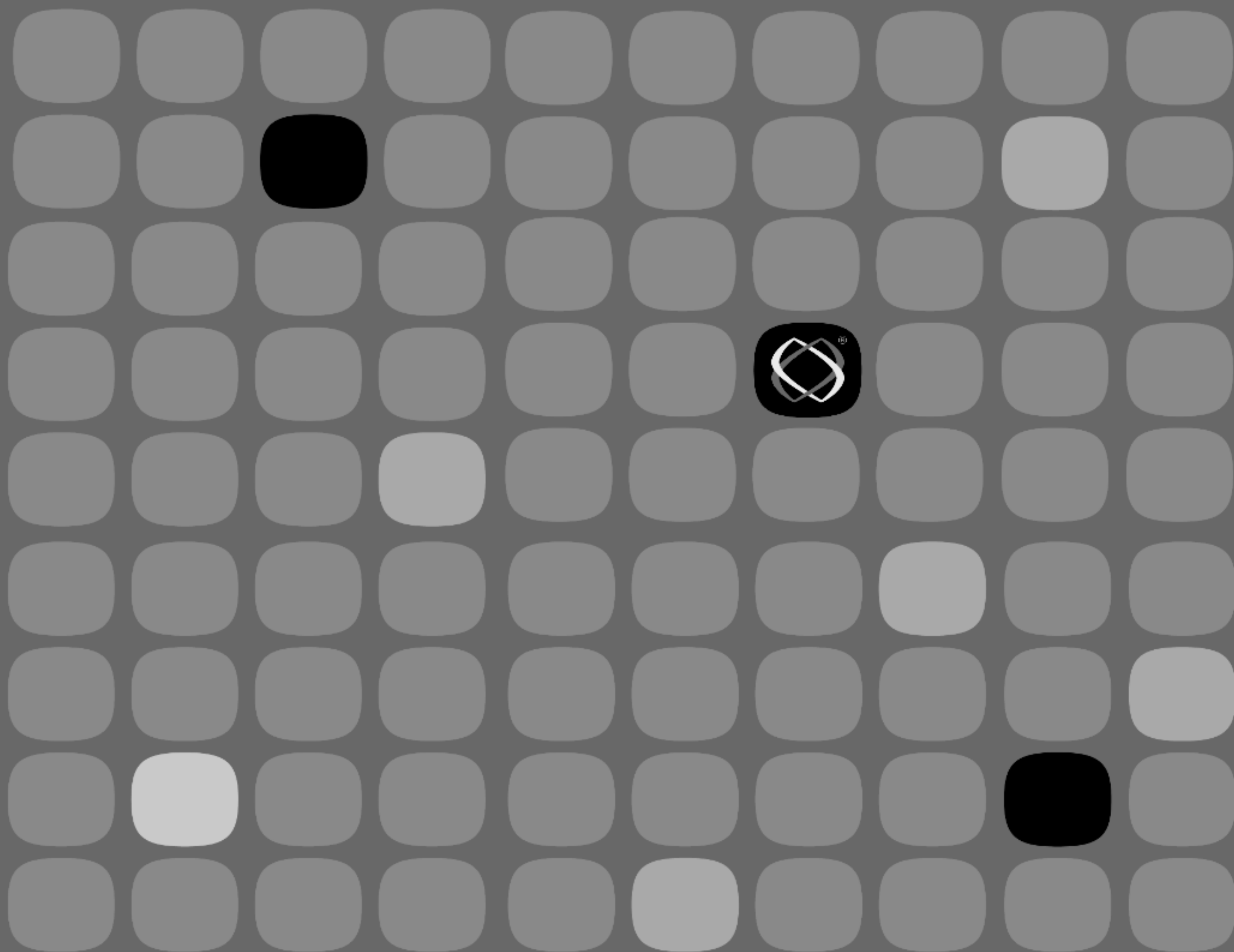


Basic Developer's Guide

UNISYS

INFOCONNECT®

Development Kit

P/N UD 028155

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of a duly executed agreement to purchase or lease equipment or to license software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

© 1993 Unisys Corporation. All rights reserved.

RESTRICTED RIGHTS LEGEND
Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227–7013 and FARS 52.227–14 for commercial computer software.

Attachmate and the Attachmate logo are registered trademarks of Attachmate Corporation in the United States and other countries. INFOConnect is a trademark and Unisys is a registered trademark of Unisys Corporation.

All other trademarks and registered trademarks are property of their respective owners.

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of a duly executed agreement to purchase or lease equipment or to license software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED RIGHTS LEGEND. Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227–7013 and FARS 52.227–14 for commercial computer software.

Correspondence regarding this publication may be forwarded using the Documentation Questionnaire supplied with this document, or may be addressed directly to Unisys Corporation, INFOConnect Development Group, Malvern Development Center, 2450 Swedesford Road, Room B101, Paoli, Pennsylvania, 19301.

Borland products are trademarks or registered trademarks of Borland International, Inc.

IBM is a registered trademark of International Business Machines Corporation.

Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation.

Novell is a registered trademark of Novell, Inc.

Unisys and INFOConnect are trademarks of Unisys Corporation.

UNIX AND UNIX System V are registered trademarks of UNIX System Laboratories.

XVT is a trademark of XVT Software Inc.

Contents

About this Guide xvii

Section 1. Installation

Windows Platform 1–1
- System Requirements 1–1
- Package Contents 1–3
  - Basic IDK Package 1–3
  - Basic IDK Samples Package 1–9
- Installing the IDK 1–17
- Environment Variables 1–17
- System Verification Checklist for Windows 1–19

Preparing XVT for Use with INFOConnect 1–22

Section 2. An Introduction to INFOConnect Connectivity Services

Architecture Diagram 2–5

Terminology 2–6

Accessory API Functions 2–15

Accessory API Events 2–18

Section 3. Writing INFOConnect/Windows Applications

Basic Procedures for Windows Applications 3–2
- Initializing INFOConnect Connectivity Services 3–2
- Opening a Session and Allocating Buffers 3–5
- Transmitting a Buffer 3–8
- Receiving a Buffer 3–12
- Using Datacomm Buffers 3–15
- Basic Error Handling 3–17
- Basic Status Handling 3–22
- Closing a Session 3–26
- Terminating your Application 3–28

Advanced Procedures for Windows Applications 3–32
Canceling Pending Requests 3–32
Handling Data Communications Errors 3–33
Advanced Status and Error Handling 3–36
Encoding and Decoding 3–38
Data Compression and Error Detection 3–38
Running with Old Versions of INFOConnect 3–39

Procedures for INFOConnect Accessories 3–41
Calling INFOConnect Accessories 3–41
Making your Application an INFOConnect Accessory 3–42

Compiling 3–44

Resource Files 3–46

Linking 3–51

IcWinApp - a Sample Windows Application 3–53

CoupleW - a Windows Application that Connects Two INFOConnect Sessions 3–76

Section 4. Writing INFOConnect/XVT Applications

Basic procedures for XVT Applications 4–2
Initializing INFOConnect Connectivity Services 4–2
Opening a Session and Allocating Buffers 4–3
Transmitting a Buffer 4–6
Receiving a Buffer 4–8
Using Datacomm Buffers 4–10
Basic Error Handling 4–12
Basic Status Handling 4–16
Closing a Session 4–19
Terminating your Application 4–20

Advanced Procedures for XVT Applications 4–24
Canceling Pending Requests 4–24
Handling Data Communications Errors 4–25
Advanced Status and Error Handling 4–26
Using Event Hooks with XVT 3.0 4–29
Using Keyboard and Event Hooks with XVT2.0 ... 4–30
Encoding and Decoding 4–31
Data Compression and Error Detection 4–32
Running with Old Versions of INFOConnect 4–32

Procedures for INFOConnect Accessories 4–33
Calling INFOConnect Accessories 4–34
Making your Application an INFOConnect Accessory 4–34

Compiling 4–37
Windows Platform 4–37
Compiler Errors 4–39

Resource Files 4–39

Linking 4–40
Windows Platform 4–40

IcXvtApp - A sample XVT application 4–44

Couple - An XVT Application that Connects Two INFOConnect Sessions 4–46

IcOpenAc - An XVT Application that Opens an INFOConnect Accessory 4–48

Section 5. Writing INFOConnect/DosLink Applications

Basic Procedures for DosLink Applications 5–3
Initializing INFOConnect Connectivity Services 5–3
Opening a Session 5–4
Transmitting a Buffer 5–8
Receiving a Buffer 5–11
Allocating and Using Datacomm Buffers 5–13
Error Handling 5–15
Closing a Session 5–17

Advanced Procedures for DosLink Applications 5–19
Canceling Pending Requests 5–19
Handling Data Communications Errors 5–20
Running with Old Versions of INFOConnect 5–21

A Closer Look at the DosLink Solution 5–23

Compiling and Linking 5–26

IcDosApp - a Sample DosLink Application 5–29

IcBDrive - a Sample DosLink TSR 5–30

Section 6. A Closer Look at the INFOConnect Architecture

Manager Components 6–1

Structure of Service and External Interface Libraries 6–3

ICS Control Flow 6–4
Processing an Open Session Request 6–4
Processing a Transmit (or Receive) Request 6–8
Processing Status and Error Events 6–11

Section 7. Writing INFOConnect Libraries for Windows 3.x

Design Issues 7–2
Choosing Between Accessories, Services and Interfaces 7–2
Session Attributes 7–3
Configuration Management 7–4
Table Design 7–6
Table Description 7–8
Table Processing 7–12
Session and Channel Runtime-record Layout 7–13
Error Management 7–15
Status Management 7–16
Using IC_STATUS_BUFFER Extended Status 7–17
Version Control 7–19
Filtering Service Libraries 7–24
Windows 3.x Issues 7–25

Writing the Required IcLib Functions 7–26
IcLibInstall 7–26
IcLibTerminate 7–28
IcLibUpdateConfig 7–29
IcLibVerifyConfig 7–31
IcLibPrintConfig 7–33
IcLibOpenChannel 7–34
IcLibCloseChannel 7–35
IcLibOpenSession 7–36
IcLibCloseSession 7–37
IcLibXmt / IcLibRcv 7–38
IcLibLcl 7–40
IcLibSetResult 7–41
IcLibEvent 7–42
IcLibGetSessionInfo 7–46

IcLibGetString 7–48
IcLibIdentifySession 7–51
Windows DLL requirements 7–52

Other Procedures and Guidelines 7–53
Session and Channel Aliasing 7–53
Generating Errors from your Library 7–57
Requesting Session Termination 7–60
On-line Help 7–61
Modifying Global Variables 7–63
Communicating with Applications Using Status Messages 7–63
System Timers 7–68
Tracing 7–70
Running with Multiple Versions of INFOConnect 7–71
Library Checklist 7–72

Compiling 7–73

Resource Files 7–75

INFOConnect Header Files 7–84

Linking 7–86

PS2TTY - A Sample Service Library 7–89

CoupleS - A Sample Service Library 7–90

Service - A Generic Service Library 7–92

Reflect - A Sample External Interface 7–93

IcStack2- A Sample External Interface 7–94

Intrface - A Generic External Interface 7–95

Section 8. Debugging

General Debugging Procedures 8–1
Tracing INFOConnect Datacomm Activity 8–1
Adding Trace Information to the Trace Log 8–3
Using the Diagnostic Library 8–5
Using the Assert Macro in Applications 8–5
Using the INFOConnect -d Debug Option 8–6

Windows 3.x Debugging Procedures 8–6
Windows 3.1 Issues 8–6
Source Level Debugging 8–6
Using Debug version of Windows 8–7

Common Coding Mistakes 8–7

Section 9. Packaging INFOConnect Components

Overview 9–1
Terminology 9–2

Writing a .INF Script File 9–6

Creating the Package Diskette(s) 9–9

A Closer Look at Processing Flow 9–10
Installation Flow 9–10
Deinstallation Flow 9–15

INF SYNTAX 9–16
Data Section 9–18
Dialog Section 9–19
Disks Section 9–20
Package Section 9–21
Progman.Groups
SectionPublish.Progman.Groups
SectionStandAlone.Progman.Groups Sect...... 9–23
Needed.Space Section 9–26
App.Copy.AppStuff
SectionApp.Copy.NoRemove
SectionApp.Copy.Publish
SectionApp.Copy.Subscribe
SectionApp.Copy.StandAlone Section 9–27

.INF Examples 9–34
IcWinApp INF 9–34
Reflect INF 9–36

Section 10. Converting from Previous Releases

Converting From Release 2.0 to 3.0 10–1
3.0 Features 10–1
Architecture Diagram 10–1
Manager Components 10–3
Application Interface Library 10–3
Interprocess Interface Library 10–4

Stack Interface Library 10–4
Switching Library 10–4
Hook Library 10–4
Shell 10–4
Configuration API 10–5
Configurator 10–5
ICS Version Numbers 10–5
Component Numbers 10–7
Trace Functions 10–7
IC_STATUS_BUFFER 10–8
Changes Affecting Applications and Libraries 10–8
INFOConnect Version Identification 10–8
Component Numbers 10–12
Defining Component Numbers 10–12
Managing Branded Component Numbers 10–13
Changes Affecting Applications 10–13
CommMgr.lib Removed 10–14
New XVT Link Libraries (.LIB) 10–14
ICXVTWIN Tag 10–15
New IC_STATUS_COMMMGR Status 10–15
New IC_STATUS_TRANS Statuses 10–15
IcRegisterMsgSession 10–15
Running with Old Versions of INFOConnect 10–16
Changes Affecting Libraries 10–16
Library API Changes 10–16
Changes for IcLib Functions 10–17
Changes for Service Libraries 10–22
Running with Multiple Versions of INFOConnect 10–22

Converting From Release 1.0 to 2.0 10–23
2.0 Features 10–23
Architecture Diagram 10–23
Terminology 10–25
INFOConnect Installation Manager 10–27
Accessory IDs 10–27
Accessory Window States 10–27
DosLink 10–28
Library Configuration 10–28
Library API Control Flow Diagrams 10–28
Session and Channel Aliasing 10–28
Tracing INFOConnect Activity 10–28
INFOConn.ini 10–29

Running Old INFOConnect Modifiers and Interfaces 10–29
Multiplexing Libraries 10–29

Changes Affecting Applications 10–29
Running with Old Versions of INFOConnect 10–30
CommMgr.lib Replaced by IcWin.lib 10–30
HANDLE Versus HIC_SESSION 10–30
New INFOConnect Events 10–31
Freeing Buffers Immediately after Session Closure 10–31
New IC_STATUS_COMMMGR Statuses 10–32
IcGetSessionName and IcGetPathName 10–32
IcOpenAccessory and ic_open_accessory 10–34
New Requirements for Accessories 10–35
Changes Affecting Libraries 10–38
COMMMGR.LIB Replaced by ICWIN.LIB 10–38
New Header (.H) Files 10–38
Function Name Changes 10–39
HANDLE Versus HIC_SESSION 10–42
IcLibInstall 10–43
Library Configuration 10–43
Session Establishment 10–43
IcP-ConfigPath Replacements 10–44
Session and Channel Aliases 10–44

Figures

2–1. Application's Perspective of ICS 2–2
2–2. Simplified Transmit Illustration 2–3
2–3. INFOConnect 3.0 Architecture 2–5
2–5. Path 2–11
2–6. Session/Path Relationship 2–12
2–5. Session/Architecture Relationship 2–13
2–7. Path Template 2–14

5–1. The DosLink Solution 5–23

6–1. Manager Components and APIs 6–2
6–2. Processing During Session Establishment 6–4
6–3. Processing During Transmit 6–8
6–4. Status Initiated by the Application 6–11
6–5. Status Initiated by a Library 6–13

10–2. INFOConnect 3.0 Architecture 10–2
10–1. INFOConnect 2.0 Architecture 10–24

Tables

1–1. IDK Files/Purpose 1–3
1–2. Sample Files/Purpose 1–9

6–1. Manager Component, DLL/EXE, and API 6–1

About This Guide

Purpose

This guide explains how to install and use the 3.0 release of the INFOConnect Development Kit . It is intended to help the experienced programmer write INFOConnect applications and components.

Why use the INFOConnect Development Kit?

- To decrease your development time.
- Because INFOConnect has been chosen as the standard workstation interconnect platform for the Unisys Architecture.
- To avoid writing and debugging communications code.
- To increase your application's transport options.

Scope

This guide is relative to release 3.0 of INFOConnect Connectivity Services and the INFOConnect Development Kit. This is a Basic INFOConnect Developer's Kit. It describes how to develop INFOConnect Connectivity Services (ICS) Accessories, additional data filters (Service Libraries) and connection types (External Interface Libraries). It does not provide support for developing INFOConnect Application Interface Libraries, Quick Configuration Libraries, Hook Libraries or for developing with Visual Basic.

Audience

This guide is written for application and system developers who are developing cooperative applications that use INFOConnect Connectivity Services (ICS) for data communications. This manual is also geared towards the developer who wishes to build additional data filters (Service Libraries) and connection types (External Interface Libraries).

The *INFOConnect Development Kit Basic Developer's Guide* does not describe how to develop INFOConnect Application Interface Libraries, Configurators, Shells, Quick Configuration Libraries, or Hook Libraries.

Prerequisites

You should be familiar with the C programming language, event-driven programming, and the appropriate platform for which you are developing. For example, Windows, XVT, and so forth.

How to Use This Guide

Use this guide in conjunction with the *INFOConnect Development Kit Basic Programming Reference Manual*.

It is recommended that all developers read the following sections:

Section 1, "Installation"
Section 2, "An Introduction to INFOConnect Connectivity Services"
Section 8, "Debugging"
Section 9, "Packaging INFOConnect Components"

Native Windows application developers should read:

Section 3, "Writing INFOConnect/Windows Applications"

XVT application developers should read:

Section 4, "Writing INFOConnect/XVT Applications"

DosLink application developers should read:

Section 5, "Writing INFOConnect/DosLink Applications"

Library developers should read:

Section 6, "A Closer Look at the INFOConnect Architecture"
Section 7, "Writing INFOConnect Libraries for Windows 3.x"

Developers upgrading from previous INFOConnect Releases should read:

Section 10, "Converting from Previous Releases"

Organization

This guide consists of the following sections.

Section 1. Installation

This section describes the installation of the INFOConnect Development Kit.

Section 2. An Introduction to INFOConnect Connectivity Services

This section discusses INFOConnect Architecture, basic INFOConnect terminology, and the Accessory Application Programming Interface (AAPI) functions and events.

Section 3. Writing INFOConnect/Windows Applications

This section discusses how to write an INFOConnect application using the facilities of Microsoft Windows.

Section 4. Writing INFOConnect/XVT Applications

This section discusses how to write an INFOConnect application using the XVT presentation toolkit.

Section 5. Writing INFOConnect/DosLink Applications

This section discusses how to write an INFOConnect application that allows DOS applications to make connections to other computers using INFOConnect Connectivity Services.

Section 6. A Closer Look at the INFOConnect Architecture

This section discusses the architecture of INFOConnect in more detail, the structure of Service and External Interface Libraries, and ICS control flow.

Section 7. Writing INFOConnect Libraries for Windows 3.x

This section discusses how to build INFOConnect Service and External Interface Libraries.

Section 8. Debugging

This section discusses procedures for debugging applications you have written for INFOConnect.

Section 9. Packaging INFOConnect Components

This section discusses how to package your INFOConnect components to provide a common look and feel with all INFOConnect packages. It also covers information about setting up your components for easy installation with INFOConnect.

Section 10. Converting from Previous Releases

This section describes the new features of release 3.0 and provides conversion guidelines for upgrading existing INFOConnect components from release 2.0 to 3.0. It also describes the new features of release 2.0 and provides conversion guidelines for upgrading INFOConnect components from release 1.0 to 2.0.

Related Product Information

***INFOConnect™ Developer's Kit Basic Programming Reference Manual* (4173 5390-000)**

Provides detailed information about the programming interfaces, messages and data types for the basic features of INFOConnect Connectivity Services.

***INFOConnect™ Connectivity Services Installation and Configuration Guide* (4240 0119-200)**

Contains information on installing and configuring the INFOConnect runtime product.

Microsoft® Windows™ Software Development Kit Reference

Contains reference material for the Windows SDK.

Microsoft® Windows™ Software Development Kit Guide to Programming

Describes how to use the Windows SDK to develop Windows applications and dynamic link libraries.

XVT™ Programmer's Manual

Contains reference material for the XVT developers kit.

Section 1
Installation

This section covers the installation and preparation of the INFOConnect Development Kit (IDK). Although only the Windows version is currently available, a directory structure is used that will allow additional versions of the IDK to exist on the same machine in the future. After installation, a directory will exist on your hard disk named IDK.

The directory structure used by the IDK is:

```
idk
idk\include
idk\lib
idk\lib\win
idk\run
idk\run\win
idk\win
idk\sample
```

Windows Platform

System Requirements

- Unisys PC or compatible device
  - 80286 or higher processor (80386 recommended)
  - 2 MB memory (4MB recommended)
  - EGA graphics
  - Mouse
  - 40 MB Hard disk (80 MB recommended)
  - 3.5 or 5.25 floppy disk drive

Note: *Be sure to check the hardware and software requirements for your C-language compiler and the Windows 3.x Software Development Kit.*

- MS-DOS 3.3 or higher running MS-Windows 3.x
- Windows 3.x Software Development Kit

- INFOConnect Connectivity Services
  The INFOConnect runtimes are packaged separately from the IDK.
- C language compiler
  The compiler must be compatible with Microsoft Windows. This includes, but is not restricted to:
  - Microsoft C version 5.1 or later
  - Microsoft QuickC version 2.0 or later
  - Borland C++ 2.0 or later

  The IDK sample programs were developed using the Microsoft 7.0 compiler. The IDK does not provide INFOConnect/C++ classes yet.
- XVT/Win Release 2.1 or later (optional)

To build on-line help files, you will need a word processor capable of building Rich Text Format (.RTF) files.

Package Contents

The IDK consists of one installation disk that contains two INFOConnect packages: Basic INFOConnect Development Kit and Basic IDK Samples. Check the IDKReadM.txt file for last minute changes not included in this document.

Basic IDK Package

Table 1-1 contains an alphabetical list of all files in the Basic INFOConnect Development Kit package, the directory in which they are found, and their purpose:

Table 1–1. IDK Files/Purpose

IDK File Name	Directory	Purpose
DcDevice.hic	\idk\include	Generic datacomm device shell header file defining library-specific errors, statuses and configuration tables
DosLink.lib	\idk\lib\win	DosLink API .lib file for linking DosLink applications
Ic.hic	\idk\include	C header file that defines generic component numbers and branded (supplier specific) component supplier numbers
Ic2XvtL.lib	\idk\lib\win	Resolves all references to the 2.0 and 3.0 releases of ICS functions for XVT3.0x (Large memory model). Use this link library for XVT applications that will run with both ICS 2.0 and 3.0.
Ic2XvtM.lib	\idk\lib\win	Resolves all references to the 2.0 and 3.0 releases of ICS functions for XVT 3.0x (Medium memory model). Use this link library for XVT applications that will run with both ICS 2.0 and 3.0.

continued

Table 1–1. IDK Files/Purpose (cont.)

IDK File Name	Directory	Purpose
Ic2Xvt2L.lib	\idk\lib\win	Resolves all references to the 2.0 and 3.0 releases of ICS functions for XVT 2.0 (Large memory model). Use this link library for XVT applications that will run with both ICS 2.0 and 3.0.
Ic2Xvt2M.lib	\idk\lib\win	Resolves all references to the 2.0 and 3.0 releases of ICS functions for XVT 2.0 (Medium memory model). Use this link library for XVT applications that will run with both ICS 2.0 and 3.0.
IcAbtOem.dll	\idk\run\win	INFOConnect About DLL for Original Equipment Manufacturer
IcAComs.hic	\idk\include	Communications Management Systems header file defining library-specific errors, statuses, and configuration tables
IcAssert.h	\idk\include	IDK C header file that provides a version of the Assert macro to be used in Windows DLL libraries. Note: the standard Assert macro cannot be used in Windows DLL libraries
IcConfig.h	\idk\include	C header file for configuration utilities
IcDef.h	\idk\include	C header file included in IcWin.h and IcXvt.h
IcDef.rh	\idk\include	Resource include file that defines the version template for INFOConnect version stamping
IcDict.h	\idk\include	C header file for the data dictionary -- INFOConnect libraries should include this into the resource file
IcDos.h	\idk\include	C header file that contains the function prototypes for the INFOConnect DosLink services available to DOS applications

continued

Table 1–1. IDK Files/Purpose (cont.)

IDK File Name	Directory	Purpose
IcDosLnk.hic	\idk\include	INFOConnect DosLink Access header file defining library-specific errors, statuses and configuration tables
IcError.h	\idk\include	C header file for INFOConnect error information
IcExit.h	\idk\include	C header file for install/exit hook libraries ***Note:*** *IcExit.h will be decommitted in the next release of INFOConnect. The necessary definitions have been moved to IcLib.h. The function prototypes have been moved to IcProto.h.*
IcDosApp.exe	\idk\run\win	IcDosApp executable
IcHLCNTS.hic	\idk\include	INFOConnect HLCNTS protocol header file defining library-specific errors, statuses and configuration tables
IcInstal.h	\idk\include	C header file for install/exit hook libraries ***Note:*** *IcInstal.h will be decommitted in the next release of INFOConnect. The necessary definitions have been moved to IcLib.h. The function prototypes have been moved to IcProto.h.*
IcLCW.hic	\idk\include	INFOConnect LCW service library header file defining library-specific errors, statuses, and configuration tables
IcLib.h	\idk\include	C header file to be included into service, external interface, application interface, and interprocess interface libraries
IcLocal.hic	\idk\include	Local protocol header file defining library-specific errors, statuses and configuration tables
IcMem.h	\idk\include	C header file for the memory management API
IcMgr.hic	\idk\include	IcMgr header file defining library-specific errors, statuses and configuration tables

Table 1–1. IDK Files/Purpose (cont.)

