311
GetWindowTitle 146
GetWinMetrics 130
GetXfer 129
GetXpos 148
GetYpos 149

H

Handle 174, 200, 209, 222, 236, 244, 256, 284, 291, 299, 308, 314
Height 232, 306
HighIntensity 189, 271
highlights xiii
Hiragana 197, 281

I

InputInhibited 99, 199, 283
InsertMode 198, 282
IsAlphanumeric 92
IsAPIEnabled 31
IsAPL 93
IsCapsLock 97
IsCommErrorReminder 98
IsCommStarted 30
IsDBCS 95
IsDisplay 74
IsHighIntensity 74
IsHiragana 94
IsInsertMode 97
IsKatakana 93
IsMaximized 160
IsMessageWaiting 99
IsMinimized 159
IsModified 74
IsNumeric 74, 96
IsPenDetectable 74
IsProtected 74
IsReady 32
IsRestored 161
IsStarted 29
IsUpperShift 95
IsVisible 157

K

Katakana 196, 281
keywords 319

L

Length 187, 269

LotusScript objects 253
lsxECLConnection 255
lsxECLConnList 261
lsxECLConnMgr 265
lsxECLField 268
lsxECLFieldList 275
lsxECLOIA 279
lsxECLPS 288
lsxECLSession 298
lsxECLWinMetrics 304
lsxECLXfer 313
lsxECLConnection 255
methods 259
StartCommunications 259
StopCommunications 259
properties 255
APIEnabled 257
CodePage 257
CommStarted 257
ConnHandle 256
ConnName 256
ConnType 256
Ready 258
Started 257
usage notes 255
lsxECLConnection Methods 259
lsxECLConnList 261
methods 263
FindConnectionByHandle 263
FindConnectionByName 264
Refresh 263
properties 261
Count 261
usage notes 261
lsxECLConnList Methods 263
lsxECLConnMgr 265
methods 266
StartConnection 266
StopConnection 267
properties 265
lsxECLConnList 265
lsxECLConnMgr Methods 266
lsxECLField 268
methods 273
GetScreen 273
SetText 273
properties 268
Display 271
EndCol 269
EndRow 269
HighIntensity 271
Length 269
Modified 270
Numeric 270
PenDetectable 271
Protected 270
StartCol 269
lsxECLField *(continued)*
properties *(continued)*
StartRow 268
lsxECLField Methods 273
lsxECLFieldList 275
methods 276
FindFieldByRowCol 276
Refresh 276
properties 275
Count 275
lsxECLFieldList Methods 276
lsxECLOIA 279
methods 287
properties 279
Alphanumeric 280
APIEnabled 285
APL 280
CapsLock 282
CodePage 284
CommErrorReminder 282
CommStarted 285
ConnHandle 284
ConnName 283
ConnType 284
DBCS 281
Hiragana 281
InputInhibited 283
InserMode 282
Katakana 281
MessageWaiting 283
NumLock 282
Ready 286
Started 285
UpperShift 281
usage notes 279
lsxECLOIA Methods 287
lsxECLPS 288
description 253
methods 293
GetScreen 295
GetScreenRect 297
SearchText 294
SendKeys 293
SetCursorPos 293
SetScreen 296
properties 289
APIEnabled 292
CodePage 291
CommStarted 292
ConnHandle 291
ConnName 290
ConnType 291
CursorPosCol 290
CursorPosRow 290
NumCols 289
NumRows 289
Ready 292

lsxECLPS *(continued)*
properties *(continued)*
Started 292
usage notes 288
lsxECLPS Methods 293
lsxECLSession 298
methods 303
properties 298
APIEnabled 301
CodePage 300
CommStarted 301
ConnHandle 299
ConnName 299
ConnType 299
lsxECLOIA 302
lsxECLPS 301
lsxECLWinMetrics 302
lsxECLXfer 302
Ready 301
Started 300
usage notes 298
lsxECLSession Methods 303
lsxECLWinMetrics 304
methods 311
GetWindowRect 311
SetWindowRect 311
properties 305
APIEnabled 310
CodePage 309
CommConnected 309
ConnHandle 308
ConnName 308
ConnType 308
Height 306
IsActive 307
Maximized 307
Minimized 307
Ready 310
Restored 307
Started 309
Visible 306
Width 306
WindowTitle 305
Xpos 305
Ypos 306
usage notes 304
lsxECLXfer 313
methods 317
ReceiveFile 318
SendFile 317
properties 314
APIEnabled 316
CodePage 315
CommStarted 316
ConnHandle 314
ConnName 314
ConnType 314
lsxECLXfer *(continued)*
properties *(continued)*
Ready 316
Started 315
usage notes 313
lsxECLXfer Methods 317

M

Maximized 234, 307
MessageWaiting 199, 283
Migrating from EHLLAPI 6
Events 9
Execution/Language Interface 6
Features 6
Presentation Space Models 8
PS Connect/Disconnect, Multithreading 9
SendKey Interface 9
Session IDs 7
Minimized 234, 307
mnemonic 319
Modified 187, 270

N

Name 174, 200, 209, 222, 235, 243, 256, 283, 290, 299, 308, 314
NotifyError 57, 89, 136, 141
NotifyEvent 57, 88, 136, 141
NotifyStop 58, 89, 137, 142
NumCols 208, 289
Numeric 188, 198, 270, 282
NumRows 208, 289