IDK File Name	Directory	Purpose
IcMgrCfg.h	\idk\include	C header file for the INFOConnect configuration API
IcMgrIns.hic	\idk\include	IcMgrIns header file defining library-specific errors, statuses and configuration tables
IcMon.hic	\idk\include	INFOConnect Session Monitor header file defining library-specific errors, statuses and configuration tables
IcNBios.hic	\idk\include	INFOConnect NetBios protocol header file defining library-specific errors, statuses and configuration tables
IcNotWin.h	\idk\include	C header file for DOS applications to define 'standard' types normally defined in windows.h
IcOpenAc.exe	\idk\run\win	IcOpenAc executable
IcProto.h	\idk\include	C header file to prototype the procedures in ICS library components
IcSample.hic	\idk\include	C header file that defines component numbers for the IDK samples
IcShell.h	\idk\include	C header file for Shell utilities
IcStatus.h	\idk\include	C header file for INFOConnect status definitions
IcTCP.hic	\idk\include	INFOConnect TCP header file defining library-specific errors, statuses and configuration tables
IcTelnet.hic	\idk\include	INFOConnect Telnet service library header file defining library-specific errors, statuses and configuration tables
IcTrace.hic	\idk\include	INFOConnect Trace header file defining library-specific errors, statuses and configuration tables
IcTraceL.hic	\idk\include	INFOConnect Trace Log header file defining library-specific errors, statuses and configuration tables

continued

Table 1–1. IDK Files/Purpose (cont.)

IDK File Name	Directory	Purpose
IcTTY.hic	\idk\include	INFOConnect TTY ASYNC protocol header file defining library-specific errors, statuses and configuration tables
IcUISMal.hic	\idk\include	C header file that defines component numbers for the Malvern components
IcUtil.h	\idk\include	C header file for routines that are useful within the INFOConnect environment, but the routines themselves are not directly associated with the INFOConnect Manager
IcWin.h	\idk\include	C header file to be included into applications
IcWin.lib	\idk\lib\win	Resolves all the references to the INFOConnect Connectivity Services functions
IcWin20.lib	\idk\lib\win	Resolves all references to the 2.0 and 3.0 releases of ICS functions. Use this link library for applications and libraries that will run with both ICS 2.0 and 3.0.
IcXEHook.c	\idk\win	IcXEHook -- C source base file for terminals providing an XVT interface
IcXEvent.obj	\idk\lib\win	IcXEvent.obj (INFOConnect event_hook routine) must always be included when linking XVT applications (Medium memory model)
IcXEvntL.obj	\idk\lib\win	IcXEvntL.obj (INFOConnect event_hook routine) must always be included when linking XVT applications (Large memory model)
IcXKey.obj	\idk\lib\win	IcXKey.obj is a default key_hook module provided by INFOConnect. Replace it if you are providing your own key_hook routine (Medium memory model)
IcXKeyL.obj	\idk\lib\win	IcXKeyL.obj is a default key_hook module provided by INFOConnect. Replace it if you are providing your own key_hook routine (Large memory model)

continued

Table 1–1. IDK Files/Purpose (cont.)

IDK File Name	Directory	Purpose
IcXMem.h	\idk\include	C header file for INFOConnect memory management using XVT
IcXNS.hic	\idk\include	INFOConnect XNS (IPX/SPX interface) external interface library header file defining library-specific errors, statuses and configuration tables
IcXvt2L.lib	\idk\lib\win	Resolves all references to the 3.0 Release of ICS functions for XVT 2.0 (Large memory model)
IcXvt2M.lib	\idk\lib\win	Resolves all references to the 3.0 Release of ICS functions for XVT 2.0 (Medium memory model)
IcXvt.h	\idk\include	C header file to be included in XVT applications
IcXvtL.lib	\idk\lib\win	Resolves all references to the 3.0 Release of ICS functions for XVT 3.0x (Large memory model)
IcXvtM.lib	\idk\lib\win	Resolves all references to the 3.0 Release of ICS functions for XVT 3.0x (Medium memory model)
IcXvtMod.exe	\idk	Utility for XVT header file conversion
IDKReadM.txt	\idk	Last minute documentation changes
INFOConn.ico	\idk\win	INFOConnect icon file
Sys.ico	\idk\win	Icon file containing the letters SYS, there is also a UNI icon file (UNISYS)
Uni.ico	\idk\win	Icon file containing the letters UNI, there is also a SYS icon file (UNISYS)

Basic IDK Samples Package

Table 1-2 contains an alphabetical list of files that comprise the Basic IDK Samples package, the directory in which they are found, and their purpose:

Table 1–2. Sample Files/Purpose

Sample File Name	Directory	Purpose
Couple.c	\idk\sample	Couple -- C-language source for an XVT sample application that connects two INFOConnect sessions
Couple.def	\idk\sample	Couple module-definition file used to link
Couple.h	\idk\sample	Couple header file
Couple.ico	\idk\sample	Couple icon file
Couple.url	\idk\sample	Couple resource file
CoupleS.c	\idk\sample	CoupleS -- C-language source for a sample service library that intercepts, encodes, and transmits statuses across a connection to a partner CoupleS service library
CoupleS.def	\idk\sample	CoupleS module-definition file used to link
CoupleS.dlg	\idk\sample	CoupleS configuration dialog source statements
CoupleS.doc	\idk\sample	CoupleS on-line help text in Microsoft Word format
CoupleS.h	\idk\sample	CoupleS header file
CoupleS.hh	\idk\sample	CoupleS header file for on-line help
CoupleS.hic	\idk\sample	CoupleS header file defining library-specific errors, statuses and configuration tables

continued

Table 1–1. Sample Files/Purpose (cont.)

Sample File Name	Directory	Purpose
CoupleS.hlp	\idk\sample	CoupleS help file
CoupleS.hpj	\idk\sample	CoupleS help project file for on-line help
CoupleS.rc	\idk\sample	CoupleS resource file
CoupleS.rtf	\idk\sample	CoupleS on-line help text in Rich Text Format
CoupleW.c	\idk\sample	CoupleW -- C-language source for a Windows sample application that connects two INFOConnect sessions
CoupleW.def	\idk\sample	CoupleW module-definition file used to link
CoupleW.h	\idk\sample	CoupleW header file
CoupleW.ico	\idk\sample	CoupleW icon file
CoupleW.rc	\idk\sample	CoupleW resource file
IcBDrive.c	\idk\sample	IcBDrive -- C-language source for a DosLink sample application that maps a subset of the BDrive API to INFOConnect
IcBDrive.msg	\idk\sample	IcBDrive message source file
IcDosApp.c	\idk\sample	IcDosApp -- C-language source for a DosLink sample application that uses INFOConnect services
IcOpenAc.c	\idk\sample	IcOpenAc -- C-language source code for an XVT application that opens a INFOConnect accessory
IcOpenAc.def	\idk\sample	IcOpenAc module-definition file used to link
IcOpenAc.h	\idk\sample	IcOpenAc header file
IcOpenAc.ico	\idk\sample	IcOpenAc icon file
IcOpenAc.url	\idk\sample	IcOpenAc resource file

continued

Table 1–1. Sample Files/Purpose (cont.)

Sample File Name	Directory	Purpose
IcStack2.c	\idk\sample	IcStack2 -- C-language source code for Stack Library that stacks one INFOConnect path on top of another.
IcStack2.def	\idk\sample	IcStack2 module-definition file used to link
IcStack2.dlg	\idk\sample	IcStack2 configuration dialog source statements
IcStack2.doc	\idk\sample	IcStack2 on-line help text in Microsoft Word format
IcStack2.h	\idk\sample	IcStack2 header file
IcStack2.hh	\idk\sample	IcStack2 header file for on-line help
IcStack2.hic	\idk\sample	IcStack2 header file defining library-specific errors, statuses and configuration tables
IcStack2.hlp	\idk\sample	IcStack2 help file
IcStack2.hpj	\idk\sample	IcStack2 help project file for on-line help
IcStack2.rc	\idk\sample	IcStack2 resource file
IcStack2.rtf	\idk\sample	IcStack2 on-line help text in Rich Text Format
IcWinApp.c	\idk\sample	IcWinApp -- C source file for a windows application that allows a user to enter messages to be sent across the communications path using dialog boxes
IcWinApp.def	\idk\sample	IcWinApp module-definition file used to link
IcWinApp.h	\idk\sample	IcWinApp header file
IcWinApp.ico	\idk\sample	IcWinApp icon file

continued

Table 1–1. Sample Files/Purpose (cont.)

Sample File Name	Directory	Purpose
IcWinApp.inf	\idk\sample	IcWinApp installation script file
IcWinApp.rc	\idk\sample	IcWinApp resource file
IcWinApp.txt	\idk\sample	IcWinApp README .txt file for .INF file
IcXvtAp2.c	\idk\sample	IcXvtAp2-- C-language source file for a sample XVT 2.0 application that allows a user to enter messages to be sent across the communications path using dialog boxes
IcXvtAp2.def	\idk\sample	IcXvtAp2 module-definition file used to link
IcXvtAp2.h	\idk\sample	IcXvtAp2 header file
IcXvtAp2.ico	\idk\sample	IcXvtAp2 icon file
IcXvtAp2.url	\idk\sample	IcXvtAp2 resource file
IcXvtApp.c	\idk\sample	IcXvtApp -- C-language source file for a sample XVT 3.0 application that allows a user to enter messages to be sent across the communications path using dialog boxes
IcXvtApp.def	\idk\sample	IcXvtApp module-definition file used to link
IcXvtApp.h	\idk\sample	IcXvtApp header file
IcXvtApp.ico	\idk\sample	IcXvtApp icon file
IcXvtApp.url	\idk\sample	IcXvtApp resource file
Intrface.c	\idk\sample	Intrface -- C-language source which can be used as a starting point for external interface library development

continued

Table 1–1. Sample Files/Purpose (cont.)

Sample File Name	Directory	Purpose
Intrface.def	\idk\sample	Intrface module-definition file used to link
Intrface.dlg	\idk\sample	Intrface configuration dialog source statements
Intrface.doc	\idk\sample	Intrface on-line help text in Microsoft Word format
Intrface.h	\idk\sample	Intrface header file
Intrface.hh	\idk\sample	Intrface header file for on-line help
Intrface.hic	\idk\sample	Intrface header file defining library-specific errors, statuses and configuration tables
Intrface.hlp	\idk\sample	Intrface help file
Intrface.hpj	\idk\sample	Intrface help project file for on-line help
Intrface.rc	\idk\sample	Intrface resource file
Intrface.rtf	\idk\sample	Intrface on-line help text in Rich Text Format
Makedos	\idk\sample	Makefile for compiling ICS DosLink applications using Microsoft C 7.0 and Visual C++ 1.0
Makedos.bcc	\idk\sample	Makefile for compiling ICS DosLink applications using Borland C++ 3.1
Makefile	\idk\sample	Makefile for compiling ICS applications using Microsoft C 7.0 and Visual C++ 1.0
Makefile.bcc	\idk\sample	Makefile for compiling ICS applications using Borland C++ 3.1

continued

Table 1–1. Sample Files/Purpose (cont.)

Sample File Name	Directory	Purpose
Makelib	\idk\sample	Makefile for compiling ICS library components using Microsoft C 7.0 and Visual C++ 1.0
Makelib.bcc	\idk\sample	Makefile for compiling ICS library components using Borland C++ 3.1
PS2TTY.c	\idk\sample	PS2TTY -- C-language source for a sample service library that scans receive buffers and removes any poll/select escape-sequences before passing the buffer on to the application
PS2TTY.def	\idk\sample	PS2TTY module-definition file used to link
PS2TTY.dlg	\idk\sample	PS2TTY configuration dialog source statements
PS2TTY.doc	\idk\sample	PS2TTY on-line help text in Microsoft Word format
PS2TTY.h	\idk\sample	PS2TTY header file
PS2TTY.hh	\idk\sample	PS2TTY header file for on-line help
PS2TTY.hic	\idk\sample	PS2TTY header file defining library-specific errors, statuses and configuration tables
PS2TTY.hlp	\idk\sample	PS2TTY help file
PS2TTY.hpj	\idk\sample	PS2TTY help project file for on-line help
PS2TTY.rc	\idk\sample	PS2TTY resource file

continued

Table 1–1. Sample Files/Purpose (cont.)

Sample File Name	Directory	Purpose
PS2TTY.rtf	\idk\sample	PS2TTY on-line help text in Rich Text Format
Reflect.c	\idk\sample	Reflect -- C-language source for a sample external interface library that stores any transmitted messages from the application and returns them to the application when a receive is issued
Reflect.def	\idk\sample	Reflect module-definition file used to link
Reflect.dlg	\idk\sample	Reflect configuration dialog source statements
Reflect.doc	\idk\sample	Reflect on-line help text in Microsoft Word format
Reflect.h	\idk\sample	Reflect header file
Reflect.hh	\idk\sample	Reflect header file for on-line help
Reflect.hic	\idk\sample	Reflect header file defining library-specific errors, statuses and configuration tables
Reflect.hlp	\idk\sample	Reflect help file
Reflect.hpj	\idk\sample	Reflect help project file for on-line help
Reflect.inf	\idk\sample	Reflect installation script file
Reflect.rc	\idk\sample	Reflect resource file
Reflect.rtf	\idk\sample	Reflect on-line help text in Rich Text Format
Service.c	\idk\sample	Service -- C-language source which can be used as a starting point for service library development

continued

Table 1–1. Sample Files/Purpose (cont.)

Sample File Name	Directory	Purpose
Service.def	\idk\sample	Service module-definition file used to link
Service.dlg	\idk\sample	Service configuration dialog source statements
Service.doc	\idk\sample	Service on-line help text in Microsoft Word format
Service.h	\idk\sample	Service header file
Service.hh	\idk\sample	Service header file for on-line help
Service.hic	\idk\sample	Service header file defining library-specific errors, statuses and configuration tables
Service.hlp	\idk\sample	Service help file
Service.hpj	\idk\sample	Service help project file for on-line help
Service.rc	\idk\sample	Service resource file
Service.rtf	\idk\sample	Service on-line help text in Rich Text Format

Installing the IDK

The Basic INFOConnect Development Kit consists of two packages, ICW10-IDK and ICW10-IDK-SAMPLES. The first package contains the Basic INFOConnect Development Kit and the second package contains the Basic IDK Samples. Both packages are on the same distribution media (disk or floppy).

Insert the disk containing the Basic INFOConnect Development Kit. From Program Manager, run a:\install. You are given an opportunity to name the destination directory. The default directory is c:\idk. After the IDK files are installed, installation of the Basic IDK samples (optional) is next. The default directory for the Basic IDK samples is c:\idk\sample.

If you don't install the Basic IDK samples during the Basic IDK installation, they can be installed later. Insert the Basic IDK Samples Disk into drive A. From Program Manager, run a:\install. When prompted for the package root directory or the package scriptfile name, type in a:\icsample.inf. You are given an opportunity to name the destination directory. The default directory is c:\idk\sample.

Note: *If you choose a destination directory other than c:\idk, the sample program make files in the sample directories will need to be updated to reflect the destination directory chosen.*

Environment Variables

INCLUDE and LIB

The INCLUDE and LIB environment variables must be updated for those compilers (e.g. Microsoft) that use them. If you've installed the C compiler or XVT, these variables should be defined in AutoExec.bat. Modify them to include the IDK directories as shown below:

```
set include = c:\idk\include;%include%
set lib = c:\idk\lib\win;%lib%
```

Microsoft compatible and Borland compatible makefiles

To build the sample programs, the IDK provides different makefiles for different C compilers. Several environment variables are used with the Microsoft and Borland makefiles to locate the different directories needed during INFOConnect development. Recognized environment variables and their default values are listed below. If the default value for a variable is correct for your machine, you don't have to set that variable in your AutoExec.bat. In other words, if you installed the Windows SDK into c:\windev, then you don't need to do the SET WINLIB= statement because the IDK makefiles will correctly assume that the Windows libraries are in c:\windev\lib. The WINSDKVER environment variable specifies the level of the Windows SDK installed on your machine. The 3.1 SDK is the default. Be sure to rerun AutoExec.bat or reboot your computer after you update any of these environment variables:

```
set icwin=c:\idk\win

set iclib=c:\idk\lib\win

set xvtver=0x0301                 ; XVT 3.01
           0x0302                 ; XVT 3.02

set xvtdir=c:\xvt\bin             ; XVT CURL processor

set winsdkver = 0x0300            ; Windows 3.0 SDK
                0x030a            ; Windows 3.1 SDK (default)

set winlib=c:\windev\lib          ; Microsoft C

set bcclib=c:\borlandc\lib        ; Borland C

set                               ; Borland C
bccinclude=c:\borlandc\include
```

System Verification Checklist for Windows

Before compiling and linking your own INFOConnect applications, please review this checklist to insure that all the other required software has been properly installed on your machine.

Did you update your INCLUDE environment variable properly?

The order of the directories in the INCLUDE environment variable are important. If you are using Microsoft C, then the Windows INCLUDE directory must precede the C compiler's INCLUDE directory. The recommended order of products in the INCLUDE variable is: IDK, XVT, WINDOWS, C.

Are you using the Microsoft Segmented Linker?

If you have already successfully built Windows applications on your machine, you can skip this check.

Some Microsoft compilers provide two linkers that are named LINK.EXE: an overlay linker for DOS applications and a segmented linker for Windows and OS/2 applications. **You must use the segmented linker to build Windows applications**. If you get a lot of link errors, you are probably using the overlay linker. Both linkers display a title line identifying themselves. You may want to delete or rename the overlay linker on your system. Microsoft C 5.1 provides both linkers; you may need to manually copy the segmented linker from the OS/2 disks. See the *Windows SDK Tools Guide*, "Chapter 2", for more information about linking Windows applications.

Did you install the Windows 3.0 SDK properly for DLL development?

If you have already successfully built Windows Dynamic Link Libraries (DLL) on your machine, you can skip this check. Windows 3.1 SDK users can also skip this check.

This warning concerns Windows DLLs and is only relevant for INFOConnect library developers using the Windows 3.0 SDK. If you will only be developing applications and accessories, you can ignore it.

There are special Windows SDK libraries needed to build DLLs. During the 3.0 SDK installation, there is a question about optionally installing these libraries which are named with the following format: \WINDEV\LIB\...DLLC...W.LIB. If they are missing from your machine, you will get various errors when you try to link service libraries and external-interface libraries. You can install these libraries without reinstalling the entire SDK by running INSTALL with the -L option. See the *Windows SDK Installation and Update Guide* for more information.

Microsoft C 7.0/Visual C++ 1.0 and OLDNAMES.LIB

If you get unresolved external references when you try to link the sample programs, you may need to use the OLDNAMES.LIB library provided with the Microsoft C 7.0 compiler. Simply add the following statement:

```
set oldnames=oldnames
```

to AutoExec.bat to add OLDNAMES.LIB to the list of libraries to be searched. For more information about OLDNAMES.LIB, refer to the "ANSI Compatibility" section in the Microsoft C 7.0 *Run-Time Library Reference* manual.

```
link @reflectq.lnk

Microsoft (R) Segmented Executable Linker Version 5.30
Copyright (C) Microsoft Corp 1984-1992. All rights reserved.

Object Modules [.obj]: /noe/nod/map/line/co reflectq.obj C:\windev\LIB\libentry
Run File [reflectq.exe]: reflectq.dll
List File [c:reflectq.map]: reflectq.map
Libraries [.lib]: icwin Sdllcew libw
Definitions File [nul.def]: reflectq.def

reflectq.obj(reflectq.c) : error L2029: '_itoa' : unresolved external
```

With Visual C++ 1.0 the opposite problem can occur: having oldnames set may cause unresolved external references. If oldnames is set, execute the following command to unset oldnames and retry the compile:

```
set oldnames=
```

```
 link @ICXVTAP2.lnk

Microsoft (R) Segmented Executable Linker Version 5.50
Copyright (C) Microsoft Corp 1984-1993. All rights reserved.

Object Modules [.obj]: /noe/nod/map/line/co ICXVTAP2.obj C:\Karen\IDK\LIB\WIN\icxevent C:\Karen\IDK\LIB\WIN\icxkey
Run File [ICXVTAP2.exe]: ICXVTAP2.exe
List File [c:ICXVTAP2.map]: ICXVTAP2.map
Libraries [.lib]: icxvt2MMmxvtw      oldnames Mlibcew libw
Definitions File [nul.def]: ICXVTAP2.def

c:\windev\lib\Mlibcew.lib(_cflush.asm) : error L2029: '_flushall' : unresolved external
substitute symbol '__flushall' not found
c:\windev\lib\Mlibcew.lib(fflush.c) : error L2029: '_flushall' : unresolved external
substitute symbol '__flushall' not found
c:\windev\lib\Mlibcew.lib(ftell.c) : error L2029: '_lseek' : unresolved external
substitute symbol '__lseek' not found
```

Try building a sample Windows application

Before building an INFOConnect application, try building one of the Windows SDK sample programs like GENERIC.

Try building a sample INFOConnect application

The final and best verification test is to actually build and execute one of the sample programs provided with the IDK.

The IDK sample directories contains a variety of makefiles to build the sample programs with different compilers. If you use any of the integrated environments like PWB, IDE or Visual C++, you will have to set them up yourself. If you get "Library not found" errors, you may also have to edit the IDK makefiles depending on the "flavor" of Windows libraries installed on your machine (i.e. alternate/emulator/387 math package, small/medium/large memory models).

Microsoft C 6.0, 7.0, and Visual C++ 1.0

```
nmake -f makefile PROGRAM=icwinapp
nmake -f makefile PROGRAM=icwinapp MODEL=L
nmake -f makefile PROGRAM=icxvtapp XVT=y
nmake -f makefile PROGRAM=icxvtap2 XVT=y XVTVER=0x200
nmake -f makelib LIBRARY=service
nmake -f makelib LIBRARY=service MODEL=M
nmake -f makelib LIBRARY=intrface
nmake -f makedos PROGRAM=icdosapp
```

Borland C++

Between the 2.0 and 3.0 releases, Borland changed the names of libraries used during the link process. If you are using the 2.0 release, you must edit the names in the BCCLIBS macro in the *.BCC makefiles.

```
make -f makefile.bcc -DPROGRAM=icwinapp
make -f makefile.bcc -DPROGRAM=icwinapp -DMODEL=L
make -f makefile.bcc -DPROGRAM=icxvtapp -DXVT=y
make -f makelib.bcc -DLIBRARY=service
make -f makelib.bcc -DLIBRARY=service -DMODEL=M
make -f makelib.bcc -DLIBRARY=intrface
make -f makedos.bcc -DPROGRAM=icdosapp
```

Preparing XVT for Use with INFOConnect

If you will be writing XVT applications and haven't installed XVT yet, you must do that before proceeding.

Notes:

- ***The IDK is compatible with XVT 2.0, 3.01 and 3.02, however, the XVT toolkit changed substantially between XVT 2.0 and 3.0x. The sample code fragments shown in this document are based on XVT 3.0x.***
- ***XVT 3.0 users must upgrade to the XVT 3.01 or 3.02 maintenance releases before using the IDK. The INFOConnect/XVT interface is based on the XVT event_hook mechanism which changed between XVT 3.0 and 3.01.***

INFOConnect makes modifications to the XVT include files. To update the XVT include files appropriately, a program called IcXvtMod is provided. IcXvtMod should be run each time a new version of the IDK is installed. To execute it, type the following at the DOS command line:

```
icxvtmod [XVT include directory]
```

For example:

```
cd \idk
icxvtmod \xvt\include
```

IcXvtMod automatically detects and works with XVT 2.0 or XVT 3.0. The new header file, XvtType.h, (or Xvt.h for XVT 2.0) can still be used to write non-INFOConnect XVT applications. The original XvtType.h file is saved in file OldXvt.h.

If you need to rerun IcXvtMod

It is harmless if IcXvtMod is executed multiple times and reprocesses the XVT include files. Earlier versions of IcXvtMod required caution to avoid processing the include files more than once.

Try building a sample INFOConnect/XVT application

The final and best verification test is to actually build and execute one of the sample programs, IcXvtApp, provided with the IDK. See section 4, Writing INFOConnect/XVT Applications, for details about IcXvtApp.

Section 2
An Introduction to INFOConnect Connectivity Services

What is the purpose of INFOConnect Connectivity Services?

INFOConnect Connectivity Services (ICS) provides data communications APIs to applications written for graphical environments such as Windows. INFOConnect is the standard workstation interconnect platform for the Unisys Architecture. The INFOConnect APIs are available in both native-platform formats and XVT. Here are some of the advantages and implications of using INFOConnect Connectivity Services.

- Applications can immediately use new data communications transports without having to be rebuilt. Your application is not "left behind" as new transport technologies develop.
- Applications can be more readily ported to other GUIs by using a common data communications API.
- Applications can easily coexist with other data communications applications on the same workstation. (Datacomm applications have a tendency to take control of datacomm resources and prevent other programs from simultaneously operating).
- Outside developers can supply conforming components at the different layers of INFOConnect's open architecture.
- Application developers can concentrate on the primary purpose of their application and not get sidetracked in all the details of writing data communications code.
- Applications are not concerned as to where the "partner" process is executing. The same APIs are used no matter if the other process is on the same machine or a different one.
- Two non-conforming, non-INFOConnect applications using completely different data communications protocols can be made to communicate through an INFOConnect "coupler" (or "translator") application.

What is the application's perspective of INFOConnect Connectivity Services?

INFOConnect helps you receive and send data across a communications connection much like a file management system helps you read and write data to a file. An INFOConnect ***path*** defines a data communications connection. Paths have names just as files have names. To open a file, you provide the file's name and the file system worries about the details of retrieving the file from disk. Similarly, to establish a datacomm connection, you provide the path name and INFOConnect worries about the specific components and transports needed to complete the connection. After a path is opened, it is called a ***session***. Just as you read and write data blocks to a file, you receive and transmit data buffers over a session. Just as you can have more than one file open at once, you can have more than one INFOConnect session open at a time. Multiple sessions can be open to the same host or different hosts. When you are finished, you must close sessions just like you close files.

The following diagram shows the relationship of an application, INFOConnect, and the lower data communications transport layers. INFOConnect insulates the application from the lower communications layers.

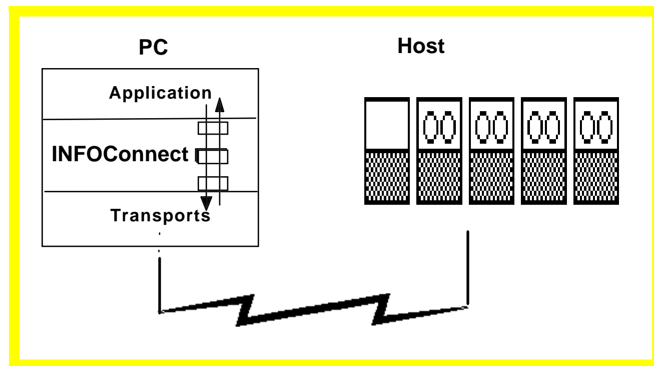


Figure 2–1. Application's Perspective of ICS

However, unlike a simple file management system, INFOConnect functions must operate in an event-driven, multitasking environment. Therefore, INFOConnect functions must always return to the calling code immediately. Often, this means returning before an action is completed. Later, when the action is completed, a message is posted to the application indicating the results. This design models the event-driven architectures of GUI environments.

The following diagram is a simplified illustration of an application calling INFOConnect to transmit data (a more detailed diagram is provided in Section 6, "A Closer Look at the INFOConnect Architecture"). A solid arrow represents a function call or return. Function parameters are written beside the arrows. Functions themselves are represented by boxes. Dotted arrows indicate posted messages which result in separate execution sequences within the GUI system.

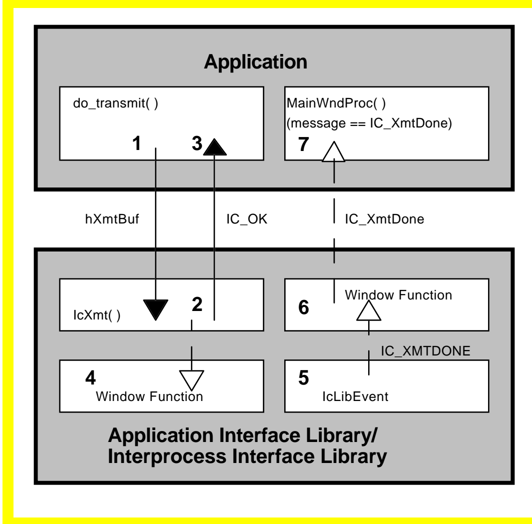


Figure 2–2. Simplified Transmit Illustration

1. The application fills a buffer and passes its handle, hXmtBuf in the diagram, to IcXmt. IcXmt is the name of the INFOConnect AAPI function for transmitting a buffer of data. Do_transmit represents any function in your application that is transmitting some data.

2. The Application Interface Library/Interprocess Interface Library (AIL/IIL) does preliminary verification of the transmit request. If it looks like a valid request, the AIL/IIL posts a message to an internal window on the INFOConnect message queue and returns IC_OK to the application.

3. The application resumes processing until the transmit finishes. The data buffer being transmitted, hXmtBuf, is still unavailable to the application until the transmit done event is returned.

4. The GUI system gives control to the AIL/IIL's Window function to process the message posted in step 2. The AIL/IIL passes the transmit request to the Communication Manager.

5. The Communication Manager passes the transmit request through the INFOConnect architecture, eventually returning a transmit done message to the AIL/IIL. The AIL/IIL posts the transmit done message (IC_XmtDone) to an internal window on the application message queue.
6. When the GUI system gives control to the AIL/IIL, the AIL/IIL posts the transmit done message (IC_XmtDone) to the application.
7. The GUI system gives control to the application to process the transmit done message.

Note: *Applications running on event-driven GUI platforms operate as a series of short execution sequences in response to messages or events. In this example, the do_transmit and MainWndProc functions are executed in two separate execution sequences.*

Architecture Diagram

The following diagram illustrates the primary components of the INFOConnect Connectivity Services architecture:

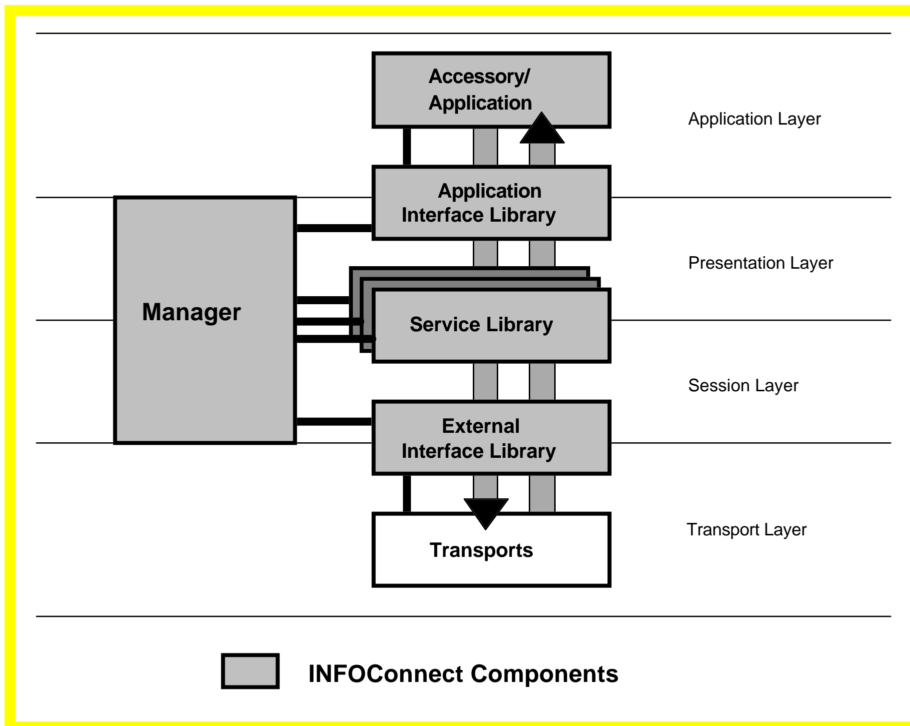


Figure 2–3. INFOConnect 3.0 Architecture

INFOConnect Connectivity Services operate at the upper layers of the OSI communications model. Each corresponding OSI layer - application, presentation, session and transport - is shown down the right side of the diagram. The large vertical arrows represents the application's data passing through the INFOConnect architecture. The thinner black lines indicate the flow of control through the INFOConnect architecture. The INFOConnect Manager controls all interaction between the four components of the INFOConnect architecture: Accessory/Application, Application Interface Library, Service Library, and External Interface Library. Developers can write any of these INFOConnect components.

Terminology

The following terminology is specific to INFOConnect:

Manager Components

Some of the services provided by the manager components are:

- The INFOConnect Manager provides the user interface to all INFOConnect functions: installation, configuration and administration.
- The Communication Manager routes all calls between the library layers, starting from the Application Interface Library down to the External Interface Library (and back). At session establishment, the Communication Manager loads the set of libraries required for the selected path.
- The Configuration Manager is used to configure paths. It is also responsible for controlling library configuration information, the interaction between libraries during configuration, and access to the configuration database.
- The Database Manager maintains a database of all the paths configured by the user. These database files are called INFOConn.cfg and InstMgr.cfg.
- The Installation Manager controls the installation and deinstallation of packages.
- The Quick Configuration Manager controls the sequence of package quick configuration.

Cooperative Application

A cooperative application consists of multiple components usually executing on different systems. INFOConnect helps you write the workstation component of a cooperative application. It is assumed that a graphical user interface (GUI) is running on the workstation. INFOConnect Connectivity Services provides the framework for a data communication session between the components of a cooperative application.

Application

Unless specifically noted otherwise, the terms *INFOConnect application, ICS application* or *application* in this document refer to the workstation piece of a cooperative application. Applications are usually end-user programs that appear as icons on the user's screen. They make use of INFOConnect data communications services.

Accessory

INFOConnect applications that can be invoked and controlled by other INFOConnect applications are called *accessories*. INFOConnect accessories are written so that they can be used to build more sophisticated INFOConnect accessories. It is easy and very useful to extend your INFOConnect application to become an accessory.

Accessory API

The Accessory API (AAPI) defines a collection of services for sending and receiving data across a data communications connection in a transport-independent manner.

DosLink API

The DosLink API is a subset of the Accessory API and defines INFOConnect data communications services available to DOS applications.

XVT

XVT is a software toolkit produced by XVT Software Inc. that provides graphical presentation services like windows, list boxes, scroll bars, etc. to applications.

Application Interface Library

An Application Interface Library (AIL) is the component that supports a specific set of INFOConnect interfaces and provides the application interface to INFOConnect communications. In INFOConnect 3.0, the INFOConnect Accessory AIL (IcAAPI16.dll) exports the session related interfaces of the Accessory API for the Windows (Win16) platform.

The AIL component of an INFOConnect session enables the coexistence of multiple communications interfaces. The AIL requests establishment of an INFOConnect session by calling the Communication Manager to associate the AIL with a path. This allows different applications to use the same path at the same time, or different times, even though they may use different communication APIs.

Service Library

A Service Library (SL) is a filter between the application and the external interface. Each message transmitted or received by the application is passed through the service library. Zero or more service libraries can be stacked in a single INFOConnect session, as indicated in the architecture diagram. Each service library generally operates independently and is unaware of other libraries in the session. Message status as well as message data can be altered by service libraries. Service libraries operate at the OSI Presentation and Session layers. They can be used for things like data compression, data encryption and data conversion. Service libraries can be written for specific applications, but ideally they should be designed to be useful in as many situations as possible. For example, some services may be reusable for a specific class of accessories like Mapper graphics engines, or for a specific hub platform like OS1100, or for a specific transport type like TCP/IP.

Generally, applications do not have to be concerned about service libraries.

External Interface Library

An External Interface Library (EIL) is an adapter to a particular type of communications hardware or software. Each path is configured with a single External Interface Library. EILs typically map into the OSI Transport layer. Applications are unaware of EILs. As new transport technologies develop, new external interfaces can be inserted beneath your applications and service libraries without any changes.

EILs act as the point where a session connects to another "environmental context." This is often an external communications driver, but an EIL can also connect to another INFOConnect session and initiate another pass through the INFOConnect architecture. This effectively means that sessions can be stacked on top of each other and introduces many interesting possibilities.

Generally, applications do not have to be concerned about external interface libraries.

Interprocess Interface Library

An Interprocess Interface library acts as both an AIL and an EIL. An IIL associates two sessions in **different processes** by internally linking the EIL role of one session to the AIL role of the other session. The IIL is automatically included in sessions when an AIL requests a path that must be opened in a different process. An IIL shields the application from needing to know in which underlying environment the INFOConnect subsystem is running.

Stack Interface Library

A Stack Interface library acts as both an AIL and an EIL. A Stack Interface library associates two sessions in the **same processes** by internally linking the EIL role of one session to the AIL role of the other session. Stack libraries can be included in path templates as an EIL.

Multiplexing Library

A multiplexing library is a type of Stack Interface library that can support multiple communication sessions (where it is configured as an EIL) on top of another session. The sessions are associated with a channel in the EIL role which is associated with a lower level path.

Switching Library

A type of Stack Interface library that stacks one session (where it is configured as an EIL) on top of another session and filters the data stream for commands to open and close the lower session. A switching library can also dynamically switch connections of lower sessions to upper sessions.

Hook Library

A special purpose library that provides additional features to the INFOConnect Connectivity Manager. The INFOConnect Trace Facility uses a hook library, called trace log, that manages the trace log file and writes trace information to it.

Quick Configuration Library

A Quick Configuration (Quick Config) library is called by the Quick Configuration Manager after package installation and before package deinstallation. A Quick Config library is also called when a user selects the Config All button or selects a package and then selects the Configure button in the INFOConnect Packages window. The intent of Quick Configuration after package installation is to get the package into a turnkey state with a minimum amount of user input. The intent of Quick Configuration before package deinstallation is to allow the package to participate in package removal.

Exit Hook Library

An Exit Hook library is called by the Installation Manager at different points during the installation and deinstallation of a package. Exit hooks are used to augment the installation process.

INFOConnect Library

Application Interface, Service, External Interface, Stack Interface, Interprocess Interface, Stack Interface, Multiplexing, Switching, Hook, Quick Configuration, and Exit-Hook libraries are collectively referred to as INFOConnect libraries.

Shell

A Shell is an INFOConnect accessory that runs as the INFOConnect Manager. The INFOConnect architecture requires the Shell to be a separately running Windows task. The Shell can optionally provide a user interface to allow session monitoring. It may also include a configurator. Different INFOConnect Shells can be developed using the INFOConnect Shell API. Ic16SS.exe is an alternate INFOConnect Shell available with INFOConnect 3.0.

Configurator

A Configurator is an INFOConnect accessory that provides the user interface for INFOConnect configuration. The INFOConnect architecture allows more than one configurator to be executing simultaneously. Configurators use the Configurator API.

Path

A path defines a data communications connection between the components of a cooperative application. A path is usually defined by the user and has a name called a PathID. A path defines a particular configuration of INFOConnect libraries that must be loaded by the Communications Manager to complete a connection. A path consists of one external interface library and zero or more service libraries. A path can involve communications within the system or with another computer.

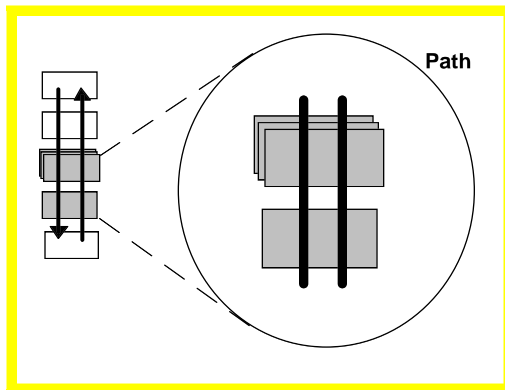


Figure 2–4. Path

Session

A session is an open or active instance of a path. There is often, but not always, a one-to-one relationship between paths and sessions. Some paths can be used to establish multiple, simultaneous sessions. Within an application, a session is uniquely identified by a session handle. The HIC_SESSION data type is used to define sessions. The following diagram shows the relationship between a session and a path:

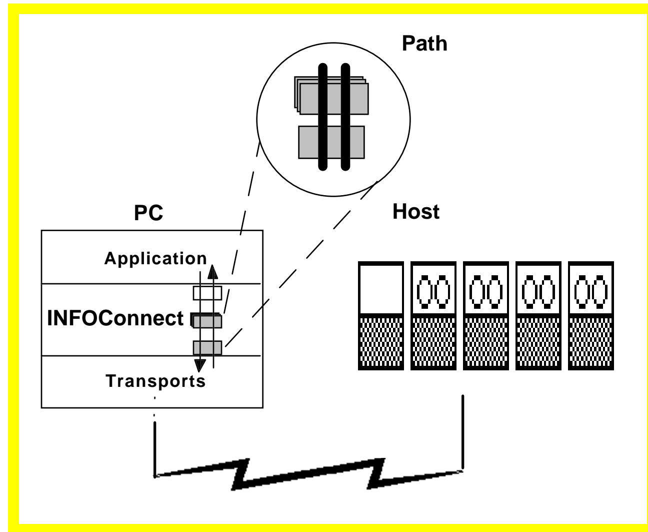


Figure 2–5. Session/Path Relationship

A session consists of the following INFOConnect components: Accessory/Application, Application Interface library, Service libraries (optional), and External Interface library. The following diagram shows the relationship between a session and the INFOConnect architecture:

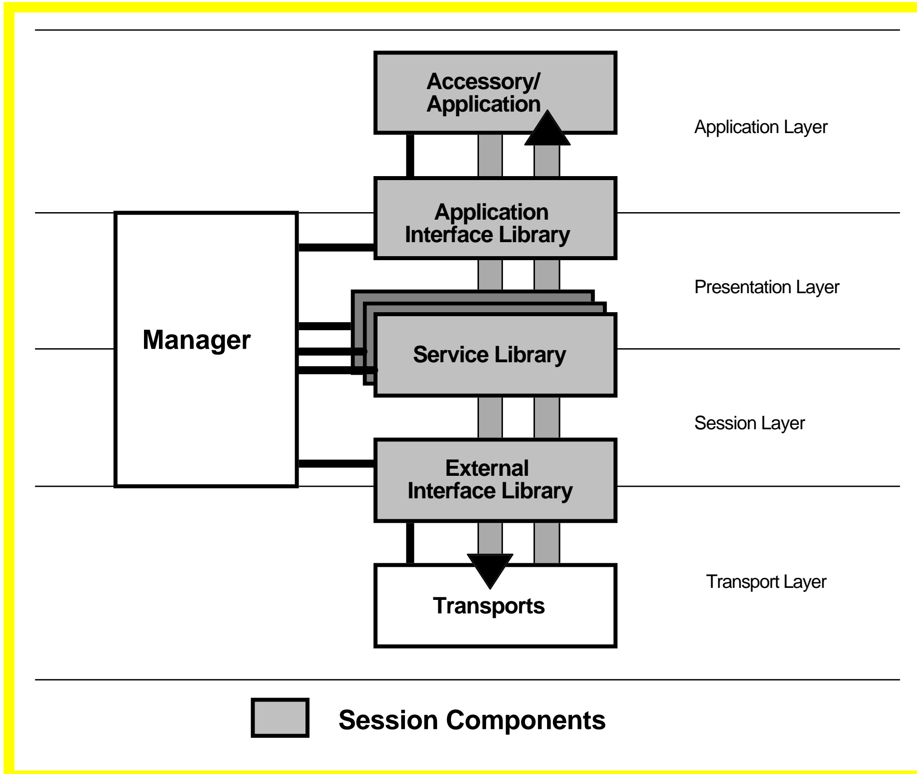


Figure 2–1. Session/Architecture Relationship

Path Template

A path template defines which libraries and channels go together. Templates are "filled in" during path configuration to create paths. The empty boxes and question marks in the diagram below represent dialog box fields that must be completed during path configuration. Templates generally categorize the basic types of connections available on a workstation. This simplifies the path configuration process by reducing many different combinations of libraries to a small set of path templates.

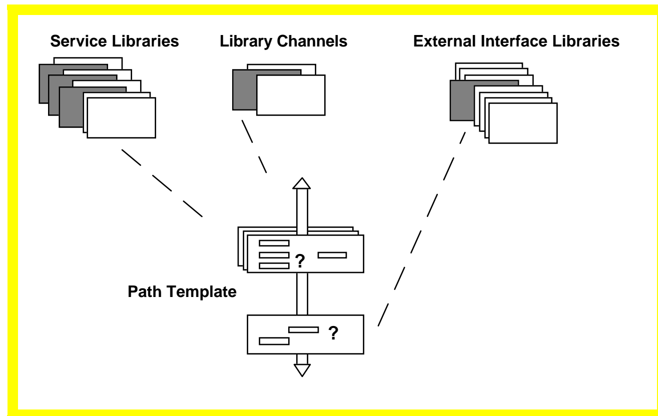


Figure 2–6. Path Template

Library Channel

Paths and path templates reference library channels to avoid duplicating information. Channel data is shared by all paths and templates that use that channel. Channels are provided as a convenience to libraries. Only some libraries define channels. Channels often describe hardware characteristics.

For example, suppose a library makes use of a COM port. Instead of making every path contain the port number, baud rate, etc. that information can be stored in a single library channel record which the paths and path templates then reference.

Inter-application Buffers

Different platforms have different requirements for sharing buffers between applications. Any data that must be passed between your application and INFOConnect must be allocated with memory management functions provided by INFOConnect. A primary example of this type of buffer is a datacomm buffer used to pass transmit/receive data between your application and INFOConnect. IcAllocBuffer allocates fixed, shareable memory. (For XVT, use ic_buf_alloc to obtain an IC_BUFHND handle).

Intra-application Buffers

Some applications require large blocks (greater than provided by a normal heap) of data for local use. GlobalAlloc can be used to allocate large amounts of global memory. (For XVT, use ic_galloc to obtain an IC_MEMHND handle).

Session Information Block

The Session Information Block provides characteristics about a specific session. The INFOConnect libraries are responsible for building the session information block although from the application's perspective it is just a structure provided by INFOConnect. An example of some information in the session block is the maximum transmission blocksize supported across the connection.

IC_RESULT

An IC_RESULT is a small packet of data used to describe errors and statuses. Most INFOConnect functions and events return an IC_RESULT indicating success or failure. Functions exist to translate 'error' IC_RESULTs into displayable text strings. IC_RESULT consists of three parts: a context, a type and a value. Macros exist to tear down or build an IC_RESULT from its parts.

Context

The context is one part of an IC_RESULT packet. It indicates where an error or status is defined, but not necessarily who issued the error or status. A unique context is dynamically assigned at execution time to each INFOConnect component (services, interfaces and accessories). INFOConnect components register a uniquely-identifying string with INFOConnect. Functions exist to convert the assigned context to the registered context-strings and vice versa. Dynamic contexts simplifies the integration of third party INFOConnect components into the INFOConnect environment.

Accessory API Functions

INFOConnect applications have several groups of functions available to them through the Accessory API:

Basic session management functions
Path management functions
Error handling functions
Accessory management functions

The function names listed here are the from the Microsoft Windows version of the Accessory API. The XVT interfaces are equivalent, but have different names which are appropriate for the naming conventions XVT.

Basic Session Management Functions

Every INFOConnect application uses the following functions for basic data communications:

IcInitIcs	Initializes INFOConnect Connectivity Services
IcOpenSession	Initiates session establishment
IcCloseSession	Initiates session termination
IcXmt	Initiates a transmit of a buffer of data
IcRcv	Requests a buffer of data
IcLcl	Cancels pending transmits and/or receives

Less frequently used session management functions:

IcGetSessionID	Returns a session identification string
IcGetSessionInfo	Returns pertinent information about a session
IcChangeHandle	Changes the ownership of an open session
IcSetStatus	Sends a status message
IcExitOk	Responds to an ICS exit request

Path Management Functions

A session is established by selecting and opening a path. Your application can let INFOConnect present a dialog box to the user with the available paths, or you can use the following functions to programmatically choose or create a path to be opened. This gives the application complete control over the user interface and presentation style during path selection.

IcGetNewPath	Initiates a path configuration dialog box
IcGetPathID	Returns the path identification string an active session
IcGetPathNames	Provides a list of configured paths

Error Handling Functions

INFOConnect provides flexible error reporting capabilities. All errors are returned to the application. The application can return the error to INFOConnect for default processing, or it can request text for the error and display the error itself. This gives the application complete control over the user interface and presentation style of errors.

IcDefaultErrorProc	Lets INFOConnect handle the error
IcGetString	Converts an error code to a string
IcSetError	Used by accessories to generate errors

Accessory Management

Accessory functions allow applications to invoke and interact with INFOConnect accessories and also allow applications to register as accessories themselves.

IcOpenAccessory	Used to start an INFOConnect accessory and open a local session to it
IcRunAccessory	Used to start an INFOConnect accessory, but without opening a session to it
IcGetContextString	Converts a context into the accessories string identifier
IcGetContext	Companion to IcGetContextString, converts a string identifier into a context
IcRegisterAccessory	Identifies your application as an accessory
IcDeregisterAccessory	Companion to IcRegisterAccessory

Memory Management Functions

INFOConnect provides a special set of functions for the allocation of data communications buffers. Buffers passed to INFOConnect functions must be allocated with INFOConnect routines. GUI platforms often have special requirements for memory blocks that are shared by multiple tasks or processes; INFOConnect ensures that these requirements are met.

IcAllocBuffer	Allocates fixed, shareable memory (for data communications buffers)
IcGetBufferSize	Returns the size of a buffer allocated with IcAllocBufer
IcReAllocBuffer	Resizes memory allocated with IcAllocBuffer
IcFreeBuffer	Frees memory allocated with IcAllocBuffer
IcLockBuffer	Locks memory allocated with IcAllocBuffer
IcUnlockBuffer	Unlocks memory locked with IcLockBuffer

Accessory API Events

The heart of a GUI application is the message handler. (Some GUI environments may use the term *event* instead of message). An INFOConnect application must be prepared to handle INFOConnect messages in addition to the standard messages generated by the native platform. Some INFOConnect functions return results immediately to the caller, but others (for example, opening a session) return before the action is completed. When the action is completed, an event is passed to the application's message handler.

The event names listed here are the from the Microsoft Windows version of the Accessory API. The XVT interfaces are equivalent, but have different names which are appropriate for the naming conventions XVT.

IC_SESSIONESTABLISHED	Generated after an earlier IcOpenSession request has completed
IC_SESSIONCLOSED	Generated after an earlier IcCloseSession request has completed
IC_RCVDONE	Generated when a receive request completes
IC_RCVERROR	Generated when a receive request fails
IC_XMTDONE	Generated when a transmit request completes
IC_XMTERROR	Generated when a transmit request fails
IC_ERROR	Generated when an unexpected error occurs
IC_STATUS	Generated when a change in INFOConnect status occurs
IC_NEWPATH	Generated when a user finishes an application-initiated configuration dialog

The following code fragment illustrates how your application's Main Window Procedure (MainWndProc) is updated to acknowledge INFOConnect messages. Once again, the Windows interface is used in the example, but the XVT environment is very similar.

```
long FAR PASCAL MainWndProc(HWND hWnd,
          UINT message,
          WPARAM wParam,
          LPARAM lParam)
{

 switch (message) {   /* Windows messages  */
 case WM_COMMAND:
  break;
 case WM_PAINT:
  break;
 case WM_DESTROY:
  break;

 ...
       /* INFOConnect messages */
 case (IC_MSGBASE+IC_SESSIONESTABLISHED):
  ICSSessionEstablished(hWnd, HSESSION, ICRESULT);
  break;
 case (IC_MSGBASE+IC_SESSIONCLOSED):
  ICSSessionClosed(hWnd, HSESSION);
  break;

 ...

 default:
  return(DefWindowProc(hWnd, message, wParam, lParam));
 }
 return(NULL);
}
```

Section 3
Writing INFOConnect/Windows Applications

Writing a Windows application that uses INFOConnect Connectivity Services is like writing a normal Windows application enhanced by extra message capabilities, data structures and functions. It is assumed you are familiar with developing standard Windows applications.

Section 3 leads you through the development of a simple Windows application that uses INFOConnect Connectivity Services. An actual program, IcWinApp, is presented at the end of the section. All source files necessary to build IcWinApp are provided with the IDK development kit.

A Windows application is made up of several source files, some of which include:

a C-language source (.C) file
a Header (.H) file
a Resource (.RC) file

Let's begin by looking at some of the basic data communications tasks you will encounter as you write the C-language source (.C) file.