O

objects, automation 171
autECLConnList 173
autECLConnMgr 181
autECLFieldList 184
autECLOIA 195
autECLPS 207
autECLSession 221
autECLWinMetrics 230
autECLXfer 243
description 171
objects, C++ 11
description 11
ECLBase 17
ECLConnection 23
ECLConnList 37
ECLConnMgr 45
ECLConnNotify 53
ECLErr 59
ECLField 63
ECLFieldList 77

objects, C++ *(continued)*
ECLKeyNotify 84
ECLOIA 90
ECLPS 101
ECLSession 125
ECLStartNotify 133
ECLUpdateNotify 138
ECLWinMetrics 143
ECLXfer 164
objects, LotusScript 253
lsxECLConnection 255
lsxECLConnList 261
lsxECLConnMgr 265
lsxECLField 268
lsxECLFieldList 275
lsxECLOIA 279
lsxECLPS 288
lsxECLSession 298
lsxECLWinMetrics 304
lsxECLXfer 313

P

PenDetectable 189, 271
Protected 188, 270

R

Ready 176, 202, 212, 224, 238, 246, 258, 286, 292, 301, 310, 316
ReceiveFile 168, 249, 318
Refresh 43, 78, 177, 191, 263, 276
RegisterCommEvent 35
RegisterKeyEvent 121
RegisterStartEvent 50
RegisterUpdateEvent 131
Restored 235, 307

S

SearchText 111, 215, 294
SendFile 166, 248, 317
SendKeys 110, 215, 293
Sendkeys mnemonic keywords 319
SetActive 158
SetConnectionByHandle 204, 213, 227, 247
SetConnectionByName 204, 213, 227, 239, 247
SetCursorPos 109, 214, 293
SetHeight 154
SetMaximized 161
SetMinimized 160
SetRestored 162
SetText 72, 117, 194, 217, 273, 296
SetVisible 157
SetWidth 152
SetWindowRect 156, 241, 311
SetWindowTitle 147
SetXpos 148
SetYpos 150
Start Connection 48
StartCol 185, 269
StartCommunication 33, 179, 205, 219, 228, 241, 250, 259
StartConnection 182, 266
Started 175, 201, 211, 223, 237, 245, 257, 285, 292, 300, 309, 315
StartRow 185, 268
StopCommunication 34, 179, 206, 219, 229, 242, 251, 259
StopConnection 49, 183, 267
sxECLWinMetrics Methods 311

T

TextPlane 323

U

UnregisterCommEvent 35
UnregisterKeyEvent 123
UnregisterStartEvent 51
UnregisterUpdateEvent 132
UpperShift 197, 281
Usage Notes 37, 54, 85, 90, 101, 125, 134, 139, 143, 164, 231

V

Visible 233, 306

W

Width 232, 306
WindowTitle 231, 305

X

Xpos 231, 305

Y

Ypos 231, 306

Communicating Your Comments to IBM

eNetwork Personal Communications
Version 4.2
for Windows 95 and Windows NT
Host Access Class Library

Publication No. SC31-8527-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

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

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

If you prefer to send comments by mail, use the RCF at the back of this book.

If you prefer to send comments by FAX, use this number:

United States and Canada: **1-800-227-5088**

If you prefer to send comments electronically, use this network ID:

- IBM Mail Exchange: **USIB2HPD at IBMMAIL**
- IBMLink: **CIBMORCF at RALVM13**
- Internet: **USIB2HPD@VNET.IBM.COM**

Make sure to include the following in your note:

Title and publication number of this book
Page number or topic to which your comment applies.

Help us help you!

eNetwork Personal Communications
Version 4.2
for Windows 95 and Windows NT
Host Access Class Library

Publication No. SC31-8527-00

If your concern is service related, you can reach Service at 1-800-992-4777 in the United States. Outside the United States, please check your phone listing for the IBM Service Center nearest you.

We hope you find this publication useful, readable and technically accurate, but only you can tell us! Please take a few minutes to let us know what you think by completing this form.

Overall, how satisfied are you with the information in this book?	Satisfied	Dissatisfied

How satisfied are you that the information in this book is:	Satisfied	Dissatisfied
Accurate		
Complete		
Easy to find		
Easy to understand		
Well organized		
Applicable to your task		

Specific Comments or Problems:

Please tell us how we can improve this book:

Thank you for your response. When you send information to IBM, you grant IBM the right to use or distribute the information without incurring any obligation to you. You of course retain the right to use the information in any way you choose.

Your Internet Address: _____

Name

Address

Company or Organization

Phone No.

Help us help you!
SC31-8527-00

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

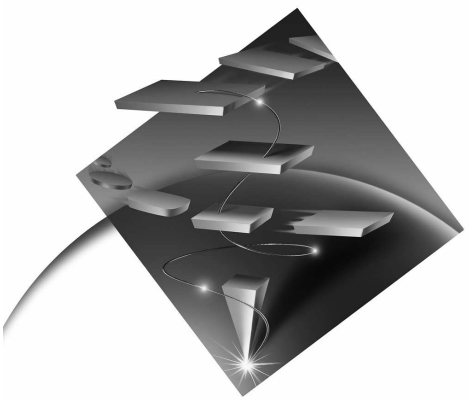
Information Development
Department CGMD
International Business Machines Corporation
PO BOX 12195
RESEARCH TRIANGLE PARK NC 27709-9990

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

SC31-8527-00

Cut or Fold
Along Line

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



SC31-8527-00

Spine information:

eNetwork Personal Communications
Version 4.2
for Windows 95 and Windows NT

Host Access Class Library