Basic Procedures for Windows Applications

This section shows how to use the INFOConnect functions to accomplish the basic procedures or tasks that all INFOConnect applications must follow.

Initializing INFOConnect Connectivity Services

All INFOConnect functions and types are defined in one header file, IcWin.h. Include this file after the standard Windows.h header file.

The messages that INFOConnect uses to communicate with your application must be registered with an INFOConnect function, called IcRegisterMsgSession. This function registers messages with Windows on a per-session basis. This function allows developers to add INFOConnect messages to the message switch statement in MainWinProc.

Sample code

The sample code is from IcWinApp.c. It demonstrates how to register INFOConnect messages IcRegisterMsgSession when the application is running with ICS Release 3.0 or higher.

```
#include <windows.h>
#include <icwin.h>

#define HSESSION wParam

IC_RESULT icerror;
unsigned ICSVersion = IC_VERSION_3_0;

BOOL InitInstance(HANDLE hInstance,
         int nCmdShow,
         LPSTR lpCmdLine)
{
...

  /* Initialize INFOConnect Interfaces */
  icerror = IcInitIcs(IC_VERSION_3_0,IC_REVISION_3_0);

...
  if (ICSVersion >= IC_VERSION_3_0)
    icerror = IcRegisterMsgSession(s.hSession, hWnd, s.hSession,
      IC_MSGBASE, IC_LCLRESULT);
...
}
```

```
long FAR PASCAL MainWndProc( HWND hWnd,
             UINT message,
             WPARAM wParam,
             LPARAM lParam)
{
...
 switch (message) {
...
 case (IC_MSGBASE+IC_SESSIONESTABLISHED):
   ICSSessionEstablished(hWnd, HSESSION, ICRESULT);
   break;

 case (IC_MSGBASE+IC_SESSIONCLOSED):
   ICSSessionClosed(hWnd, HSESSION);
   break;

 case (IC_MSGBASE+IC_STATUS):
   ICSStatus(hWnd, HSESSION, ICRESULT);
   break;

 case (IC_MSGBASE+IC_XMTDONE):
   ICSXmtDone(hWnd, HSESSION);
   break;

 case (IC_MSGBASE+IC_RCVDONE):
   ICSRcvDone(hWnd, HSESSION, ICBUFFER, ICLENGTH);
   break;
...
 default:
   if ((ICSVersion >= IC_VERSION_3_0) ||
     (! ProcessICS20Msg(hWnd, message, wParam, lParam)))
     return(DefWindowProc(hWnd, message, wParam, lParam));
 }
 return(NULL);
}
```

Another way to register the messages that INFOConnect uses to communicate with your application is with the Windows function RegisterWindowMessage. Use this method if the application is intended to run with old versions of ICS. A good place to register messages is in your InitInstance routine. The REGISTERWINDOWMESSAGE macro used in the following example is not required, but you may find it useful.

Sample code

The sample code is from IcWinApp.c. It demonstrates how to register INFOConnect messages RegisterWindowMessage when the application is running with ICS Release 2.0. The messages are registered in function RegisterICS20Msgs.

```
#include <windows.h>
#include <icwin.h>

IC_RESULT icerror;

/* INFOConnect message numbers for 2.0 support*/
static unsigned IC_SessionEstablished;
static unsigned IC_SessionClosed;
...
static unsigned IC_LdResult;

BOOL InitInstance(HANDLE hInstance,
          int nCmdShow,
          LPSTR lpCmdLine)
{
...
   icerror = IcInitIcs(IC_VERSION_2_0,IC_REVISION_2_0);
   if IC_CHECK_RESULT_SEVERE(icerror) {
     LoadString(hInst,ICS_INIT_FAILED,sNoteBuf,sizeof(sNoteBuf));
     MessageBox(hWnd,sNoteBuf,QAPPNAME,MB_OK);
   } else {
     RegisterICS20Msgs();
     ICSVersion=IC_VERSION_2_0;/* for IcOpenSession/IcRegisterMsgSession */
   }
...
}

BOOL RegisterICS20Msgs (void)
{
#define REGISTERWINDOWMESSAGE(n) (n=RegisterWindowMessage(#n))

  /* Register INFOConnect message numbers */
  if ((!REGISTERWINDOWMESSAGE(IC_SessionEstablished)) ||
     (!REGISTERWINDOWMESSAGE(IC_SessionClosed)) ||
...
     (!REGISTERWINDOWMESSAGE(IC_LdResult))) {
    assert(FALSE);
    return(FALSE);
  }
  return(TRUE);
}
```

Opening a Session and Allocating Buffers

Before you can send data with INFOConnect, you must open an INFOConnect session. The steps to open a session and allocate datacomm buffers follow:

Initialize INFOConnect

Initialize INFOConnect Connectivity Services using IcInitIcs if you haven't done so already.

Call IcOpenSession

Call IcOpenSession to request a path to be opened. You can supply a path or let INFOConnect prompt the user for a path. If INFOConnect returns with a non-severe error, then a pending session handle has been created. INFOConnect will pass an IC_SessionEstablished message to your window function later when the session is actually established.

Do not use the pending session handle before the IC_SessionEstablished message occurs. No IC_SessionEstablished message is generated if INFOConnect returns a severe error on the IcOpenSession call.

Handle the IC_SessionEstablished message and allocate buffers

Add a test for IC_SessionEstablished to your window function with code to process the session establishment message and allocate buffers for transmitting and receiving data. Call IcGetSessionInfo to find the maximum buffer size the underlying communications software can support. Since connections may support very large buffers, you may want to put a ceiling on the buffer size as shown in the code fragment. Function IcAllocBuffer must be used to allocate the buffers to satisfy underlying platform requirements for shared global handles. Be sure to check for NULL on the allocation requests.

Define a boolean indicating 'session establishment'

It's a good idea to create and set a global boolean variable in your application to indicate the current state of your session. This is primarily needed during the time between the IcOpenSession call and before the IC_SessionEstablished message is returned to your application. The sample code uses *bSessionEst* within the session structure and makes no INFOConnect requests using the pending session handle until *bSessionEst* is TRUE.

Call IcCloseSession after errors

If the IC_SessionEstablished message contains a severe error you must call IcCloseSession to release the pending session handle.

Sample code

```
struct aSession {
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
  unsigned uBufsize;
}s;

IC_SINFO sinfo;
IC_RESULT icerror;

#define MAXBUFSIZE 4096
#define HSESSION wParam

BOOL InitInstance(HANDLE hInstance,
          int nCmdShow,
          LPSTR lpCmdLine)
{
  HWND hWnd;

  hInst = hInstance;
  hWnd = CreateWindow(...);

  if ((icerror = IcInitIcs(IC_VERSION_3_0, IC_REVISION_3_0)) != IC_OK) {
    HandleIcError(hWnd, NULL, NULL, icerror);
    return(FALSE);
  }
  s.hSession = NULL_HIC_SESSION;
  s.hXmtBuf = NULL;
  s.hRcvBuf = NULL;
  s.bSessionEst = FALSE;
  s.uBufsize = 0;

  icerror = IcOpenSession(hWnd, NULL, &s.hSession);
  if IC_CHECK_RESULT_SEVERE(icerror) {
    IcDefaultErrorProc(hWnd, s.hSession, NULL, icerror);
    return(FALSE);
  }
  return(TRUE);
}

long FAR PASCAL MainWndProc(HWND hWnd,
               UINT message,
               WPARAM wParam,
               LPARAM lParam)
{
...
  switch (message) {
...
  case (IC_MSGBASE+IC_SESSIONESTABLISHED):
    ICSSessionEstablished(hWnd, HSESSION, ICRESULT);
    break;
...
}
```

```
void ICSSessionEstablished(HWND hWnd,
           HIC_SESSION hSession,
           IC_RESULT result)

{
 assert(hSession==s.hSession); /* bad session */
 if IC_CHECK_RESULT_SEVERE(result) {
  HandleIcError(hWnd,hSession,IC_SESSIONESTABLISHED,result);
  IcCloseSession(hSession);
 }
 else {
  icerror = IcGetSessionInfo( hSession, &sinfo);
  if IC_CHECK_RESULT_SEVERE(icerror)
   HandleIcError(hWnd,hSession,NULL,icerror);
  s.bFocusNotify = sinfo.focus_notify;
  s.bBlockingOn = sinfo.block_mode;
  UpdateWindowName( hWnd, sWindowName);

  /* Allocate INFOConnect buffers. */
  s.uBufsize=min((unsigned)sinfo.max_size,MAXBUFSIZE);
  s.hXmtBuf = IcAllocBuffer(s.uBufsize);
  s.hRcvBuf = IcAllocBuffer(s.uBufsize);
  assert(s.hXmtBuf!=NULL);
  assert(s.hRcvBuf!=NULL);
  s.bSessionEst = TRUE;
  IcSetStatus(hSession, IC_TRANSACTION_ON);
  s.nRcvTries = 1;
  icerror = IcRcv( hSession, s.hRcvBuf, s.uBufsize);
  if IC_CHECK_RESULT_SEVERE(icerror) {
   s.nRcvTries = 0;
   HandleIcError( hWnd,hSession,NULL,icerror);
  }
 }
}
```

Transmitting a Buffer

The IcXmt function attempts to transmit a buffer of data. This function is asynchronous in nature; it returns immediately to the application before the transmit is completed. A non-severe error is returned indicating the transmit has been initiated. When the transmit finishes, one of two messages is passed to your window function: IC_XmtDone or IC_XmtError. The basic steps to follow are:

Don't transmit prematurely

Don't transmit over a session before the IC_SessionEstablished message is returned for that session. Don't transmit while a previous transmit request is still pending for that session. The sample code below uses two variables, bSessionEst and nXmtTries, to manage these conditions.

Allocate and lock a transmit buffer

Allocate a transmit buffer using IcAllocBuffer if you haven't already done so. Lock the transmit buffer with IcLockBuffer.

Fill the buffer and unlock it

Fill the transmit buffer and unlock it with IcUnlockBuffer.

Call IcXmt

Pass the buffer to INFOConnect with IcXmt.

You should declare a variable to indicate that a transmit request is outstanding. Check this variable to avoid transmitting a second buffer before a previous transmit has finished.

One approach is to simply use a boolean to indicate outstanding transmit requests. Don't transmit unless the boolean is clear, then set the boolean after calling IcXmt successfully. Clear the boolean when an IC_XMTDONE or IC_XMTERROR message is received.

The sample program, IcWinApp, uses a different approach. A counter named nXmtTries indicates when a transmit request is pending and will also be useful later to manage transmit errors and retries.

Handle the IC_XmtDone and IC_XmtError messages

Add tests for IC_XmtDone and IC_XmtError to your window function. Set your variable to indicate that transmits are now allowed.

The IC_XmtDone message contains the buffer handle and buffer length of the transmitted data.

The IC_XmtError message contains an IC_RESULT with the reason for the transmit failure. Don't ignore these messages. You may want to display the error or retry the transmit. See page 3 - 32 for more on handling data communications errors.

Sample code

```
struct aSession {
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
  unsigned uBufsize;
  int nXmtTries;
}s;

long FAR PASCAL MainWndProc(HWND hWnd,
                UINT message,
                WPARAM wParam,
                LPARAM lParam)
{
  switch (message) {
  case WM_CHAR:
    /*
      The user has pressed a key. Use a dialog box
      to get the message text to be transmitted.
      Put it in sNOTEBUF, then use 'lstrcpy' to move
      the message text into the transmit buffer.
    */

    if ((s.nXmtTries>0) || (!s.bSessionEst)) {
      LoadString(hInst, XMT_NOT_DONE,
          sNoteBuf, sizeof(sNoteBuf));
      MessageBox (hWnd,
          (LPSTR)sNoteBuf,
          QAPPNAME,
          MB_ICONEXCLAMATION | MB_OK);
    }
    else {
      lpXmtDlg = MakeProcInstance(XmtDlg, hInst);
      if (DialogBox(hInst, "XmtDlg", hWnd, lpXmtDlg)) {
        if ((buf=IcLockBuffer(s.hXmtBuf)) == NULL) {
          assert(FALSE);
        }
        else {
          if (!s.bBlockingOn)
            lstrcat(sNoteBuf, "\r\n");
          lstrcpy(buf, (LPSTR)sNoteBuf);
          IcUnlockBuffer(s.hXmtBuf);
          IcSetStatus(s.hSession, IC_TRANSACTION_BEGIN);
          s.nXmtTries = 1;
          icerror = IcXmt(s.hSession, s.hXmtBuf,
                strlen(sNoteBuf));
          if IC_CHECK_RESULT_SEVERE(icerror) {
            s.nXmtTries = 0;
            IcSetStatus(s.hSession, IC_TRANSACTION_END);
            HandleIcError(hWnd, s.hSession,
                  NULL, icerror);
          }
        }
      }
      FreeProcInstance(lpXmtDlg);
    }
    break;
...
}
```

```
void ICSXmtDone(HWND hWnd,
        HIC_SESSION hSession)
{
 NOREF(hWnd);
 assert(hSession==s.hSession);
 s.nXmtTries = 0; /* no outstanding transmits */
}

void ICSXmtError(HWND hWnd,
         HIC_SESSION hSession,
         IC_RESULT result)
{

 /* Ignore warnings and informational errors,
   Xmt request still outstanding.
 */
 if (IC_CHECK_RESULT_SEVERE(result)) {
   assert(hSession==s.hSession);
   if (++(s.nXmtTries) > MAXRETRIES) {
     HandleIcError(hWnd,hSession,IC_XMTERROR,result);
     s.nXmtTries = 1;
   }
   if (s.bSessionEst) {
     /* Try again. No need to IC_TRANSACTION_BEGIN since its
       already been sent in WM_CHAR.
     */
     icerror = IcXmt(hSession,s.hXmtBuf,
           strlen(sNoteBuf));
     if IC_CHECK_RESULT_SEVERE(icerror) {
       HandleIcError(hWnd,hSession,NULL,icerror);
       s.nXmtTries = 0;
       IcSetStatus(s.hSession,IC_TRANSACTION_END);
     }
   }
   else
     s.nXmtTries = 0;
 }
}
```

Receiving a Buffer

The IcRcv function requests a buffer of data. This function is asynchronous in nature; it returns immediately to the application before the receive request is completed. A non-severe error is returned indicating the receive request was initiated. When the receive request finishes, one of two messages is passed to your window function: IC_RcvDone or IC_RcvError. The basic steps to follow are:

Don't issue a receive request prematurely

Don't use a session before the IC_SessionEstablished message is returned for that session. Don't make a receive request while a previous receive request is still pending for that session. The sample code below uses two variables, bSessionEst and nRcvTries, to manage these conditions.

Allocate a receive buffer

Allocate a receive buffer using IcAllocBuffer if you haven't already done so.

Call IcRcv

Pass the buffer to INFOConnect with IcRcv.

You should declare a variable to indicate that a receive request is outstanding. Check this variable to avoid making a second receive request with the previous one still outstanding.

One approach is to simply use a global boolean to indicate outstanding receive requests. Don't issue a receive request unless the boolean is clear, then set the boolean after calling IcRcv successfully. Clear the boolean when an IC_RcvDone or IC_RcvError message is received.

The sample program, IcWinApp, uses a different approach. A counter named nRcvTries indicates when a receive request is pending and will also be useful later to manage receive errors and retries.

Handle the IC_RcvDone and IC_RcvError messages

Add tests for IC_RcvDone and IC_RcvError messages to your window function. Set your variable to indicate new receive requests are now allowed.

IC_RcvDone messages contain the buffer handle and buffer length of the received data.

IC_RcvError messages contain an IC_RESULT with the reason for the receive failure. Don't ignore these messages. You may want to display the error or retry the receive. See page 3 - 32 for more on handling data communications errors.

Sample code

```
struct aSession {
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
  unsigned uBufsize;
  int nRcvTries;
}s;

void ICSSessionEstablished(HWND hWnd,
            HIC_SESSION hSession,
            IC_RESULT result)

{
...
   /* Allocate INFOConnect buffers. */
   s.uBufsize=min((unsigned)sinfo.max_size,MAXBUFSIZE);
   s.hXmtBuf=IcAllocBuffer(s.uBufsize);
   s.hRcvBuf=IcAllocBuffer(s.uBufsize);
   assert(s.hXmtBuf!=NULL);
   assert(s.hRcvBuf!=NULL);
   s.bSessionEst=TRUE;
   IcSetStatus(hSession,IC_TRANSACTION_ON);
   s.nRcvTries=1;
   icerror=IcRcv(hSession,s.hRcvBuf,s.uBufsize);
   if IC_CHECK_RESULT_SEVERE(icerror){
     s.nRcvTries=0;
     HandleIcError(hWnd,hSession,NULL,icerror);
   }
  }
}
```

```
void ICSRcvDone(HWND hWnd,
        HIC_SESSION hSession,
        HANDLE hBuffer,
        unsigned buflen)
{
  LPSTR buf;
  char sTitle[100];
  unsigned i;

  /* Display the received message */
  assert(hSession==s.hSession);
  s.nRcvTries = 0; /* no outstanding receives */
  buf = IcLockBuffer(hBuffer);
  assert(buf!=NULL);
  for (i=0; i < buflen && i < sizeof(sNoteBuf)-2; i++)
    sNoteBuf[i] = buf[i];
  sNoteBuf[i] = 0;
  IcUnlockBuffer(hBuffer);
  IcSetStatus(hSession, IC_TRANSACTION_END);
  LoadString(hInst, RCV_MSG_PREFIX, sTitle, sizeof(sTitle));
  MessageBox (hWnd,
        (LPSTR)sNoteBuf,
        sTitle,
        MB_ICONINFORMATION | MB_OK);
  s.nRcvTries = 1;
  icerror = IcRcv(hSession, s.hRcvBuf, s.uBufsize);
  if IC_CHECK_RESULT_SEVERE(icerror) {
    s.nRcvTries = 0;
    HandleIcError(hWnd, hSession, NULL, icerror);
  }
}

void ICSRcvError(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{
  /* Ignore warnings and informational errors,
    Rcv request still outstanding. */
  if (IC_CHECK_RESULT_SEVERE(result)) {
    /* Don't send IC_TRANSACTION_END since we keep trying until
      successful.
    */
    assert(hSession==s.hSession);
    if (++(s.nRcvTries) > MAXRETRIES) {
      HandleIcError(hWnd, hSession, IC_RCVERROR, result);
      s.nRcvTries = 1;
    }
    if (s.bSessionEst) {
      /* try receive again */
      icerror = IcRcv(hSession, s.hRcvBuf, s.uBufsize);
      if IC_CHECK_RESULT_SEVERE(icerror) {
        HandleIcError(hWnd, hSession, NULL, icerror);
       s.nRcvTries = 0;
      }
    }
    else
      s.nRcvTries = 0;
  }
}
```

Using Datacomm Buffers

This section covers some guidelines that will help you better manage your application's datacomm buffers. The IcRcv and IcXmt functions are asynchronous in nature; they return immediately before the datacomm request is actually completed. This means your application must be careful about accessing the datacomm buffers that were passed to these functions. The following list will help you better manage the data communications of your application.

Allocate datacomm buffers using INFOConnect routines

Use IcAllocBuffer to allocate buffers that will be passed to INFOConnect. This ensures that the buffers have the proper system attributes. Datacomm buffers must be global and shareable across applications.

INFOConnect datacomm is asynchronous

Calls to IcRcv and IcXmt return to your application immediately before the request has actually completed. A message will be sent to your window function when the request is completed. Until this message is returned, your request is referred to as pending.

Don't use pending buffers

Do not access a buffer that is associated with a pending request. It's a good idea to define and set a variable for each buffer that tracks when the buffer is associated with a pending request.

Don't issue a receive or transmit request until the IC_SessionEstablished message has been returned.

Don't issue a receive request while a pending receive request exists for the same session. The state of the first receive request is undefined. Normally, the first request is canceled without returning an IC_RcvDone or IC_RcvError message, but your application cannot assume that all external interfaces will behave similarly for this situation. The same holds true for premature transmit requests.

Use one receive buffer and one transmit buffer

You can have as many datacomm buffers as you like, but most sessions can only have one active transmit buffer and one active receive buffer at a time. Therefore, it is recommended to allocate one receive buffer and one transmit buffer.

You can use one buffer for both transmitting and receiving, but for maximum interoperability use separate buffers. Your application can be more responsive using separate buffers since it can keep a receive request pending while waiting for a pending transmit request to complete.

Note: *Some sessions, dependent on the libraries, may be able to handle multiple transmit requests and multiple receive requests. Most libraries only allow one outstanding receive request and one outstanding transmit request at a time. This should be considered during the design phase of the application so as not to limit the usage of the application.*

Cancel pending requests with IcLcl

Pending requests can be canceled by using IcLcl. You must wait until an IC_LclResult message is returned to your application on behalf of the canceled messages before you can safely access any datacomm buffers.

Don't ignore errors

Don't ignore error messages (IC_XmtError and IC_RcvError). You may want to retry the request at least some number of times. See page 3 - 32 for more on handling data communications errors.

Basic Error Handling

Nearly every INFOConnect function and message returns a 'long' (type IC_RESULT) indicating the success of the function or message. Any value other than IC_OK indicates an error. Your application can handle errors in several ways:

- Pass errors back to INFOConnect for handling
- Display and handle errors yourself
- Test for standard errors and resume without any user intervention
- Handling IC_Error messages

Here are the four categories of errors and recommended actions for each:

IC_ERROR_INFO	Log this error if the application has a log file. Don't bother displaying a message to the user. The requested function was completed.
IC_ERROR_WARNING	The requested function was completed, but something unusual or noteworthy happened. The application can choose to log or display this error. The default error procedure will display these errors.
IC_ERROR_SEVERE	The requested function did not complete successfully. The user should usually see this error.
IC_ERROR_TERMINATE	Display this error and close the session.

Passing errors back to INFOConnect for handling

If your application doesn't need to control the presentation style of the error message, the simplest way to handle errors is to pass them back to the INFOConnect default error procedure.

Note: *The INFOConnect default error procedure will automatically call IcCloseSession on your behalf for errors of type IC_ERROR_TERMINATE. On return from the default procedure, it is good practice to test for IC_ERROR_TERMINATE type errors and, at a minimum, mark the session as closed. Otherwise, you might inadvertently make a transmit or receive request before the IC_SessionClosed event is sent to your application.*

The following code fragment illustrates using the default error procedure after receiving an INFOConnect message (IC_XMTERROR) and also after calling an INFOConnect function.

```
void ICSXmtError(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{

 /* Ignore warnings and informational errors,
   Xmt request still outstanding.
 */
 if (IC_CHECK_RESULT_SEVERE(result)) {
   assert(hSession==s.hSession);
   if (++(s.nXmtTries) > MAXRETRIES) {
     HandleIcError(hWnd, hSession, IC_XMTERROR, result);
     s.nXmtTries = 1;
   }
   if (s.bSessionEst) {
     /* Try again. No need to IC_TRANSACTION_BEGIN since its
       already been sent in WM_CHAR.
     */
     icerror = IcXmt(hSession, s.hXmtBuf,
           strlen(sNoteBuf));
     if IC_CHECK_RESULT_SEVERE(icerror) {
       HandleIcError(hWnd, hSession, NULL, icerror);
       s.nXmtTries = 0;
       IcSetStatus(s.hSession, IC_TRANSACTION_END);
     }
   }
   else
     s.nXmtTries = 0;
 }
}


void HandleIcError(HWND hWnd,
         HIC_SESSION session,
         unsigned message,
         IC_RESULT icerror)
{
 IcDefaultErrorProc(hWnd, session, message, icerror);
 if (IC_GET_RESULT_TYPE(icerror) >= IC_ERROR_TERMINATE)
   TerminateApplication(hWnd);}
```

The advantage of using function HandleIcError instead of calling IcDefaultErrorProc directly is to leave your application the flexibility of adding code in HandleIcError to format your own messages in the future.

Look at the two calls to HandleIcError. Compare the 3rd parameter on each call. On the first call, the message type is known (IC_XMTERROR) and is passed to IcDefaultErrorProc. On the second call, NULL is passed because there is no relevant 'message type' after making an INFOConnect function call. Most of the time, you will be passing NULL for the 3rd parameter. For more examples on using the message-type parameter with IcDefaultErrorProc, see the IDK sample programs.

Displaying and handling errors yourself

To display error messages directly from your application, IcGetString is available to retrieve the text of the error. You must also satisfy the following requirements normally provided by IcDefaultErrorProc.

- Don't display informational errors. Informational errors are of type IC_ERROR_INFO. IC_GET_RESULT_TYPE is an INFOConnect macro used to extract the error type from an IC_RESULT.
- There are a few termination messages used by INFOConnect to close sessions without displaying anything to the user. You should suppress displaying the following:

  IC_ERROR_TERMINATE_NOMSG
  IC_ERROR_TERMINATE_CLEAR
  IC_ERROR_TERMINATE_EXIT
  IC_ERROR_TERMINATE_SHUTDOWN.
- Close the session for errors of type IC_ERROR_TERMINATE. IC_GET_RESULT_TYPE is an INFOConnect macro used to extract the error-type from an IC_RESULT.

```
void foobar()
{
 IC_RESULT icerror;

 icerror = IcGetSessionInfo(hSession,...)
 if IC_CHECK_RESULT_SEVERE(icerror)
         HandleIcError(hWnd,hSession,NULL,icerror);

}

void HandleIcError(HWND    hWnd,
        HIC_SESSION   hSession,
        unsigned  uType,
        IC_RESULT icerror)
{
  char msg[256];

  if (((IC_GET_RESULT_TYPE(icerror) & IC_ERROR_MASK) ==
                                          IC_ERROR_INFO) &&
    (icerror != IC_ERROR_TERMINATE_NOMSG) &&
    (icerror != IC_ERROR_TERMINATE_CLEAR) &&
    (icerror != IC_ERROR_TERMINATE_EXIT))
  {
   if (IcGetString(hSession,
          icerror,
          (LPSTR)msg,
          sizeof(msg)) == IC_OK) {

     MessageBox(hWnd, msg, "My Caption", IDOK);
   }
  }
  if ((IC_GET_RESULT_TYPE(icerror) & IC_ERROR_MASK) ==
     IC_ERROR_TERMINATE ) {
   IcCloseSession(hSession);
  }
  return IC_OK;
}
```

Testing for standard errors

Standard INFOConnect errors are defined in IcError.h. You may find some errors that your application can intercept and resolve without bothering the user.

Handling IC_Error messages

Although they are infrequent, all applications must be prepared to handle IC_Error messages. Assuming you have written a function like HandleIcError shown above, you can call it as follows:

```
#define HSESSION wParam

long FAR PASCAL MainWndProc(HWND hWnd,
                UINT message,
                WPARAM wParam,
                LPARAM lParam)
{
  switch (message) {
...
  case (IC_MSGBASE+IC_ERROR):
    ICSError(hWnd, HSESSION, ICRESULT);
    break;
...
}

void ICSError(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{
  NOREF(hWnd);
  assert(hSession==s.hSession);
  HandleIcError( hWnd,hSession,IC_ERROR,result);
}
```

Basic Status Handling

Status messages are used to communicate between your application and INFOConnect. Statuses can travel in one of two directions: from the application to INFOConnect or from INFOConnect to the application. Your application uses function IcSetStatus to send statuses and watches for message IC_Status to receive statuses from INFOConnect.

Standard statuses are defined in the IcStatus.h header file. Here is a sampling of statuses:

```
/* Line state values */
#define IC_LINESTATE_XMT ...
#define IC_LINESTATE_RCV ...
#define IC_LINESTATE_LCL ...

/* Connection state */
#define IC_CONNECT_EOF ...
#define IC_CONNECT_CLOSE ...
#define IC_CONNECT_OPEN ...
#define IC_CONNECT_NOACTIVITY ...
#define IC_CONNECT_ACTIVITY ...

/* Reactivate session when sinfo.focus_notify == TRUE */
#define IC_REACTIVATE_ON ...
#define IC_REACTIVATE_OFF ...

/* Requests to applications */
#define IC_CONTROL_ACTIVATE ...
#define IC_CONTROL_RCVREADY ...
#define IC_CONTROL_RCVAVAIL ...
```

Does my application need to do something for every possible status?

No. Most statuses can be ignored by your application. Here are the primary statuses your application should support:

- IC_CONTROL_ACTIVATE
- IC_REACTIVATE_ON and IC_REACTIVATE_OFF
- IC_CONTROL_RCVAVAIL
- IC_STATUS_TRANS

Handling IC_CONTROL_ACTIVATE statuses

If INFOConnect wants your application to become the active window, it sends an IC_CONTROL_ACTIVATE status. The following code fragment shows how to support this status:

```
void ICSStatus(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{
...
  if (result==IC_CONTROL_ACTIVATE)
    SetFocus(hWnd);
...
}
```

Sending IC_REACTIVATE statuses

Sometimes the underlying INFOConnect libraries are required to be notified when your application-window gains and loses focus. This is an attribute of the session and is referred to as focus-notification. Your application must do two things related to focus-notification.

- Call IcGetSessionInfo to determine if your session requires focus-notification
- Recognize WM_ACTIVATE messages and send IC_REACTIVATE statuses

```
long FAR PASCAL MainWndProc(HWND hWnd,
            UINT message,
            WPARAM wParam,
            LPARAM lParam)
{
 switch (message) {
 case WM_ACTIVATE:
  /*
   'wParam' indicates whether we are 'activating'
   or 'deactivating' a window.
  */
  if ((s.bFocusNotify) && (s.bSessionEst)) {
   if (wParam)
    s.icstatus = IC_REACTIVATE_ON;
   else
    s.icstatus = IC_REACTIVATE_OFF;
   icerror=IcSetStatus(s.hSession, s.icstatus);
   if IC_CHECK_RESULT_SEVERE(icerror) {
    s.icstatus = IC_OK;
    s.bFocusNotify = FALSE;
    HandleIcError(hWnd, s.hSession, NULL, icerror);
    s.bFocusNotify = TRUE;
   }
  }
  return(DefWindowProc(hWnd, message, wParam, lParam));
...
}

void ICSSessionEstablished(HWND hWnd,
            HIC_SESSION hSession,
            IC_RESULT result)

{
...
  icerror = IcGetSessionInfo(hSession, &sinfo);
  if IC_CHECK_RESULT_SEVERE(icerror)
   HandleIcError(hWnd, hSession, NULL, icerror);
  s.bFocusNotify = sinfo.focus_notify;
  s.bBlockingOn = sinfo.block_mode;
  UpdateWindowName(hWnd, sWindowName);
...
}
```

Handling IC_CONTROL_RCVAVAIL statuses

If your application normally stays in receive mode, you do not need to watch for IC_CONTROL_RCVAVAIL statuses, but applications that intentionally stay out of receive mode for long periods of time should recognize this status. When an application is not in receive mode and a message becomes available, the underlying INFOConnect libraries can send an IC_CONTROL_RCVAVAIL status to request your application to go into receive mode. Ignoring IC_CONTROL_RCVAVAIL statuses may result in your application appearing sluggish and unresponsive.

Sending IC_STATUS_TRANS statuses

In order for INFOConnect to keep an accurate count of transactions, your application needs to notify INFOConnect of the beginning and the end of the transactions. Use IC_TRANSACTION_ON and IC_TRANSACTION_OFF to indicate whether or not the transactions will be flanked by IC_TRANSACTION_BEGIN and IC_TRANSACTION_END status messages.

Closing a Session

Call IcCloseSession to end an INFOConnect session. This function returns immediately. Message IC_SessionClosed is passed to the window function when the session is actually closed. The basic steps to follow are:

Clear the boolean indicating 'session establishment'

The sample code uses a variable named bSessionEst for this purpose. This variable was originally set when the IC_SessionEstablished message was received by the application.

Call IcCloseSession and deallocate buffers

Use INFOConnect memory management routines to free datacomm buffers. Notice that datacomm buffers can be freed immediately after returning from IcCloseSession rather than waiting for the IC_SessionClosed message. The Manager cancels any outstanding transmit or receive requests during IcCloseSession to ensure that the datacomm buffers are idle. It's a good idea to set the buffer handle variables to NULL after releasing the buffers.

Handle the IC_SessionClosed message

This is the best place to clear the session handle with NULL_HIC_SESSION rather than immediately after the IcCloseSession function call.

Sample code

```
#define HSESSION wParam

void TerminateApplication(HWND hWnd)
{
  s.bSessionEst = FALSE;
  if (s.hSession != NULL_HIC_SESSION) {
    IcCloseSession(s.hSession);
  }
  if (s.hXmtBuf != NULL) {
    IcFreeBuffer(s.hXmtBuf);
    s.hXmtBuf = NULL;
  }
  if (s.hRcvBuf != NULL) {
    IcFreeBuffer(s.hRcvBuf);
    s.hRcvBuf = NULL;
  }
  if (bDestroyWindow) {
    bDestroyWindow = FALSE;
    DestroyWindow(hWnd); /* sends a WM_DESTROY msg */
  }
  if (iccontext != IC_RESULT_CONTEXT_INVALID) {
    IcDeregisterAccessory(iccontext);
    iccontext = IC_RESULT_CONTEXT_INVALID;
  }
}

long FAR PASCAL MainWndProc(HWND hWnd,
              UINT message,
              WPARAM wParam,
              LPARAM lParam)
{
  switch (message) {
  case (IC_MSGBASE+IC_SESSIONCLOSED):
    ICSSessionClosed(hWnd, HSESSION);
    break;

...
}
```

Terminating your Application

The sequence of messages during application termination can vary depending on who initiated the termination: the user, INFOConnect, the system or your application itself. For each scenario, the first message passed to your application is different. See the discussion after the code fragment for specific details about each scenario.

Close any open INFOConnect sessions and free any buffers before terminating your application. Notice the precautions taken in the code fragment below such as setting released resources to NULL.

The following scheme handles termination cleanly no matter how it is initiated.

```
long FAR PASCAL MainWndProc( HWND hWnd,
          UINT message,
          WPARAM wParam,
          LPARAM lParam)
{
 switch (message) {
 case WM_COMMAND:
  switch (wParam) {
  case IDM_EXIT:
   TerminateApplication(hWnd);
   break;
...
  break;

 case WM_CLOSE:
  TerminateApplication();
  break;

 case WM_DESTROY:
  PostQuitMessage(0);
  break;

 case (IC_MSGBASE+IC_SESSIONCLOSED):
  ICSSessionClosed(hWnd, HSESSION);
  break;

 case (IC_MSGBASE+IC_STATUS):
  ICSStatus(hWnd, HSESSION, ICRESULT);

 default:
  return(DefWindowProc(hWnd,message,wParam,lParam));
 }
 return(NULL);
}
```

```
void TerminateApplication(HWND hWnd)
{
  /*
    Clean up any INFOConnect resources still assigned.
  */
  static BOOL bDestroyWindow = TRUE;
  s.bSessionEst = FALSE;
  if (s.hSession != NULL_HIC_SESSION) {
    IcCloseSession(s.hSession);
  }
  if (s.hXmtBuf != NULL) {
    IcFreeBuffer(s.hXmtBuf);
    s.hXmtBuf = NULL;
  }
  if (s.hRcvBuf != NULL) {
    IcFreeBuffer(s.hRcvBuf);
    s.hRcvBuf = NULL;
  }
  if (bDestroyWindow) {
    bDestroyWindow = FALSE;
    DestroyWindow(hWnd); /* sends a WM_DESTROY msg */
  }
  if (iccontext != IC_RESULT_CONTEXT_INVALID) {
    IcDeregisterAccessory(iccontext);
    iccontext = IC_RESULT_CONTEXT_INVALID;
  }
}

void ICSSessionClosed(HWND hWnd, HIC_SESSION hSession)
{
  /*
    Calling TerminateApplication() from here results in an
    orderly termination if the user clears your application's
    session from the INFOConnect status window.
    If your application can stay active in this situation,
    don't call TerminateApplication() from here.
  */
  assert(hSession == s.hSession);
  s.hSession = NULL_HIC_SESSION;
  TerminateApplication(hWnd);
}
```

```
void ICSStatus(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{
  assert(hSession==s.hSession ||
     IC_GET_RESULT_TYPE(result)==IC_STATUS_COMMMGR);
  if (result==IC_CONTROL_ACTIVATE)
    SetFocus(hWnd);
  if ((result==IC_COMMMGR_QUERYEXIT) ||
     (result==IC_COMMMGR_QUERYSHUTDOWN))
    IcExitOk(TRUE);
  if (result==IC_COMMMGR_CANCELEXIT)
    ;/* Some other app responded QUERYEXIT/FALSE.
      INFOConnect won't exit after all.
     */
  if (result==IC_COMMMGR_EXIT)
    ;/* INFOConnect is really going away.
      All apps responded QUERYEXIT/TRUE.
     */
...
}
```

User-initiated termination of your application - WM_COMMAND

Most applications have a File-menu/Exit-option that allows the user to terminate the application by choosing the Exit option. This results in an WM_COMMAND message. Your application needs to be prepared to close INFOConnect sessions and free any associated buffers. The code fragment shown above accomplishes this by calling the TerminateApplication function.

INFOConnect-initiated termination - IC_SessionClosed

If the user clears the session associated with your application from the INFOConnect Manager window, INFOConnect sends your application an IC_SessionClosed message. To abort your application at this point, the recommended procedure is to call TerminateApplication from IC_SessionClosed processing. Do not call TerminateApplication from IC_SessionClosed processing if you want to keep your application active in this situation.

User-initiated termination of the INFOConnect Manager

If the user shuts down INFOConnect from the INFOConnect Manager window (ALT-F4 keystroke), a series of status messages are sent to all active INFOConnect applications. First your application receives IC_COMMMGR_QUERYEXIT. The application must call IcExitOk with either TRUE if it's OK to terminate or FALSE if its not OK to terminate. If it's OK, then INFOConnect will continue querying the other INFOConnect applications.

If an application refuses to shut down, all applications that had agreed to the shutdown are sent IC_COMMMGR_CANCELEXIT and may continue with normal execution. If all applications agree to the shutdown, the Manager then sends IC_COMMMGR_EXIT to confirm that all applications have agreed to close. As a final notice, the application will receive a terminate severity error, IC_ERROR_TERMINATE_EXIT. The application can temporarily delay closing the session if necessary.

System-initiated termination - WM_QUIT, WM_CLOSE, WM_DESTROY

Termination invoked by the system will result in WM_QUIT, WM_CLOSE, and WM_DESTROY messages. The recommended procedure is to call your local TerminateApplication function.

Application-initiated termination

Your application can force termination itself by calling TerminateApplication. The bDestroyWindow boolean in TerminateApplication is used as a sanity check to prevent the Windows DestroyWindow function from being called more than once. This is necessary because there are so many ways an application can be terminated; some of them can result in multiple calls on TerminateApplication.

Advanced Procedures for Windows Applications

Canceling Pending Requests

Many applications typically stay in receive mode so they're ready to respond to any messages from the partner activity. Occasionally, it may be necessary to cancel this outstanding receive request.

The IcLcl function selectively cancels pending actions. A parameter indicates what is to be canceled: the pending transmit request, pending receive request, or both. Your application should wait for an IC_LclResult message before accessing the datacomm buffers associated with the canceled actions. By waiting for this message, you are sure to process any IC_RcvDone or IC_XmtDone messages that were already in your message queue before the IcLcl was done.

```
/* The following definitions are in ICDEF.H      */
/* You do NOT need to define these in your application */

#define IC_LCL_RCV  1
#define IC_LCL_XMT  2
#define IC_LCL_RCVXMT (IC_LCL_RCV | IC_LCL_XMT)
#define IC_LCL_CLOSESESSION 4
```

Sample code

The following code cancels any pending receive or transmit requests.

```
icerror = IcLcl(hSession, IC_LCL_RCVXMT);
if IC_CHECK_RESULT_SEVERE(icerror) {
  IcDefaultErrorProc(hWnd, hSession, NULL, icerror);
```

Handling Data Communications Errors

Data communications errors show up as IC_XmtError and IC_RcvError messages in your window function. A good application will act on these messages. Often, you may just want to retry the request for some number of times.

The following code fragment will retry datacomm errors five times before displaying an error to the user. Two variables, *nXmtTries* and *nRcvTries*, are the basis of this technique. They are incremented when a request results in an error. They are set to zero after successful completion (IC_XmtDone and IC_RcvDone) indicating there is no longer an outstanding request.

```
#define MAXRETRIES 5

struct aSession {
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
  unsigned uBufsize;
  int nXmtTries;
  int nRcvTries;
} s;

void ICSXmtDone(HWND hWnd,
        HIC_SESSION hSession)
{
  NOREF(hWnd);
  assert(hSession==s.hSession);
  s.nXmtTries = 0; /* no outstanding transmits */
}
```

```
void ICSXmtError(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{
 /* Ignore warnings and informational errors,
   Xmt request still outstanding.
 */
 if (IC_CHECK_RESULT_SEVERE(result)) {
   assert(hSession==s.hSession);
   if (++(s.nXmtTries) > MAXRETRIES) {
     HandleIcError(hWnd,hSession,IC_XMTERROR,result);
     s.nXmtTries = 1;
   }
   if (s.bSessionEst) {
     /* Try again. No need to IC_TRANSACTION_BEGIN since its
       already been sent in WM_CHAR.
     */
     icerror = IcXmt(hSession, s.hXmtBuf,
           strlen(sNoteBuf));
     if IC_CHECK_RESULT_SEVERE(icerror) {
       HandleIcError(hWnd,hSession,NULL,icerror);
       s.nXmtTries = 0;
       IcSetStatus(s.hSession, IC_TRANSACTION_END);
     }
   }
   else
     s.nXmtTries = 0;
 }
}

void ICSRcvDone(HWND hWnd,
        HIC_SESSION hSession,
        HANDLE hBuffer,
        unsigned buflen)
{
 LPSTR buf;
 char sTitle[100];
 unsigned i;

 /* Display the received message */
 assert(hSession==s.hSession);
 s.nRcvTries = 0; /* no outstanding receives */
 buf = IcLockBuffer(hBuffer);
 assert(buf!=NULL);
 for (i=0; i < buflen && i < sizeof(sNoteBuf)-2; i++)
   sNoteBuf[i] = buf[i];
 sNoteBuf[i] = 0;
 IcUnlockBuffer(hBuffer);
 IcSetStatus(hSession, IC_TRANSACTION_END);
 LoadString(hInst, RCV_MSG_PREFIX, sTitle, sizeof(sTitle));
 MessageBox (hWnd,
        (LPSTR)sNoteBuf,
        sTitle,
        MB_ICONINFORMATION | MB_OK);
 s.nRcvTries = 1;
 icerror = IcRcv(hSession, s.hRcvBuf, s.uBufsize);
 if IC_CHECK_RESULT_SEVERE(icerror) {
   s.nRcvTries = 0;
   HandleIcError(hWnd,hSession,NULL,icerror);
 }
}
```

```
void ICSRcvError(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{

  /* Ignore warnings and informational errors,
    Rcv request still outstanding.
  */
  if (IC_CHECK_RESULT_SEVERE(result)) {
    /* Don't send IC_TRANSACTION_END since we keep trying until
      successful.
    */
    assert(hSession == s.hSession);
    if (++(s.nRcvTries) > MAXRETRIES) {
      HandleIcError(hWnd, hSession, IC_RCVERROR, result);
      s.nRcvTries = 1;
    }
    if (s.bSessionEst) {
      /* try receive again */
      icerror = IcRcv(hSession, s.hRcvBuf, s.uBufsize);
      if IC_CHECK_RESULT_SEVERE(icerror) {
        HandleIcError(hWnd, hSession, NULL, icerror);
        s.nRcvTries = 0;
      }
    }
    else
      s.nRcvTries = 0;
  }
}
```

Advanced Status and Error Handling

Before reading this discussion, you should be familiar with the definition of *context* given in Section 2, "An Introduction to INFOConnect Connectivity Services."

What is typedef IC_RESULT?

INFOConnect statuses and errors are defined with type IC_RESULT. An IC_RESULT type is a 'long' made up of three parts: a context, a type and a value. The formal names for these three are: IC_RESULT_CONTEXT, IC_RESULT_TYPE and IC_RESULT_VALUE.

The following macros are available for building and tearing down IC_RESULT types.

IC_MAKE_RESULT	Builds IC_RESULT from its 3 parts
IC_GET_RESULT_CONTEXT	Extracts 'context' from IC_RESULT
IC_GET_RESULT_TYPE	Extracts 'type' from IC_RESULT
IC_GET_RESULT_VALUE	Extracts 'value' from IC_RESULT

How do I test for library-specific statuses and errors?

If necessary, you can determine where a status or error is defined by extracting the context from the IC_RESULT. This might be useful when you are formatting your own error messages and want to include the name of the library that defines the error.

Note: *The library that defines a message is not necessarily the only library that uses or generates that message, although that is generally the case. Standard INFOConnect statuses and errors have a context of IC_RESULT_CONTEXT_STD and are defined in IcError.h and IcStatus.h.*

Each INFOConnect library or accessory that defines unique statuses and errors must provide a header file defining the types and values for its statuses and errors. The naming convention for the header file is to use the .HIC suffix for the header file. For example, TTY-specific definitions from IcTTY.dll are defined in IcTTY.hic.

What is the scope or visibility of statuses and errors?

Generally, statuses and errors are only 'seen' by the components in the current path: the application, service libraries and external interface. Status and error messages are not sent across the connection by INFOConnect except for accessories executing locally (invoked via IcOpenAccessory). One of the sample programs, CoupleS, is a service library that extends the scope of statuses and errors by encoding them and sending them across the connection.

What are these status messages telling me?

The following paragraphs give a brief description of several of the status messages listed in Appendix B of the *IDK Programming Reference Manual*.

IC_CONTROL_RCVAVAIL status

The IC_CONTROL_RCVAVAIL status is sent to the application when a message is available but the application isn't in receive mode and ready to get it. Although a terminal emulator generally returns to receive mode as quickly as possible after each host message, there might be times when an IC_CONTROL_RCVAVAIL status is sent to the terminal because another message is available, but the terminal is still in a local state.

IC_CONNECT_... statuses

These statuses originate with the external interface. They can be used by the application to show the user some kind of status about the datacomm connection.

The IC_CONNECT_ACTIVITY and IC_CONNECT_NOACTIVITY pair of statuses indicate the presence of line activity. The exact meaning of a *line* depends on the specific external interface, but generally a line carries much more than the traffic for just one session. Therefore, IC_CONNECT_ACTIVITY means that something is happening on the line, but not necessarily for your session. IC_CONNECT_NOACTIVITY indicates the absence of any line activity for some reasonable period of time (for example, 10 seconds). External interfaces must choose an appropriate time period so that the application is not flooded with nuisance statuses.

The IC_CONNECT_OPEN and IC_CONNECT_CLOSE pair of statuses indicate activity for a specific INFOConnect session, namely, the current one. Once again, the exact meaning varies with the external interface.

IC_CONTROL_ACTIVATE status

When a session with an INFOConnect terminal is minimized and focus is switched to the main INFOConnect window, the INFOConnect window shows the active sessions. By selecting the Goto button while the session for the INFOConnect terminal is selected, INFOConnect sends an IC_CONTROL_ACTIVATE status to the application requesting it to grab the input focus.

IC_LINESTATE_... statuses

Three line state statuses, IC_LINESTATE_XMT, IC_LINESTATE_RCV, and IC_LINESTATE_LCL, are intended to help terminal emulators tell the user about the current state of the line. Depending on the underlying libraries, calls to IcRcv and IcXmt are sometimes followed by IC_LINESTATE_RCV and IC_LINESTATE_XMT statuses. Also, IC_RCVDONE messages are sometimes followed by IC_LINESTATE_LCL statuses.

Using IC_STATUS_BUFFER extended status

When an application needs to exchange more information with an ICS library than IC_RESULT_VALUE can store, it can send a buffer of information with the IC_STATUS_BUFFER extended status. To accomplish this, a HIC_STATUSBUF buffer handle is assigned to the IC_RESULT_VALUE member of the IC_RESULT structure.

Extended statuses can be exchanged in two ways: synchronously and asynchronously. Refer to Section 7, "Writing INFOConnect Libraries for Windows 3.x" for detailed steps on exchanging extended statuses.

Encoding and Decoding

Some transports do not guarantee that all binary data streams can be safely sent across the connection (that is, SINFO.transparent=FALSE). Binary data must be encoded by the sender and decoded by the receiver. This requires close coordination between the workstation and host components; INFOConnect does not currently provide encoding and decoding services. A service library is a good place to implement the workstation side of this functionality.

Data Compression and Error Detection

INFOConnect does not currently provide routines for data compression and error detection. Error detection is generally the responsibility of the communications layers beneath the INFOConnect architecture. Data compression, on the other hand, is an ideal application for an INFOConnect service library.

Running with Old Versions of INFOConnect

Note: *IcWinApp has been coded to run with ICS Releases 2.0 and 3.0.*

The basic philosophy of INFOConnect version control is "Old applications must still run with new versions of the Manager, but new applications are not required to run with old Managers." In fact, the Manager normally refuses to run applications built with a version of the IDK that is newer than the Manager itself. The assumption is that new applications might make new API calls that an old Manager doesn't know about.

If you are upgrading an existing application to a new version of the IDK, and can function properly without making new INFOConnect API calls, this section describes how to build your application with the latest IDK and initialize with different versions of the Manager at run time. Your application must remember which version of the Manager is executing and only make appropriate API calls known by that version of the Manager.

The version of an application is normally "marked" on the initial call to IcInitIcs:

```
icerror = IcInitIcs(IC_VERSION_3_0, IC_REVISION_3_0);
if IC_CHECK_RESULT_SEVERE(icerror)
  // error processing. INFOConnect services are not available.
```

IC_VERSION_3_0 and IC_REVISION_3_0 are defined in IcDef.h. A new IC_VERSION_... is added at major release levels (Release 1.0, 2.0, 3.0). IC_REVISION_... has a finer granularity -- a new one is added each time the INFOConnect APIs are extended or changed.

If you compile your application using the 2.0 values for version and revision (IC_VERSION_2_0 and IC_REVISION_2_0) and attempt to run with INFOConnect 1.0, an error IC_ERROR_NEWVERSION is returned. However, your application can successfully run with INFOConnect 2.0 or 3.0.

The following code sample is suggested for initializing with the Manager:

```
int icversion; /* global variable */
IC_RESULT icerror;

if ((icerror=IcInitIcs(IC_VERSION_CHECK,IC_REVISION_1_0)) == IC_OK)
  icversion = IC_VERSION_1_0;
else if ((icerror=IcInitIcs(IC_VERSION_3_0,IC_REVISION_3_0)) == IC_OK)
  icversion = IC_VERSION_3_0;
else if ((icerror=IcInitIcs(IC_VERSION_2_0,IC_REVISION_2_0)) == IC_OK)
  icversion = IC_VERSION_2_0;
else {
  IcDefaultErrorProc(hWnd, NULL, NULL, icerror);
  /* INFOConnect services are unavailable!! */
}
```

At the end of this code fragment, *icversion* is set to the level of the Manager with which you are running.

CAUTION

Be careful to check *icversion* before making any API calls undefined in older versions of INFOConnect. Otherwise, you will encounter unexpected behavior.

The recommended way to isolate new API calls in your application follows:

```
if (icversion >= IC_VERSION_3_0) {
  /* call some new 3.0 INFOConnect API */
}
else {
  /* alternative action using pre 3.0 API calls */
}
```

Procedures for INFOConnect Accessories

INFOConnect applications that can be invoked and controlled by other INFOConnect applications are called accessories. INFOConnect accessories are written so that they can be used to build more sophisticated INFOConnect accessories. You communicate with an accessory through an INFOConnect session. It is easy and very useful to extend your INFOConnect application to become an accessory.

Calling INFOConnect Accessories

Two functions are available to invoke accessories: IcOpenAccessory and IcRunAccessory. IcOpenAccessory is the more powerful of the two; a connection is established between your application and the accessory using the Local external interface.

Sample code

```
#define HSESSION wParam
IC_RESULT icerror;
IC_SINFO sinfo;
HIC_SESSION hMyHostConnection, hMyAnsiSession;

/* Retrieve characteristics of the host connection */
icerror = IcGetSessionInfo( hMyHostConnection, &sinfo);
if IC_CHECK_RESULT_SEVERE(icerror {
  HandleIcError( hWnd, HSESSION, NULL, icerror);
  TerminateApplication();
}

/* Open an accessory named ANSI using pathname MYPATH */
icerror = IcOpenAccessory(hWnd, "ANSI", NULL,
        "MYPATH", &sinfo, &hMyAnsiSession);
if IC_CHECK_RESULT_SEVERE(icerror) {
  HandleIcError( hWnd, NULL, NULL, icerror);
  TerminateApplication();
}
```

Notice the sinfo structure required as one of the parameters on IcOpenAccessory. This code fragment assumes that a host connection has already been established. The session attributes of the host connection are retrieved and used to initialize the local connection to the ANSI accessory. For an example of using IcOpenAccessory in complete context, see the sample program, IcOpenAc. Although IcOpenAc uses the INFOConnect/XVT API, translating to the INFOConnect/Windows API is straightforward. Search for the function ic_open_accessory.

Making your Application an INFOConnect Accessory

Note: *See Section 6 of the IDK Programming Reference Manual for a complete list of requirements for accessories.*

All INFOConnect accessories are expected to do the following:

Parse the command line

All INFOConnect accessories should accept the -P and -L command line options for pathname and window location.

For a coding example of parsing the pathname, see function GetCmdLineOption in the sample program IcWinApp.

Display the INFOConnect session name

Where relevant, accessories should incorporate the INFOConnect session name in the window's title bar. This will help the user differentiate between multiple, active copies of an accessory.

The following code fragment was taken from the IcWinApp sample program.

```
void UpdateWindowName(HWND hWnd, LPSTR lpWindowName)
{
 /*
   Append the INFOConnect session name to the Window name
   and display it.
 */

 char SessionName[IC_MAXSESSIONIDSIZE];

 icerror = IcGetSessionID(s.hSession,
              SessionName,
              sizeof(SessionName));
 if IC_CHECK_RESULT_SEVERE(icerror)
   HandleIcError(hWnd, s.hSession, NULL, icerror);
 lstrcat(lpWindowName, (LPSTR)" - ");
 lstrcat(lpWindowName, SessionName);
 SetWindowText(hWnd, lpWindowName);
}
```

Handle the IC_CONTROL_ACTIVATE status message

The code necessary to support this status was covered earlier under *Basic status handling*. INFOConnect sends this status when the user presses the GoTo button on the INFOConnect window display of active sessions.

Registering and Deregistering as an accessory

Accessories that define statuses or errors must obtain a context by calling IcRegisterAccessory. See the IcRegisterAccessory and IcDeregisterAccessory calls in the IcWinApp sample program.

Provide a header file for accessory callers

Accessories must provide an .HIC header file that defines their identifying context string and any nonstandard, accessory-specific errors or statuses defined by the accessory. The compile-time name and value of the context string must be unique from all other INFOConnect accessories.

```
/*****************************************/
/* SAMPLE.HIC                           */
/*                                      */
/*****************************************/

#define SAMPLE_CONTEXTSTRING "SAMPLE"

/* status types - none defined */

/* error types - none defined */
```

Compiling

Note: *Before compiling and linking any INFOConnect applications, review the System Verification Checklist in the Installation section.*

Memory Models

The medium or small memory models are recommended for Windows-specific applications.

The *Windows 3.1 SDK Guide to Programming* covers the use of various memory models in Chapter 16, "More Memory Management." The INFOConnect architecture places no additional constraints on memory model usage.

Include files

```
#include <windows.h> /* must precede icwin.h */
#include <icwin.h>
```

C compiler options

The sample programs provided with the IDK Development Kit are built with the following options using the Microsoft C compiler.

```
cl -c -AM -Gsw -W3 -Oils -Zp -FPc ciowinapp.c
```

-c	Compile only - don't link
-AM	Medium memory model
-Gsw	(s) remove stack probes (w) compile for Windows
-W3	Generate warnings
-Oils	(i) enable intrinsic functions (l) enable loop optimization (s) favor code size (t) favor execution time (d) optional flag for debugging

Notes:

- *Choosing between (s) and (t) can have a significant impact on your application. You may want to experiment before settling on one of them.*
- *Using the (d) flag disables optimization and is recommended when using debuggers like CodeView. Some of the sample programs use the (d) flag in their make files. Don't forget to remove it when building the production version of your code.*

-Zp	(p) pack structures on 1 byte boundaries (i) optional flag for CodeView debugging
-FPc	Generate calls to the emulator floating-point library

Resource Files

Note: *Only INFOConnect accessories have resource file requirements. If your application is not called by other INFOConnect applications as an accessory, you are not required to have any of these sections in your resource file. However, you may still be interested in the version information resource.*

Applications use resource files to define things like menus, icons, stringtables, bitmaps, and so forth. INFOConnect accessories must also have an *INFOConnect RCDATA* section plus some additional strings included in the STRINGTABLE section of the resource file.

This portion of IcWinApp.h shows those definitions used in the RC resource file. You will need to provide similar local definitions in your application's header file.

```
/*****************************************************************/
/* ICWINAPP.H - C Header file                          */
/*                                   */
/*       Sample INFOConnect/Windows 3.x application    */
/*                                   */
/*****************************************************************/

#define APPNAME      IcWinApp
#define QAPPNAME     "IcWinApp"
#define QMARKETINGNAME "INFOConnect Sample Accessory"
#define QVENDOR      "Unisys"
#define QMODULEID    "ICWINAPP"

#define IC_ACCESSORYID    101
#define IC_ACCESSORYDESC  102
#define IC_VENDOR         103
```

Here is the beginning of IcWinApp.rc:

```
/*****************************************************************/
/* ICWINAPP.RC - Resource file                         */
/*                                   */
/*       Sample INFOConnect/Windows 3.x application    */
/*                                   */
/*****************************************************************/

#include <windows.h>
#include <icdef.h>
#include <icdict.h>
#include <icsample.hic>
#include "icwinapp.h"
```

IcDef.h and IcDict.h are INFOConnect header files containing general definitions needed in all INFOConnect resource files. IcWinApp.h contains local definitions specific to this application.

The INFOConnect resource is an RCDATA resource that points to all other INFOConnect-related resources. It specifies whether the component is an application or library, the vendor of the component, and so forth. See data type IC_RC_NODE in the *IDK Programming Reference Manual* for a detailed description of this resource.

```
INFOConnect RCDATA
BEGIN
 IC_VERSION_2_0,
 IC_REVISION_2_0,
 IC_ACCESSORY,   //IC_SERVICE vs IC_INTERFACE vs IC_ACCESSORY
 IC_HEADER_3_0,  //size of the INFOConnect RCDATA section
 0,        //link to dictionary tables
 IC_ACCESSORYID,  //STRINGTABLE link
 IC_ACCESSORYDESC, //STRINGTABLE link
 IC_VENDOR,     //STRINGTABLE link
 0,0,0,0,
 /* The following fields are new for 2.02 */
 IC_VERSION_3_0,
 IC_REVISION_3_0,
 0,        //reserved, must be zero
 0,        //reserved, must be zero
 0,0,       //no generic component value
 SAMPLE_ICWINAPP, UIS_SAMPLE  //Supplier number for IcWinApp
END
```

IC_VERSION_2_0 and IC_REVISION_2_0 refer to the minimum (or oldest) level of Connectivity Services that the accessory requires for proper operation. Older levels of Connectivity Services will refuse to run the application.

IC_ACCESSORY in the type field specifies that this component is an accessory.

IC_HEADER_3_0 designates the size of the header.

The next field is set to 0 since it is only needed by INFOConnect libraries.

IC_ACCESSORYID defines the context string for your accessory. This is the same string that is defined in your .HIC file and used by other INFOConnect applications and libraries to detect application-specific statuses and errors generated by your application. The accessory ID can be IC_MAXACCESSORYIDLEN characters long. The user sees the accessory ID in various Manager status lists and windows.

IC_ACCESSORYDESC is the string table number of the default description which is used when the accessory is installed.

IC_VENDOR is the string table number of the vendor identification string.

The next four fields pertain to INFOConnect libraries and are set to 0.

IC_VERSION_3_0 and IC_REVISION_3_0 specify the maximum (or latest) level of Connectivity Services with which the application was developed in order to take advantage of that level of ICS features.

Note: *Specify IC_VERSION_3_0 or IC_REVISION_3_0 only after all requirements in Chapter 10, "Converting from Release 2.0 to 3.0" have been completed. Specify IC_VERSION_2_0 and IC_REVISION_2_0 until the enhancements to the application have been completed.*

The next two fields are reserved fields and must be set to 0.

The next two entries are the LO, HI values of the generic component number defined by the IC_COMPONENT data type. Components that specify a non-zero generic IC_COMPONENT perform a specific function and must conform to the interface defined by the specific component's .HIC file. Generic component numbers are assigned by the Malvern Development Group.

The last two entries are the LO, HI values of the branded (supplier-specific) component number defined by the IC_COMPONENT data type. The branded IC_COMPONENT uniquely identifies the component. Refer to "Component Numbers" in Appendix A of the *IDK Programming Reference Manual* for more information on Component numbers.

Note: *The supplier-specific component numbers for the IDK samples are defined in IcSample.hic. If the component is defined as 0,0, a unique value will be assigned when the accessory ID is added to the INFOConnect configuration database.*

The version information can be viewed by examining the accessory. To examine the accessory, open the INFOConnect Accessories window by selecting Accessories from the Install menu. Next, select the accessory you want to view and then select the Examine button.

STRINGTABLE

You should already have a STRINGTABLE in your resource file. INFOConnect requires the STRINGTABLE to contain some additional entries with text for things like the application name, the vendor name, error messages and so forth.

```
STRINGTABLE DISCARDABLE
BEGIN
 IC_ACCESSORYID,  MYAPP_CONTEXTSTRING
 IC_ACCESSORYDESC, QMARKETINGNAME
 IC_VENDOR,    QVENDOR
END
```

Version information

Windows 3.1 has version checking capabilities available to installation programs. The INFOConnect Installation Manager does version checking when installing files that have version information in their resource section. The sample programs and makefiles use the technique shown below. Version information is compiled into the resource file if the WINSDKVER environment variable is defined and if WINSDKVER indicates that the Windows 3.1 SDK is available. The VER.H header file is not available in the Windows 3.0 SDK. The constants used on the right side of the #define statements (such as QMARKETINGNAME, QMODULEID) are generally defined in the .H file and specifically tailored for your component.

```
#ifdef WINSDKVER
#if (WINSDKVER >= 0x030a)
/* VER.DLL is only available in the Windows 3.1 SDK */
#include <ver.h>
#define VER_FILETYPE        VFT_APP
/*              VFT_APP for applications */
/*              VFT_DLL for libraries  */

#define VER_FILESUBTYPE     VFT_UNKNOWN
#define VER_FILEDESCRIPTION_STR QMARKETINGNAME
#define VER_INTERNALNAME_STR  QMODULEID
#define VER_FILEVERSION     NFILEVERSION
#define VER_PRODUCTVERSION   NPRODUCTVERSION
#define IC_FILEVERSION_STR   QVERSION
#define IC_PRODUCTVERSION_STR QVERSION
#define IC_LEGALCOPYRIGHT_STR QCOPYRIGHT
#define IC_LEGALTRADEMARKS_STR QTRADEMARK
#define IC_PRODUCTNAME_STR   QPRODUCTNAME
#define IC_COMPANYNAME_STR   QVENDOR
#include <icdef.rh>
#endif
#endif /* WINSDKVER */
```

The command line used to process the resource file for the sample Windows application is:.

```
rc -r icwinapp.rc
```

Linking

Note: *Before compiling and linking any INFOConnect applications, review the System Verification Checklist in the Installation section.*

Link in the usual way for Windows applications. The .EXE file is first built by the linker and then combined with the .RES resource file built earlier.

The sample Windows application, IcWinApp, was built with the following statements:

```
lnk @icwinapp.lnk
rc icwinapp.res icwinapp.exe
```

Linker (LNK) files

A typical LNK file (for example, IcWinApp.lnk) contains:

```
/noe/nod/map/line/co icwinapp.obj
icwinapp.exe
icwin Mlibcew libw
icwinapp.def
```

IcWinApp is the name of the application object file.

The /NOD option allows you to specify the Windows libraries explicitly. It tells the linker not to search any libraries specified in the object file to resolve external references.

The /NOE option allows you to override an object file built into the libraries with one of your own. This option prevents the linker from searching the extended dictionary, which is an internal list of symbol locations that the linker maintains.

Never use the /NOI option with Windows.

The /CO option is optionally used to prepare for debugging with the Microsoft CodeView debugger. Don't forget to remove it from the production version of your code.

IcWin.lib resolves all the references to the INFOConnect ICS functions. IcWin20.lib resolves all the references for the INFOConnect 2.0 and 3.0 ICS functions.

Mlibcew.lib is a Windows library for medium memory model.

Libw.lib is another Windows library.

Module definition (DEF) files

A typical module definition file (for example, IcWinApp.def) contains:

```
NAME     IcWinApp
DESCRIPTION 'Sample INFOConnect/Windows 3.0 Application'
EXETYPE   WINDOWS
STUB     'WINSTUB.EXE'
CODE     MOVEABLE DISCARDABLE
DATA     PRELOAD MOVEABLE MULTIPLE
SEGMENTS
;the entry point of the code (WinMain) must be PRELOAD
_TEXT    MOVEABLE DISCARDABLE PRELOAD
HEAPSIZE  1024
STACKSIZE  5120 ;recommended minimum for Windows

;All functions that will be called by any Windows routine
;MUST be exported.

EXPORTS
    MainWndProc  @1
    About      @2
    XmtDlg     @3
```

IcWinApp - a Sample Windows Application

The following is the complete source for IcWinApp, a Windows application that uses INFOConnect services. Most of the code fragments used as examples in the preceding discussion were taken from this program.

What does IcWinApp do?

IcWinApp is a simple communications program. It opens an INFOConnect session and allows the user to enter messages to be sent across the communications path using dialog boxes. Received messages are also displayed using dialog boxes.

Additional Features of IcWinApp

IcWinApp has been coded to run with ICS Releases 2.0 and 3.0.

Source file descriptions

IcWinApp.c	C-language source
IcWinApp.h	Header file
IcWinApp.rc	Resource file
IcWinApp.def	Module-definition file used to link
IcWinApp.ico	Icon file

All source files needed to build this application are provided with the IDK in the SAMPLE directory. To build the windows version of this application, do:

```
nmake -f makefile PROGRAM=icwinapp
```

Source listing for IcWinApp.C

```
/***********************************************************/
/* ICWINAPP.C                              */
/*                                         */
/*       Sample INFOConnect Windows application  */
/*                                         */
/***********************************************************/

/*

What is the purpose of this sample?

 1) Provide a template for application development.

   The source files used for this sample can be used as
   a starting point for INFOConnect/Windows cooperative
   applications.

   This sample can also be built to verify the proper
   installation of your development environment.

 2) Demonstrate techniques for managing datacomm buffers.

   This sample follows the recommended techniques for
   managing buffers during error handling, application
   termination, etc.

What does this sample program do when it is run?

  This application opens an INFOConnect session and displays
  received messages in a message box. If a key is pressed,
  the user is prompted for a message to transmit.

  Windows functions are used for all presentation services
  and INFOConnect is used for all data communications.

NOTE:

  This application was translated from IcXvtApp.
  You may find it useful to compare the two programs.

What are the #ifdef PING code sequences for?

  The PING compile time flag activates code that demonstrates
  the use of an IC_STATUS_BUFFER extended status message for
  communication between an accessory and library.

  The Reflect sample external interface library can be built
  to respond to the PING status message by simply adding a
    #define PING
  statement at the beginning of REFLECT.H

  The IcWinApp sample application can be built to emit
  the PING status message by adding a
    #define PING
  statement at the beginning of ICWINAPP.H
```

```
HISTORY:
  05/18/92 Converted to release 2.0
        Accessory registration added.
  08/31/92 PING code to demonstrate library-specific statuses
  10/02/92 Update termination processing to do DestroyWindow.
  11/2/93  Converted to Release 3.0 by adding support for
        IcRegisterMsgSession and IC_STATUS_TRANS.
        Continue to support ICS 2.0 through ProcessICS20Msg and
        RegisterICS20Msgs.
  01/26/93 Update PING code to demonstrate IC_STATUS_BUFFER
        extended status

*/

#define NOCOMM
#include <windows.h>
#include <icwin.h>
#include <assert.h>
#include <string.h>
#include <ctype.h>
#include "icwinapp.h"
#ifdef PING
  #include "reflect.hic"
#endif

/**********************************************************/
/* Global data and type declarations                      */
/**********************************************************/

HANDLE hInst;

struct aSession {
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
  BOOL bFocusNotify;
  BOOL bBlockingOn;
  unsigned uBufsize;
  int nXmtTries;
  int nRcvTries;
  IC_RESULT icstatus;
  #ifdef PING
    HIC_STATUSBUF hStatusBuf;
  #endif
}s;


IC_RESULT_CONTEXT iccontext=IC_RESULT_CONTEXT_INVALID;
IC_SINFO sinfo;
IC_RESULT icerror;
char sWindowName[256];
char sWindowHdr[256];
char sNoteBuf[256];
char sPrompt[256];
```

```
/* INFOConnect message numbers for ICS 2.0 support */
static unsigned IC_SessionEstablished;
static unsigned IC_SessionClosed;
static unsigned IC_Status;
static unsigned IC_XmtDone;
static unsigned IC_RcvDone;
static unsigned IC_XmtError;
static unsigned IC_RcvError;
static unsigned IC_Error;
static unsigned IC_NewPath;
static unsigned IC_LdResult;
static unsigned IC_StatusResult;
static unsigned IC_Timer;

unsigned ICSVersion = IC_VERSION_3_0;

#define MAXRETRIES 5
#define MAXBUFSIZE 4096

/* The following macros break down 'wParam' and 'lParam' into
   INFOConnect items.
*/
#define HSESSION ((HIC_SESSION)wParam)
#define ICRESULT ((IC_RESULT)lParam)
#define ICBUFFER ((HANDLE)HIWORD(ICRESULT))
#define ICLENGTH (LOWORD(ICRESULT))
#define ICCONTEXT (IC_GET_RESULT_CONTEXT(ICRESULT))
#define ICTYPE   (IC_GET_RESULT_TYPE(ICRESULT))
#define ICVALUE  (IC_GET_RESULT_VALUE(ICRESULT))

/************************************************************/
/* Local support routines                                    */
/************************************************************/

BOOL InitApplication(HANDLE hInstance);
BOOL InitInstance(HANDLE hInstance, int nCmdShow, LPSTR lpCmdLine);
int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
          LPSTR lpCmdLine, int nCmdShow);
long FAR PASCAL MainWndProc(HWND hWnd, UINT message,
               WPARAM wParam, LPARAM lParam);
BOOL FAR PASCAL About(HWND hDlg, UINT message,
             WPARAM wParam, LPARAM lParam);
BOOL FAR PASCAL XmtDlg(HWND hDlg, UINT message,
             WPARAM wParam, LPARAM lParam );
void HandleIcError(HWND hWnd, HIC_SESSION session,
          unsigned message, IC_RESULT icerror);
void UpdateWindowName(HWND hWnd, LPSTR lpWindowName);
void TerminateApplication(HWND hWnd);
IC_RESULT GetCmdlineOption(LPSTR sCmdLine, char option,
               char endDelimiter, LPSTR sValue,
               unsigned uValueSize);
void ICSSessionEstablished(HWND hWnd, HIC_SESSION hSession,
               IC_RESULT result);
void ICSSessionClosed(HWND hWnd, HIC_SESSION hSession);
void ICSStatus(HWND hWnd, HIC_SESSION hSession, IC_RESULT result);
void ICSXmtDone(HWND hWnd, HIC_SESSION hSession);
void ICSRcvDone(HWND hWnd, HIC_SESSION hSession, HANDLE hBuffer,
         unsigned buflen);
void ICSXmtError(HWND hWnd, HIC_SESSION hSession, IC_RESULT result);
```

```
void ICSRcvError(HWND hWnd, HIC_SESSION hSession, IC_RESULT result);
void ICSNewPath(HWND hWnd, HIC_SESSION hSession);
void ICSError(HWND hWnd, HIC_SESSION hSession, IC_RESULT result);
void ICSTimer(HWND hWnd, HIC_SESSION hSession);
void ICSStatusResult(HWND hWnd, HIC_SESSION hSession,
        IC_RESULT result);
void ICSLdResult(HWND hWnd, HIC_SESSION hSession);
long ProcessICS20Msg(HWND hWnd, UINT message,
        WPARAM wParam, LPARAM lParam);
BOOL RegisterICS20Msgs(void);
#ifdef PING
void PingReflect(HWND hWnd);
void DisplayPingAnswer(HWND hWnd, HIC_SESSION hSession,
         UINT uType, HIC_STATUSBUF hStatusBuf);
void PingStatus(HWND hWnd, HIC_SESSION hSession,
       IC_RESULT result);
void PingStatusResult(HWND hWnd, HIC_SESSION hSession,
         IC_RESULT result);
#endif

/***********************************************************/
/* Standard Windows functions                             */
/***********************************************************/

int PASCAL WinMain(HANDLE hInstance,
         HANDLE hPrevInstance,
         LPSTR lpCmdLine,
         int nCmdShow)
{
  MSG msg;

  if (!hPrevInstance)
    if (!InitApplication(hInstance))
      return (FALSE);

  if (!InitInstance(hInstance, nCmdShow, lpCmdLine))
    return (FALSE);

  while (GetMessage(&msg, NULL, NULL, NULL)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return (msg.wParam);
}
```

```
BOOL InitApplication(HANDLE hInstance)
{
  WNDCLASS wc;

  wc.style=NULL;
  wc.lpfnWndProc=MainWndProc;

  wc.cbClsExtra=0;
  wc.cbWndExtra=0;
  wc.hInstance=hInstance;
  wc.hIcon=LoadIcon(hInstance,QMYICON);
  wc.hCursor=LoadCursor(NULL,IDC_ARROW);
  wc.hbrBackground=GetStockObject(WHITE_BRUSH);
  wc.lpszMenuName= QMYMENUNAME;
  wc.lpszClassName=QMYCLASSNAME;

  if(!RegisterClass(&wc)){
    assert(FALSE);
    return(FALSE);
  }
  return(TRUE);
}

BOOL InitInstance(HANDLE hInstance,
          int nCmdShow,
          LPSTR lpCmdLine)
{
  HWND hWnd; /* Main window handle */
  char ICPath[IC_MAXPATHIDSIZE]="";
  char sAccessoryID[IC_MAXACCESSORYIDSIZE]={QAPPNAME};
  LPSTR pPathID;

  hInst=hInstance;

  LoadString(hInst,WINDOW_NAME,
        sWindowName,sizeof(sWindowName));
  hWnd=CreateWindow(
    QMYCLASSNAME,
    sWindowName,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    CW_USEDEFAULT,
    NULL,
    NULL,
    hInstance,
    NULL
  );

  if(!hWnd){
    assert(FALSE);
    return(FALSE);
  }

  ShowWindow(hWnd,nCmdShow);
  UpdateWindow(hWnd);      /* Sends WM_PAINT message */
```

```
/* Initialize INFOConnect Interfaces */
icerror = IcInitIcs(IC_VERSION_3_0,IC_REVISION_3_0);
if IC_CHECK_RESULT_SEVERE(icerror) {
  /* If you are not running on ICS 3.0, see if you are on ICS 2.0 */
  icerror = IcInitIcs(IC_VERSION_2_0,IC_REVISION_2_0);
  if IC_CHECK_RESULT_SEVERE(icerror) {
    LoadString(hInst, ICS_INIT_FAILED, sNoteBuf, sizeof(sNoteBuf));
    MessageBox(hWnd, sNoteBuf, QAPPNAME, MB_OK);
  } else {
    RegisterICS20Msgs();
    ICSVersion=IC_VERSION_2_0; /* for IcOpenSession/IcRegisterMsgSession */
  }
}

/* Register as an accessory. INFOConnect applications that
  aren't called as accessories don't have to do this.
*/
icerror = GetCmdlineOption(lpCmdLine, 'K', '',
            sAccessoryID, sizeof(sAccessoryID));
icerror = IcRegisterAccessory(sAccessoryID, 0, &iccontext);
if IC_CHECK_RESULT_SEVERE(icerror)
  goto done;

s.hSession = NULL_HIC_SESSION;
s.hXmtBuf = NULL;
s.hRcvBuf = NULL;
s.bSessionEst = FALSE;
s.bFocusNotify = FALSE;
s.bBlockingOn = FALSE;
s.uBufsize = 0;
s.nXmtTries = 0;
s.nRcvTries = 0;
s.icstatus = IC_OK;
#ifdef PING
  s.hStatusBuf = NULL_HIC_STATUSBUF;
#endif

/*
  Open an INFOConnect Session.
  Use path name from the command line, if specified.
  Otherwise, prompt the user.
*/
icerror = GetCmdlineOption(lpCmdLine, 'P', '',
            ICPath, sizeof(ICPath));
/*
  ICS Release 3.0 and above use IcRegisterMsgSession,
  so prompt for the path ID, if necessary.
*/
pPathID = ICPath;
if (icerror != IC_OK) {
  if (ICSVersion >= IC_VERSION_3_0)
    icerror = IcSelectPath(hWnd, NULL_HIC_CONFIG, 0,
              ICPath, sizeof(ICPath));
  else /* Pointer must be NULL for ICS 2.0 empty string. */
   pPathID = NULL;
}
```

```
 /*
  ICS Release 3.0 and above use IcRegisterMsgSession, so
  IcOpenSession needs to be called with a NULL window handle.
 */
 if (!IC_CHECK_RESULT_SEVERE(icerror) &&
   (icerror != IC_ERROR_CANCELOPEN)) {
  icerror = IcOpenSession(((ICSVersion < IC_VERSION_3_0) ? hWnd : NULL),
   pPathID, &s.hSession);
 }

 if (IC_CHECK_RESULT_SEVERE(icerror) ||
  (icerror == IC_ERROR_CANCELOPEN))
  goto done;

 if (ICSVersion >= IC_VERSION_3_0)
  icerror = IcRegisterMsgSession(s.hSession, hWnd, s.hSession,
   IC_MSGBASE, IC_LCLRESULT);

done:
 if IC_CHECK_RESULT_SEVERE(icerror) {
  HandleIcError(hWnd, s.hSession, NULL, icerror);
  if (iccontext != IC_RESULT_CONTEXT_INVALID) {
   IcDeregisterAccessory(iccontext);
   iccontext = IC_RESULT_CONTEXT_INVALID;
  }
  if (s.hSession != NULL_HIC_SESSION)
   IcCloseSession(s.hSession);
  return(FALSE);
 }
 return(TRUE);
}

long FAR PASCAL MainWndProc(HWND hWnd,
         UINT message,
         WPARAM wParam,
         LPARAM lParam)
{
 FARPROC lpProcAbout, lpXmtDlg;
 LPSTR buf;
 HDC hDC;
 PAINTSTRUCT ps;

 switch (message) {
 case WM_COMMAND:
  switch (wParam) {
  case IDM_ABOUT:
   lpProcAbout = MakeProcInstance(About, hInst);
   DialogBox(hInst,
       "AboutBox",
       hWnd,
       lpProcAbout);
   FreeProcInstance(lpProcAbout);
   break;

  case IDM_EXIT:
   TerminateApplication(hWnd);
   break;
```

```
    #ifdef PING
    case IDM_PING:
     PingReflect(hWnd);
     break;
    #endif

    default:
     return(DefWindowProc(hWnd,message,wParam,lParam));
    }
    break;

   case WM_CHAR:
    /*
     The user has pressed a key. Use a dialog box
     to get the message text to be transmitted.
     Put it in sNOTEBUF, then use 'lstrcpy' to move
     the message text into the transmit buffer.
    */
    if ((s.nXmtTries>0) || (!s.bSessionEst)) {
     LoadString(hInst, XMT_NOT_DONE,
         sNoteBuf, sizeof(sNoteBuf));
     MessageBox (hWnd,
         (LPSTR)sNoteBuf,
         QAPPNAME,
         MB_ICONEXCLAMATION | MB_OK);
    }
    else {
     lpXmtDlg = MakeProcInstance( XmtDlg, hInst);
     if (DialogBox( hInst, "XmtDlg", hWnd, lpXmtDlg)) {
      if ( (buf=IcLockBuffer( s.hXmtBuf)) == NULL) {
       assert( FALSE);
      }
      else {
       if (!s.bBlockingOn)
        lstrcat( sNoteBuf, "\r\n");
       lstrcpy( buf, (LPSTR)sNoteBuf);
       IcUnlockBuffer( s.hXmtBuf);
       IcSetStatus(s.hSession, IC_TRANSACTION_BEGIN);
       s.nXmtTries = 1;
       icerror = IcXmt( s.hSession, s.hXmtBuf,
            strlen(sNoteBuf));
       if IC_CHECK_RESULT_SEVERE(icerror) {
        s.nXmtTries = 0;
        IcSetStatus(s.hSession, IC_TRANSACTION_END);
        HandleIcError( hWnd, s.hSession,
             NULL, icerror);
       }
      }
     }
     FreeProcInstance( lpXmtDlg);
    }
    break;

   case WM_PAINT:
    hDC = BeginPaint( hWnd, &ps);
    LoadString( hInst, WINDOW_HDR,
        sNoteBuf, sizeof(sNoteBuf));
    TextOut( hDC, 10, 10, sNoteBuf, strlen(sNoteBuf));
    EndPaint( hWnd, &ps);
    break;
```

```
case WM_ACTIVATE:
 /*
   'wParam' indicates whether we are 'activating'
   or 'deactivating' a window.
 */
 if ((s.bFocusNotify) && (s.bSessionEst)) {
   if (wParam)
     s.icstatus = IC_REACTIVATE_ON;
   else
     s.icstatus = IC_REACTIVATE_OFF;
   icerror=IcSetStatus(s.hSession,s.icstatus);
   if IC_CHECK_RESULT_SEVERE(icerror) {
     s.icstatus = IC_OK;
     s.bFocusNotify = FALSE;
     HandleIcError( hWnd, s.hSession, NULL, icerror);
     s.bFocusNotify = TRUE;
   }
 }
 return (DefWindowProc(hWnd, message, wParam, lParam));

case WM_CLOSE:
 TerminateApplication(hWnd);
 break;

case WM_DESTROY:
 PostQuitMessage(0); /* sends a WM_QUIT msg */
 break;

 /**************************/
 /* INFOConnect messages */
 /**************************/

case (IC_MSGBASE+IC_SESSIONESTABLISHED):
 ICSSessionEstablished(hWnd, HSESSION, ICRESULT);
 break;

case (IC_MSGBASE+IC_SESSIONCLOSED):
 ICSSessionClosed(hWnd, HSESSION);
 break;

case (IC_MSGBASE+IC_STATUS):
 ICSStatus(hWnd, HSESSION, ICRESULT);
 break;

case (IC_MSGBASE+IC_XMTDONE):
 ICSXmtDone(hWnd, HSESSION);
 break;

case (IC_MSGBASE+IC_RCVDONE):
 ICSRcvDone(hWnd, HSESSION, ICBUFFER, ICLENGTH);
 break;

case (IC_MSGBASE+IC_XMTERROR):
 ICSXmtError(hWnd, HSESSION, ICRESULT);
 break;

case (IC_MSGBASE+IC_RCVERROR):
 ICSRcvError(hWnd, HSESSION, ICRESULT);
 break;
```

```
case (IC_MSGBASE+IC_NEWPATH):
  ICSNewPath(hWnd, HSESSION);
  break;

 case (IC_MSGBASE+IC_ERROR):
  ICSError(hWnd, HSESSION, ICRESULT);
  break;

 case (IC_MSGBASE+IC_TIMER):
  ICSTimer(hWnd, HSESSION);
  break;

 case (IC_MSGBASE+IC_STATUSRESULT):
  ICSStatusResult(hWnd, HSESSION, ICRESULT);
  break;

 case (IC_MSGBASE+IC_LCLRESULT):
  ICSLclResult(hWnd, HSESSION);
  break;

 default:
  if ((ICSVersion >= IC_VERSION_3_0) ||
    (! ProcessICS20Msg(hWnd, message, wParam, lParam)))
   return(DefWindowProc(hWnd, message, wParam, lParam));
 }
 return (NULL);
}

/************************************************************/
/* Local functions                                          */
/************************************************************/

BOOL FAR PASCAL About(HWND  hDlg,
          UINT  message,
          WPARAM wParam,
          LPARAM lParam)
{
 NOREF(lParam);
 switch (message) {
  case WM_INITDIALOG:
   return (TRUE);

  case WM_COMMAND:
   if (wParam == IDOK
    || wParam == IDCANCEL) {
    EndDialog(hDlg, TRUE);
    return (TRUE);
   }
   break;
 }
 return (FALSE); /* Didn't process a message */
}
```

```
BOOL FAR PASCAL XmtDlg(HWND  hDlg,
          UINT  message,
          WPARAM wParam,
          LPARAM lParam )
{
 /*
  This is the dialog callback function used when the user
  wants to enter a message to transmit.
 */

 NOREF(lParam);
 switch (message) {
  case WM_COMMAND:
   switch (wParam) {
    case IDOK:
     GetDlgItemText(hDlg,
           IDC_XMTTEXT,
           sNoteBuf,
           sizeof(sNoteBuf));
     EndDialog(hDlg, TRUE );
     return(TRUE);

    case IDCANCEL:
     EndDialog(hDlg, FALSE );
     return(TRUE);
   }
   break;
  case WM_INITDIALOG:
   LoadString(hInst, DEFAULT_XMT_MSG,
       sPrompt, sizeof(sPrompt));
   SetDlgItemText(hDlg, IDC_XMTTEXT, sPrompt);
   SendDlgItemMessage(hDlg, IDC_XMTTEXT, EM_SETSEL,
          NULL, MAKELONG(0, 0x7fff));
   SetFocus(GetDlgItem(hDlg, IDC_XMTTEXT));
   break;
 }
 return(FALSE);
}

void TerminateApplication(HWND hWnd)
{
 /*
  Clean up any INFOConnect resources still assigned.
 */
 static BOOL bDestroyWindow = TRUE;

 s.bSessionEst = FALSE;
 if (s.hSession != NULL_HIC_SESSION) {
  IcCloseSession(s.hSession);
 }
 if (s.hXmtBuf != NULL) {
  IcFreeBuffer(s.hXmtBuf);
  s.hXmtBuf = NULL;
 }
 if (s.hRcvBuf != NULL) {
  IcFreeBuffer(s.hRcvBuf);
  s.hRcvBuf = NULL;
 }
```

```
if (bDestroyWindow) {
   bDestroyWindow = FALSE;
   DestroyWindow(hWnd); /* sends a WM_DESTROY msg */
  }
  if (iccontext != IC_RESULT_CONTEXT_INVALID) {
   IcDeregisterAccessory(iccontext);
   iccontext = IC_RESULT_CONTEXT_INVALID;
  }
}

void UpdateWindowName(HWND hWnd, LPSTR lpWindowName)
{
  /*
   Append the INFOConnect session name to the Window name
   and display it.
  */

  char SessionName[IC_MAXSESSIONIDSIZE];

  icerror = IcGetSessionID(s.hSession,
              SessionName,
              sizeof(SessionName));
  if IC_CHECK_RESULT_SEVERE(icerror)
   HandleIcError(hWnd, s.hSession, NULL, icerror);
  lstrcat(lpWindowName, (LPSTR)" - ");
  lstrcat(lpWindowName, SessionName);
  SetWindowText(hWnd, lpWindowName);
}

void HandleIcError(HWND      hWnd,
          HIC_SESSION session,
          unsigned  message,
          IC_RESULT  icerror)
{
  /*
   Pass all INFOConnect errors back to INFOConnect
   for the default action.

   If you want to format and display the error
   message yourself, use IcGetString instead
   of IcDefaultErrorProc.


  */
  /*
   ICS 2.0 default error procedure needs the RegisterWindowMessage
   message value. Since the SessionEstablished message is the only
   one that ICS 2.0 acts on, we can pass that here when necessary.
   For ICS 3.0 and greater, the default error procedure understands
   the defined message indices defined in icdef.h.
  */
  if ((ICSVersion < IC_VERSION_3_0) && (message==IC_SESSIONESTABLISHED))
   message=IC_SessionEstablished;
  IcDefaultErrorProc(hWnd,session,message,icerror);
  if (IC_GET_RESULT_TYPE(icerror) >= IC_ERROR_TERMINATE)
   s.bSessionEst = FALSE;
}
```

```
#define scan_blanks(ptr,len)            \
      while ((len > 0) && (*ptr == ' ')) {\
        ptr++;                \
        len–;                 \
      }

IC_RESULT GetCmdlineOption (LPSTR   sCmdLine,
                char   option,
                char   endDelimiter,
                LPSTR  sValue,
                unsigned uValueSize)
{
  /* Returns IC_OK if and only if the 'option'
    has been specified in the command line.
    'option' is case-insensitive.
    For IC_OK, the buffer pointed to by 'sValue', and
    whose size (including '\0') is 'uValueSize', will
    return the null terminated value, otherwise
    the buffer is not affected.

    Return values:
      IC_ERROR_NOFIND if 'option' is not found.
      IC_ERROR_TRUNCATED if destination buffer is too small.
  */
  IC_RESULT ok;
  LPSTR   ptr;
  unsigned len;

  ok = IC_ERROR_NOFIND;

  if (sCmdLine != NULL) {
    ptr = sCmdLine;
    len = lstrlen (sCmdLine);
    while (len > 0) {
      if (*ptr == '-') {
        ptr++;
        len–;
        if ((len > 0) &&
          ((*ptr == (char)tolower(option)) ||
           (*ptr == (char)toupper(option)))) {
          ptr++;
          len–;
          scan_blanks(ptr,len);
          while ((len > 0) &&
              (*ptr != endDelimiter) &&
              (uValueSize > 1) ) {
            ok = IC_OK;
            *sValue = *ptr;
            sValue++;
            uValueSize–;
            if (uValueSize == 0)
              return IC_ERROR_TRUNCATED;
            ptr++;
            len–;
          }
          *sValue = '\0';
        }
```

```
        else { /* skip over unwanted option string */
          scan_blanks(ptr, len);
          while ((len > 0) && (*ptr != endDelimiter)) {
            ptr++;
            len--;
          }
        }
      }
      else {
        ptr++;
        len--;
      }
    }
  }
  return ok;
}

void ICSSessionEstablished(HWND hWnd,
                HIC_SESSION hSession,
                IC_RESULT result)

{
  assert(hSession==s.hSession); /* bad session */
  if IC_CHECK_RESULT_SEVERE(result) {
    HandleIcError(hWnd,hSession,IC_SESSIONESTABLISHED,result);
    IcCloseSession(hSession);
  }
  else {
    icerror = IcGetSessionInfo(hSession, &sinfo);
    if IC_CHECK_RESULT_SEVERE(icerror)
      HandleIcError(hWnd,hSession,NULL,icerror);
    s.bFocusNotify = sinfo.focus_notify;
    s.bBlockingOn = sinfo.block_mode;
    UpdateWindowName(hWnd, sWindowName);

    /* Allocate INFOConnect buffers. */
    s.uBufsize=min((unsigned)sinfo.max_size,MAXBUFSIZE);
    s.hXmtBuf = IcAllocBuffer(s.uBufsize);
    s.hRcvBuf = IcAllocBuffer(s.uBufsize);
    assert(s.hXmtBuf!=NULL);
    assert(s.hRcvBuf!=NULL);
    s.bSessionEst = TRUE;
    IcSetStatus(hSession, IC_TRANSACTION_ON);
    s.nRcvTries = 1;
    icerror = IcRcv(hSession, s.hRcvBuf, s.uBufsize);
    if IC_CHECK_RESULT_SEVERE(icerror) {
      s.nRcvTries = 0;
      HandleIcError(hWnd,hSession,NULL,icerror);
    }
  }
}
```

```
void ICSSessionClosed(HWND hWnd,
          HIC_SESSION hSession)
{
 /*
   Calling TerminateApplication() from here results in an
   orderly termination if the user clears your application's
   session from the INFOConnect status window.
   If your application can stay active in this situation,
   don't call TerminateApplication() from here.
 */
 assert(hSession==s.hSession);
 s.hSession = NULL_HIC_SESSION;
 TerminateApplication(hWnd);
}

void ICSStatus(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{
 assert(hSession==s.hSession ||
     IC_GET_RESULT_TYPE(result) == IC_STATUS_COMMMGR);
 if (result==IC_CONTROL_ACTIVATE)
   SetFocus(hWnd);
 if ((result==IC_COMMMGR_QUERYEXIT) ||
   (result==IC_COMMMGR_QUERYSHUTDOWN))
   IcExitOk(TRUE);
 if (result==IC_COMMMGR_CANCELEXIT)
   ; /* Some other app responded QUERYEXIT/FALSE.
      INFOConnect won't exit after all.
    */
 if (result==IC_COMMMGR_EXIT)
   ; /* INFOConnect is really going away.
      All apps responded QUERYEXIT/TRUE.
    */
 #ifdef PING
   PingStatus(hWnd, hSession, result);
 #endif
}

void ICSXmtDone(HWND hWnd,
         HIC_SESSION hSession)
{
 NOREF(hWnd);
 assert(hSession==s.hSession);
 s.nXmtTries = 0; /* no outstanding transmits */
}
```

```
void ICSRcvDone(HWND hWnd,
        HIC_SESSION hSession,
        HANDLE hBuffer,
        unsigned buflen)
{
 LPSTR buf;
 char sTitle[100];
 unsigned i;

 /* Display the received message */
 assert(hSession==s.hSession);
 s.nRcvTries = 0; /* no outstanding receives */
 buf = IcLockBuffer(hBuffer);
 assert(buf!=NULL);
 for (i=0; i < buflen && i < sizeof(sNoteBuf)-2; i++)
  sNoteBuf[i] = buf[i];
 sNoteBuf[i] = 0;
 IcUnlockBuffer(hBuffer);
 IcSetStatus(hSession, IC_TRANSACTION_END);
 LoadString(hInst, RCV_MSG_PREFIX, sTitle, sizeof(sTitle));
 MessageBox(hWnd,
        (LPSTR)sNoteBuf,
        sTitle,
        MB_ICONINFORMATION | MB_OK);
 s.nRcvTries = 1;
 icerror = IcRcv(hSession, s.hRcvBuf, s.uBufsize);
 if IC_CHECK_RESULT_SEVERE(icerror) {
  s.nRcvTries = 0;
  HandleIcError(hWnd,hSession,NULL,icerror);
 }
}

void ICSXmtError(HWND hWnd,
         HIC_SESSION hSession,
         IC_RESULT result)
{

 /* Ignore warnings and informational errors,
  Xmt request still outstanding.
 */
 if (IC_CHECK_RESULT_SEVERE(result)) {
  assert(hSession==s.hSession);
  if (++(s.nXmtTries) > MAXRETRIES) {
   HandleIcError(hWnd,hSession,IC_XMTERROR,result);
   s.nXmtTries = 1;
  }
  if (s.bSessionEst) {
   /* Try again. No need to IC_TRANSACTION_BEGIN since its
    already been sent in WM_CHAR.
   */
   icerror = IcXmt(hSession, s.hXmtBuf,
          strlen(sNoteBuf));
   if IC_CHECK_RESULT_SEVERE(icerror) {
    HandleIcError(hWnd,hSession,NULL,icerror);
    s.nXmtTries = 0;
    IcSetStatus(s.hSession, IC_TRANSACTION_END);
   }
  }
  else
   s.nXmtTries = 0;
 }
}
```

```
void ICSRcvError(HWND hWnd,
        HIC_SESSION hSession,
        IC_RESULT result)
{

 /* Ignore warnings and informational errors,
   Rcv request still outstanding.
 */
 if (IC_CHECK_RESULT_SEVERE(result)) {
   /* Don't send IC_TRANSACTION_END since we keep trying until
     successful.
   */
   assert(hSession==s.hSession);
   if (++(s.nRcvTries) > MAXRETRIES) {
     HandleIcError(hWnd,hSession,IC_RCVERROR,result);
     s.nRcvTries = 1;
   }
   if (s.bSessionEst) {
     /* try receive again */
     icerror = IcRcv(hSession,s.hRcvBuf,s.uBufsize);
     if IC_CHECK_RESULT_SEVERE(icerror) {
       HandleIcError(hWnd,hSession,NULL,icerror);
      s.nRcvTries = 0;
     }
   }
   else
     s.nRcvTries = 0;
 }
}

void ICSNewPath(HWND hWnd,
        HIC_SESSION hSession)
{
 NOREF(hWnd);
 assert(hSession==s.hSession);
}

void ICSError(HWND hWnd,
       HIC_SESSION hSession,
       IC_RESULT result)
{
 NOREF(hWnd);
 assert(hSession==s.hSession);
 HandleIcError(hWnd,hSession,IC_ERROR,result);
}

void ICSTimer(HWND hWnd,
       HIC_SESSION hSession)
{
 NOREF(hWnd);
 assert(hSession==s.hSession);
}
```

```
void ICSStatusResult(HWND hWnd,
          HIC_SESSION hSession,
          IC_RESULT result)
{
 NOREF(hWnd);
 NOREF(result);
 assert(hSession==s.hSession);
 #ifdef PING
   PingStatusResult(hWnd,hSession,result);
 #endif
 s.icstatus = IC_OK;
 }

void ICSLdResult(HWND hWnd,
        HIC_SESSION hSession)
{
 NOREF(hWnd);
 assert(hSession==s.hSession);
}

/* Support ICS 2.0 */

BOOL RegisterICS20Msgs (void)
{
#define REGISTERWINDOWMESSAGE(n) (n=RegisterWindowMessage(#n))

 /* Register INFOConnect message numbers */
 if ((!REGISTERWINDOWMESSAGE(IC_SessionEstablished)) ||
    (!REGISTERWINDOWMESSAGE(IC_SessionClosed)) ||
    (!REGISTERWINDOWMESSAGE(IC_Status)) ||
    (!REGISTERWINDOWMESSAGE(IC_XmtDone)) ||
    (!REGISTERWINDOWMESSAGE(IC_RcvDone)) ||
    (!REGISTERWINDOWMESSAGE(IC_XmtError)) ||
    (!REGISTERWINDOWMESSAGE(IC_RcvError)) ||
    (!REGISTERWINDOWMESSAGE(IC_NewPath)) ||
    (!REGISTERWINDOWMESSAGE(IC_Error)) ||
    (!REGISTERWINDOWMESSAGE(IC_Timer)) ||
    (!REGISTERWINDOWMESSAGE(IC_StatusResult)) ||
    (!REGISTERWINDOWMESSAGE(IC_LdResult)) ) {
   assert(FALSE);
   return(FALSE);
 }
 return(TRUE);
}
```

```
long ProcessICS20Msg( HWND hWnd,
        UINT message,
        WPARAM wParam,
        LPARAM lParam)
{
 if (message==IC_SessionEstablished) {
  ICSSessionEstablished(hWnd, HSESSION, ICRESULT);
 }
 else if (message==IC_SessionClosed) {
  ICSSessionClosed(hWnd, HSESSION);
 }
 else if (message==IC_Status) {
  ICSStatus(hWnd, HSESSION, ICRESULT);
 }
 else if (message==IC_XmtDone) {
  ICSXmtDone(hWnd, HSESSION);
 }
 else if (message==IC_RcvDone) {
  ICSRcvDone(hWnd, HSESSION, ICBUFFER, ICLENGTH);
 }
 else if (message==IC_XmtError) {
  ICSXmtError(hWnd, HSESSION, ICRESULT);
 }
 else if (message==IC_RcvError) {
  ICSRcvError(hWnd, HSESSION, ICRESULT);
 }
 else if (message==IC_NewPath) {
  ICSNewPath(hWnd, HSESSION);
 }
 else if (message==IC_Error) {
  ICSError(hWnd, HSESSION, ICRESULT);
 }
 else if (message==IC_Timer) {
  ICSTimer(hWnd, HSESSION);
 }
 else if (message==IC_StatusResult) {
  ICSStatusResult(hWnd, HSESSION, ICRESULT);
 }
 else if (message==IC_LdResult) {
  ICSLdResult(hWnd, HSESSION);
 } else
  return( DefWindowProc(hWnd, message, wParam, lParam));
 return( NULL);
}
```

```
#ifdef PING
void PingReflect(HWND hWnd)
{
 LPIC_STATUSBUF lpStatusBuf=NULL;
 LPSTR lpData=NULL;
 IC_RESULT_CONTEXT LibContext=IC_RESULT_CONTEXT_INVALID;

 if (s.hStatusBuf) {
  MessageBox(hWnd,"Ping already activated",QAPPNAME,MB_OK);
  return;
 }
 icerror=IcGetContext(REFLECT_CONTEXTSTRING,&LibContext);
 if IC_CHECK_RESULT_SEVERE(icerror) {
  MessageBox(hWnd,"Reflect library not active",QAPPNAME,MB_OK);
  if (icerror!=IC_CONTEXTSTRING_NOT_FOUND)
   HandleIcError(hWnd,s.hSession,NULL,icerror);
 }
 else {
  s.hStatusBuf=IcAllocBuffer(sizeof(IC_STATUSBUF)+256);
  assert(s.hStatusBuf);
  lpStatusBuf=(LPIC_STATUSBUF)IcLockBuffer(s.hStatusBuf);
  assert(lpStatusBuf);
  lpStatusBuf->icstatus=IC_MAKE_RESULT(LibContext,
                    REFLECT_STATUS_PING,
                    0);
  lpStatusBuf->icerror=IC_OK;
  lpStatusBuf->uBufSize=256;
  lpData=(LPSTR)(lpStatusBuf+1);
  lstrcpy(lpData,"Hello?");
  lpStatusBuf->uDataSize=lstrlen(lpData);
  IcUnlockBuffer(s.hStatusBuf);
  s.icstatus=IC_MAKE_RESULT(IC_RESULT_CONTEXT_STD,
              IC_STATUS_BUFFER,
              s.hStatusBuf);
  icerror=IcSetStatus(s.hSession,s.icstatus);
  if IC_CHECK_RESULT_SEVERE(icerror)
   HandleIcError(hWnd,s.hSession,NULL,icerror);
 }
}
```

```
void DisplayPingAnswer(HWND        hWnd,
            HIC_SESSION  hSession,
            UINT       uType,
            HIC_STATUSBUF hStatusBuf)
{
 LPIC_STATUSBUF  lpStatusBuf=NULL;
 LPSTR        lpData=NULL;
 IC_RESULT_TYPE  sbtype;
 char         tracemsg[IC_MAXSTRINGLENGTH];

 lpStatusBuf=(LPIC_STATUSBUF)IcLockBuffer(hStatusBuf);
 if (lpStatusBuf==NULL) {
  lstrcpy(tracemsg,"lpStatusBuf was NULL");
  IcMgrTraceBuffer(iccontext,hSession,uType,
           QAPPNAME".PingStatus",
           tracemsg,lstrlen(tracemsg));
  return;
 }
 if IC_CHECK_RESULT_SEVERE(lpStatusBuf->icerror) {
  HandleIcError(hWnd,hSession,IC_STATUS,lpStatusBuf->icerror);
  MessageBox(hWnd,"Reflect library returned an error to PING",
        QAPPNAME,MB_OK);
 }
 else {
  sbtype=IC_GET_RESULT_TYPE(lpStatusBuf->icstatus);
  if (sbtype==REFLECT_STATUS_PING) {
   lpData=(LPSTR)(lpStatusBuf+1);
   MessageBox(hWnd,lpData,"Answer to your ping",MB_OK);
  }
 }
 IcUnlockBuffer(hStatusBuf);
}

void PingStatus (HWND      hWnd,
        HIC_SESSION hSession,
        IC_RESULT  result)
{
 /* Example of library-specific status message */

 HIC_STATUSBUF   hStatusBuf=NULL_HIC_STATUSBUF;
 LPSTR        lpData=NULL;
 IC_RESULT      icerror=IC_OK;
 IC_RESULT_CONTEXT iccontext;
 IC_RESULT_TYPE  ictype;
 IC_RESULT_VALUE  icvalue;

 iccontext=IC_GET_RESULT_CONTEXT(result);
 ictype=IC_GET_RESULT_TYPE(result);
 icvalue=IC_GET_RESULT_VALUE(result);
 if ((iccontext==IC_RESULT_CONTEXT_STD) &&
   (ictype==IC_STATUS_BUFFER)) {
  hStatusBuf=(HIC_STATUSBUF)icvalue;
  if (hStatusBuf!=s.hStatusBuf)
   /* This is not the extended status buffer IcWinApp */
   /* allocated or it was released             */
   return;
 }
```

```
  else {
    /* This is not an extended status buffer */
    return;
  }
  DisplayPingAnswer(hWnd, hSession, IC_STATUS, hStatusBuf);
  IcFreeBuffer(hStatusBuf);
  s.hStatusBuf = NULL_HIC_STATUSBUF;
  s.icstatus = IC_OK;
}

void PingStatusResult (HWND      hWnd,
        HIC_SESSION hSession,
        IC_RESULT  result)
{
  #define PENDINGSTATUSTYPE   (IC_GET_RESULT_TYPE(s.icstatus))

  /* check to see if we are waiting for a Ping extended status */
  if ((PENDINGSTATUSTYPE != IC_STATUS_BUFFER) ||
     (s.hStatusBuf == NULL_HIC_STATUSBUF))
    return;

  if (IC_CHECK_RESULT_SEVERE(result)) {
    MessageBox(hWnd, "Reflect library did not answer PING",
        QAPPNAME, MB_OK);
    HandleIcError(hWnd, hSession, IC_STATUSRESULT, result);
  }
  else
    DisplayPingAnswer(hWnd, hSession,
         IC_STATUSRESULT, s.hStatusBuf);
  IcFreeBuffer(s.hStatusBuf);
  s.hStatusBuf = NULL_HIC_STATUSBUF;
  s.icstatus = IC_OK;
}
#endif
```

CoupleW - a Windows Application that Connects Two INFOConnect Sessions

The CoupleW application opens two INFOConnect sessions and routes all traffic from one to the other. It is the counterpart to the LOCAL external interface which connects two sessions at the external interface layer. CoupleW prompts the user for two pathnames unless they are passed in on the command line like this:

```
couplew.exe -ppathABC -ppathDEF
```

What is CoupleW used for?

There are numerous uses for CoupleW, some of which include:

- CoupleW can be used as the skeleton for a server application that connects pairs of INFOConnect paths.
- Working with INFOConnect accessories (using IcOpenAccessory) often involves connecting two sessions as in CoupleW; one session is for a host connection and the other is for the accessory connection. The IcOpenAc sample program further demonstrates this.

How does CoupleW work?

CoupleW opens both sessions, allocates a single receive buffer for each and goes into receive mode for each session. When session A gets an IC_RcvDone event, CoupleW transmits the received buffer on session B. When session B gets the IC_XmtDone event, CoupleW goes back into receive mode on session A. Similar processing occurs when session B gets a IC_RcvDone event.

Source file descriptions

CoupleW.c	C-language source
CoupleW.h	Header file
CoupleW.rc	Resource file
CoupleW.def	Module-definition file used to link
CoupleW.ico	Icon file

All source files needed to build CoupleW are provided with the IDK in the SAMPLE directory. To build CoupleW, do:

```
nmake -f makefile PROGRAM=couplew
```

Section 4
Writing INFOConnect/XVT Applications

Note: *The IDK is compatible with XVT 2.0, 3.01 and 3.02, however, the XVT toolkit changed substantially between XVT 2.0 and 3.0x. The sample code fragments shown in this document are based on XVT 3.0x.*

Writing an XVT application that uses INFOConnect Connectivity Services is like writing a normal XVT application enhanced by extra events, data structures and functions that are added by running IcXvtMod.exe. It is assumed you are familiar with developing standard XVT applications.

This section leads you through the development of a simple XVT application that uses INFOConnect Connectivity Services in addition to the presentation services offered by XVT. All source files necessary to build IcXvtApp are provided with the IDK.

An XVT application is made up of several source files, some of which include:

a C-language source (.C) file
a header (.H) file
a resource (.URL) file

The following are some of the basic data communications tasks you will encounter as you write the C-language source (.C) file.

Basic procedures for XVT Applications

This section shows how to use the INFOConnect functions to accomplish the basic procedures or tasks that all INFOConnect applications must follow.

Initializing INFOConnect Connectivity Services

All INFOConnect events, functions and types are defined in Xvt.h which was modified by IcXvtMod.exe during IDK installation. To access the INFOConnect definitions, you must define the tag ICXVT before including Xvt.h. If your application also includes Windows.h, you must also define the tag ICXVTWIN before including Xvt.h.

Call ic_init_ics to initialize INFOConnect Connectivity Services. A good place to do this is during E_CREATE processing in the event handler routine. IC_VERSION_... and IC_REVISION_... provide version control over the INFOConnect API. Refer to page 4-32, Running with old versions of INFOConnect, for more information on version control.

Sample code

```
#define ICXVT
#include <xvt.h>

IC_RESULT icerror;

long event_handler(WINDOW win, EVENT *ep)
{

  switch (ep->type) {

  /*****************************************************/
  /* Standard XVT events are shown first. */
  /*****************************************************/

  case E_CREATE:

    /* Initialize INFOConnect Interfaces */
    icerror = ic_init_ics(IC_VERSION_3_0, IC_REVISION_3_0);
    if (icerror != IC_OK) {
          ic_default_error_proc(win, NULL_IC_SESSHND,
                    NULL, icerror);
      xvt_terminate();
    }
    break;

  case ...

}
```

Opening a Session and Allocating Buffers

Before you can send data with INFOConnect, you must open an INFOConnect session. The steps to open a session and allocate datacomm buffers follow:

Initialize INFOConnect

Initialize INFOConnect Connectivity Services using ic_init_ics if you haven't done so already.

Call ic_open_session

Call ic_open_session to request a path to be opened. You can supply a path or let INFOConnect prompt the user for a path. If INFOConnect returns a non-severe error, then a pending session handle has been created. INFOConnect will generate an E_IC_SESSION_EST event later when the session is actually established.

Do not use the pending session handle before the E_IC_SESSION_EST event occurs. No E_IC_SESSION_EST event is generated if INFOConnect returns a severe error on the ic_open_session call.

All INFOConnect events for this session will be delivered to the event handler associated with the window specified on the first parameter of the ic_open_session function call.

Handle the E_IC_SESSION_EST event and allocate buffers

Add a case statement for E_IC_SESSION_EST to the appropriate event handler with code to process the session establishment event and allocate buffers for transmitting and receiving data. When determining what buffer size to use, call ic_get_session_info to find the maximum buffer size the underlying communications software can support. Since connections may support very large buffers, you may want to put a ceiling on the buffer size as shown in the code fragment. Function ic_buf_alloc must be used to allocate the buffers to satisfy underlying platform requirements or shared global handles. Check for NULL_IC_BUFHND after the allocation requests to see that the buffers were properly allocated.

Define a BOOLEAN indicating 'session establishment'

It is a good idea to create and set a global BOOLEAN variable in your application to indicate the current state of your session. This is primarily needed during the time between the ic_open_session call and before the E_IC_SESSION_EST event is returned to your application. The sample code uses a variable named *bSessionEst* and makes no INFOConnect requests using the pending session handle until *bSessionEst* is TRUE.

Call ic_close_session after errors

If the E_IC_SESSION_EST event contains a severe error you must call ic_close_session to release the pending session handle.

Sample Code

```
#define MAXBUFSIZE 4096
#define HSESSION (ep->v.ic.session)
#define ICRESULT (ep->v.ic.v.result)

/* Global variables used with INFOConnect */
IC_SINFO sinfo;
IC_RESULT icerror;
struct aSession {
  IC_SESSHND hSession;
  IC_BUFHND hXmtBuf;
  IC_BUFHND hRcvBuf;
  BOOLEAN bSessionEst;
  unsigned uBufsize;
}s;

...

long event_handler(WINDOW win, EVENT *ep)
{
  switch (ep->type) {
  case E_CREATE:
    if (!do_ic_init(win))
      xvt_terminate();
    break;
```

```
   case E_IC_SESSION_EST:
     if (ICRESULT != IC_OK) {
       ic_default_error_proc(win, HSESSION, ep->type, ICRESULT);
       ic_close_session(HSESSION);
     }
     else { /* session establishment was successful */
       ic_get_session_info(HSESSION, &sinfo);

       /* Allocate INFOConnect buffers. */
       s.uBufsize = min((unsigned)sinfo.max_size, MAXBUFSIZE);
       s.hXmtBuf = ic_buf_alloc((long)s.uBufsize);
       s.hRcvBuf = ic_buf_alloc((long)s.uBufsize);
       assert(s.hXmtBuf != NULL_IC_BUFHND);
       assert(s.hRcvBuf != NULL_IC_BUFHND);
       s.bSessionEst = TRUE;
     }
     break;

   case E_IC_ERROR:
     ic_default_error_proc(win, HSESSION, ep->type, ICRESULT);
     break;
   }
}

BOOLEAN do_ic_init(WINDOW win)
{
   s.hSession = NULL_IC_SESSHND;
   s.hXmtBuf = NULL_IC_BUFHND;
   s.hRcvBuf = NULL_IC_BUFHND;
   s.bSessionEst = FALSE;
   s.uBufsize = 0;
   icerror = ic_open_session(win, NULL, &s.hSession);
   if (icerror != IC_OK) {
     ic_default_error_proc(win, NULL_IC_SESSHND,
               (unsigned)NULL, icerror);
     return(FALSE);
   }
   return(TRUE);
}
```

Transmitting a Buffer

The ic_xmt function attempts to transmit a buffer of data. This function is asynchronous in nature; it returns immediately to the application before the transmit request is completed. A non-severe error is returned indicating the transmit request has been initiated. When the transmit request finishes, one of two events is passed to your event_handler routine: E_IC_XMT_DONE or E_IC_XMT_ERROR. The basic steps to follow are:

Don't transmit prematurely

Don't transmit over a session before the E_IC_SESSION_EST event is returned for that session. Don't transmit while a previous transmit request is still pending for that session. The sample code below uses two variables, *bSessionEst* and *nXmtTries*, to manage these conditions.

Allocate and lock a transmit buffer

Allocate a transmit buffer using ic_buf_alloc if you haven't already done so. Lock the transmit buffer with ic_buf_lock.

Fill the buffer and unlock it

Copy your data to the transmit buffer. If the data can contain null characters, you may find the XVT function gmemcpy useful, otherwise, you can use gstrcpy. Remember that gstrcpy will stop as soon as a null character is found.

Unlock the buffer with ic_buf_unlock.

Call ic_xmt

Pass the buffer to INFOConnect with ic_xmt.

You should declare a global variable to indicate that a transmit request is outstanding. Check this variable to avoid transmitting a second buffer before a previous transmit request has finished.

One approach is to simply use a global BOOLEAN to indicate outstanding transmit requests. Don't transmit unless the BOOLEAN is clear, then set the BOOLEAN after calling ic_xmt successfully. Clear the BOOLEAN when an E_IC_XMT_DONE or E_IC_XMT_ERROR event is received.

The sample program, IcXvtApp, uses a different approach. A counter named nXmtTries indicates when a transmit request is pending and will also be useful later to manage transmit errors and retries.

Handle the E_IC_XMT_DONE and E_IC_XMT_ERROR events

Add case statements for E_IC_XMT_DONE and E_IC_XMT_ERROR to your event_handler function. Set your global variable to indicate that transmit requests are now allowed.

The E_IC_XMT_DONE event contains the buffer handle and buffer length of the transmitted data.

An E_IC_XMT_ERROR event contains an IC_RESULT with the reason for the transmit failure. Don't ignore these events. You may want to display the error or retry the transmit request. See page 4 - 24 for more on handling data communications errors.

Sample code

```
STR_FAR buf;
int nXmtTries=0;

void do_transmit(char localbuf[])
{
 if (nXmtTries>0 || !bSessionEst)
         note("Not ready to transmit");
 else {
         buf=ic_buf_lock(hXmtBuf)
         gmemcpy(buf,localbuf,(long)bufsize);
         ic_buf_unlock(hXmtBuf);
         icerror=ic_xmt(hSession,hXmtBuf,bufsize);
         if (icerror==IC_OK)
                 nXmtTries=1;
         else
                 ic_default_error_proc(win,hSession,NULL,icerror);
 }
}

long event_handler(WINDOW win,EVENT *ep)
{
 switch (ep->type) {
 case E_IC_XMT_DONE:
         nXmtTries=0;
         ...
         break;

 case E_IC_XMT_ERROR:
         ...see discussion on handling datacomm errors
         break;
 }
}
```

Receiving a Buffer

The ic_rcv function requests a buffer of data. This function is asynchronous in nature; it returns immediately to the application before the receive request is completed. A non-severe error is returned indicating the receive request was initiated. When the receive request finishes, one of two events is passed to your event_handler routine: E_IC_RCV_DONE or E_IC_RCV_ERROR. The basic steps to follow are:

Don't issue a receive request prematurely

Don't use a session before the E_IC_SESSION_EST event is returned for that session. Don't make a receive request while a previous receive request is still pending for that session. The sample code below uses two variables, *bSessionEst* and *nRcvTries*, to manage these conditions.

Allocate a receive buffer

Allocate a receive buffer using ic_buf_alloc if you haven't already done so.

Call ic_rcv

Pass the buffer to INFOConnect with ic_rcv.

You should declare a global variable to indicate that a receive request is outstanding. Check this variable to avoid making a second receive request with the previous one still outstanding.

One approach is to simply use a global BOOLEAN to indicate outstanding receive requests. Don't issue a receive request unless the BOOLEAN is clear, then set the BOOLEAN after calling ic_rcv successfully. Clear the BOOLEAN when an E_IC_RCV_DONE or E_IC_RCV_ERROR event is received.

The sample program, IcXvtApp, uses a different approach. A counter named nRcvTries indicates when a receive request is pending and will also be useful later to manage receive errors and retries.

Handle the E_IC_RCV_DONE and E_IC_RCV_ERROR events

Add case statements for E_IC_RCV_DONE and E_IC_RCV_ERROR to your event_handler routine. Set your global variable to indicate new receive requests are now allowed.

The E_IC_RCV_DONE event contains the buffer handle and buffer length of the received data.

An E_IC_RCV_ERROR events contains an IC_RESULT with the reason for the receive failure. Don't ignore these events. You may want to display the error or retry the receive request. See page 4 - 24 for more on handling data communications errors.

Sample code

```
STR_FAR buf;
int nRcvTries = 0;

void do_receive()
{
 if (nRcvTries>0 || !bSessionEst)
        note("Receive outstanding or session not ready");
 else {
        icerror = ic_rcv(hSession, hRcvBuf, bufsize);
        if (icerror == IC_OK);
                nRcvTries = 1;
        else
                ic_default_error_proc(win,
                        hSession,
                        NULL,
                        icerror);
 }
}

long event_handler(WINDOW win, EVENT *ep)
{
 switch (ep->type) {
 case E_IC_RCV_DONE:
        nRcvTries = 0;
        ...
        break;
 case E_IC_RCV_ERROR:
        ...see discussion on handling datacomm errors
        break;
 }
}
```

Using Datacomm Buffers

This section covers some guidelines that will help you better manage your application's datacomm buffers. The ic_rcv and ic_xmt functions are asynchronous in nature; they return immediately before the datacomm request is actually completed. This means your application must be careful about accessing the datacomm buffers that were passed to these functions. The following list will help you better manage the data communications of your application.

Allocate datacomm buffers using INFOConnect routines

Use ic_buf_alloc to allocate buffers that will be passed to INFOConnect. This ensures that the buffers have the proper system attributes. Datacomm buffers must be global and shareable across applications.

INFOConnect datacomm is asynchronous

Calls to ic_rcv and ic_xmt return to your application immediately before the request is actually completed. Later, an event will be sent to your application when the request is completed. Until this event is returned, your request is referred to as pending.

Don't use pending buffers

Do not access a buffer that is associated with a pending request. It is a good idea to define and set a variable for each buffer that tracks when the buffer is associated with a pending request.

Don't issue a receive or transmit request until the E_IC_SESSION_EST event has been returned.

Don't issue a receive request while a pending receive request exists for the same session. The state of the first receive request is undefined. Normally, the first request is canceled without returning an E_IC_RCV_DONE or E_IC_RCV_ERROR event, but your application cannot assume that all external interfaces will behave similarly for this situation. The same holds true for premature transmit requests.

Use one receive buffer and one transmit buffer

You can have as many datacomm buffers as you like, but a session can only have one active transmit buffer and one active receive buffer at a time. Therefore, it is recommended to allocate one receive buffer and one transmit buffer.

You can use one buffer for both transmitting and receiving, but for maximum interoperability use separate buffers. Your application can be more responsive using separate buffers since it can keep a receive request pending while waiting for a pending transmit request to complete.

Cancel pending requests with ic_lcl

Pending requests can be canceled by using ic_lcl. You must wait until an E_IC_LCL_RESULT event is returned to your application on behalf of the canceled messages before you can safely access any datacomm buffers.

Don't ignore errors

Don't ignore error events (E_IC_XMT_ERROR and E_IC_RCV_ERROR). You may want to retry the request at least some number of times. See page 4 - 24 for more on handling data communications errors.

Basic Error Handling

Nearly every INFOConnect function and event returns a 'long' (type IC_RESULT) indicating the success of the function or event. Any value other than IC_OK indicates an error. Your application can handle errors in several ways:

- Pass errors back to INFOConnect for handling
- Display and handle errors yourself
- Test for standard errors and resume without any user intervention
- Handling E_IC_ERROR events

Here are the four categories of errors and recommended actions for each:

IC_ERROR_INFO	Log this error if the application has a log file. Don't bother displaying a message to the user. The requested function was completed.
IC_ERROR_WARNING	The requested function was completed, but something unusual or noteworthy happened. The application can choose to log or display this error. The default error procedure will display these errors.
IC_ERROR_SEVERE	The requested function did not complete successfully. The user should usually see this error.
IC_ERROR_TERMINATE	Display this error and close the session.

Passing errors back to INFOConnect for handling

If your application doesn't need to control the presentation style of the error message, the simplest way to handle errors is to pass them back to the INFOConnect default error procedure.

Note: *The INFOConnect default error procedure will automatically call ic_close_session on your behalf for errors of type IC_ERROR_TERMINATE. On return from the default procedure, it is good practice to test for IC_ERROR_TERMINATE type errors and, at a minimum, mark the session as closed. Otherwise, you might inadvertently make a transmit or receive request before the E_IC_SESSION_CLOSE event is sent to your application.*

Here is a code fragment that illustrates using the default error procedure after receiving an INFOConnect event and also after calling an INFOConnect function.

```
long event_handler(WINDOW win, EVENT *ep)
{
 switch (ep->type) {
 case E_IC_RCV_ERROR:
  handle_ic_error(win,
       ep->v.ic.session,
       ep->type,  /* event type valid */
       ep->v.ic.v.result);
  /* try receive again */
  icerror = ic_rcv(ep->v.ic.session,
        hRcvBuf,
        uBufsize);
  if (icerror != IC_OK)
   handle_ic_error(win,
        ep->v.ic.session,
        NULL,    /* no event type */
        icerror);
  break;
}

void handle_ic_error(WINDOW   win,
       IC_SESSHND hSession,
       unsigned  type,
       IC_RESULT icerror)
{
 ic_default_error_proc(win, hSession, type, icerror);
 if (IC_GET_RESULT_TYPE(icerror) >= IC_ERROR_TERMINATE)
  xvt_terminate();
}
```

The advantage of using function handle_ic_error instead of calling ic_default_error_proc directly is to leave your application the flexibility of adding code in handle_ic_error to format your own messages in the future.

Look at the two calls to handle_ic_error. Compare the 3rd parameter on each call. On the first call, the event type is known (E_IC_RCV_ERROR) and is passed to ic_default_error_proc. On the second call, NULL is passed because there is no relevant 'event type' after making an INFOConnect function call. Most of the time, you will be passing NULL for the 3rd parameter. For more examples on using the event-type parameter with ic_default_error_proc, see the sample programs on the IDK disk.

Displaying and handling errors yourself

To display error messages directly from your application, function ic_get_string is available to retrieve the text of the error. You must also satisfy the following requirements normally provided by ic_default_error_proc.

- Don't display informational errors. Informational errors are of type IC_ERROR_INFO. IC_GET_RESULT_TYPE is an INFOConnect macro used to extract the error type from an IC_RESULT.
- There are a few termination messages used by INFOConnect to close sessions without displaying anything to the user. You should suppress displaying the following:

  IC_ERROR_TERMINATE_NOMSG
  IC_ERROR_TERMINATE_CLEAR
  IC_ERROR_TERMINATE_EXIT
  IC_ERROR_TERMINATE_SHUTDOWN.
- Close the session for errors of type IC_ERROR_TERMINATE. IC_GET_RESULT_TYPE is an INFOConnect macro used to extract the error-type from an IC_RESULT.

```
void foobar()
{
 IC_RESULT icerror;

 icerror=ic_get_session_info(hSession,...)
 if (icerror != IC_OK)
         handle_ic_error(win, hSession, NULL, icerror);

}

void handle_ic_error(WINDOW   win,
          IC_SESSHND hSession,
          unsigned  uType,
          IC_RESULT  icerror)
{
  char msg[256];

  if (((IC_GET_RESULT_TYPE(icerror) & IC_ERROR_MASK) ==
                                        IC_ERROR_INFO) &&
    (icerror != IC_ERROR_TERMINATE_NOMSG) &&
    (icerror != IC_ERROR_TERMINATE_CLEAR) &&
    (icerror != IC_ERROR_TERMINATE_EXIT)  &&
    (icerror != IC_ERROR_TERMINATE_SHUTDOWN))
  {
   if (ic_get_string(hSession,
           icerror,
           (LPSTR)msg,
           sizeof(msg)) == IC_OK) {

     note(win,msg);
   }
  }
  if ((IC_GET_RESULT_TYPE(icerror) & IC_ERROR_MASK) ==
     IC_ERROR_TERMINATE ) {
   ic_close_session(hSession);
  }
  return IC_OK;
}
```

Testing for standard errors

Standard INFOConnect errors are defined in IcError.h You may find some errors that your application can intercept and resolve without bothering the user. Appendix C of the *IDK Programming Reference Manual* also lists the standard errors.

Handling E_IC_ERROR events

Although they are infrequent, all applications must be prepared to handle E_IC_ERROR events. Assuming you have written a function like 'handle_ic_error' shown above, you can call it as follows:

```
long event_handler(WINDOW win, EVENT *ep)
{
 switch (ep->type) {
 case E_COMMAND:
   ...
   break;
 case ...

 case E_IC_ERROR:
   handle_ic_error(win, ep->v.ic.session, ep->type, ep->v.ic.v.result);
   break;
 }
}
```

Basic Status Handling

Status messages are used to communicate between your application and INFOConnect. Statuses can travel in one of two directions: from the application to INFOConnect or from INFOConnect to the application. Your application uses function ic_set_status to send statuses and watches for event E_IC_STATUS to receive statuses from INFOConnect.

Standard statuses are defined in the IcStatus.h header file. Here is a sampling of statuses:

```
/* Line state values */
#define IC_LINESTATE_XMT ...
#define IC_LINESTATE_RCV ...
#define IC_LINESTATE_LCL ...

/* Connection state */
#define IC_CONNECT_EOF ...
#define IC_CONNECT_CLOSE ...
#define IC_CONNECT_OPEN ...
#define IC_CONNECT_NOACTIVITY ...
#define IC_CONNECT_ACTIVITY ...

/* Reactivate session when sinfo.focus_notify == TRUE */
#define IC_REACTIVATE_ON ...
#define IC_REACTIVATE_OFF ...

/* Requests to applications */
#define IC_CONTROL_ACTIVATE ...
#define IC_CONTROL_RCVREADY ...
#define IC_CONTROL_RCVAVAIL ...
```

Does my application need to do something for every possible status?

No. Most statuses can be ignored by your application. Here are the primary statuses your application should support:

- IC_CONTROL_ACTIVATE
- IC_REACTIVATE_ON and IC_REACTIVATE_OFF
- IC_CONTROL_RCVAVAIL
- IC_STATUS_TRANS

Handling IC_CONTROL_ACTIVATE statuses

If INFOConnect wants your application to become the active window, it sends an IC_CONTROL_ACTIVATE status. The following code fragment shows how to support this status:

```
long event_handler(WINDOW win, EVENT *ep)
{
  switch (ep->type) {
  case E_COMMAND:
    ...
    break;
  case E_IC_STATUS:
    if (ICRESULT == IC_CONTROL_ACTIVATE)
      /* Support the 'GoTo' button from INFOConnect */
      set_front_window(win);
    break;
  case ...
    ...
    break;
  }
}
```

Sending IC_REACTIVATE statuses

Sometimes the underlying INFOConnect libraries are required to be notified when your application-window gains and loses focus. This is an attribute of the session and is referred to as focus-notification. Your application must do two things related to focus-notification.

- Call ic_get_session_info to determine if your session requires focus-notification
- Recognize E_FOCUS events and send IC_REACTIVATE statuses

```
BOOLEAN bSessionEst;
BOOLEAN bFocusNotify;

long event_handler(WINDOW win, EVENT *ep)
{
 IC_SINFO sinfo;

 switch (ep->type) {

 case E_IC_SESSION_EST:
  ...
  bSessionEst = TRUE;
  icerror = ic_get_session_info(hSession, &sinfo);
  if (icerror != IC_OK)
   ic_default_error_proc(win, hSession,
          NULL, icerror);
  bFocusNotify = sinfo.focus_notify;
  ...
  break;

 case E_FOCUS:
  if ((bFocusNotify) && (bSessionEst)) {
   if (ep->v.active)
    icerror=ic_set_status(hSession,IC_REACTIVATE_ON);
   else
    icerror=ic_set_status(hSession,IC_REACTIVATE_OFF);
   if (icerror != IC_OK)
    ic_default_error_proc(win, hSession,
          NULL, icerror);
  }
  break;
```

Handling IC_CONTROL_RCVAVAIL statuses

If your application normally stays in receive mode, you probably don't need to watch for IC_CONTROL_RCVAVAIL statuses, but applications that intentionally stay out of receive mode for long periods of time should recognize this status. When an application is not in receive mode and a message becomes available, the underlying INFOConnect libraries can send an IC_CONTROL_RCVAVAIL status to request your application to go into receive mode. Ignoring IC_CONTROL_RCVAVAIL statuses may result in your application appearing sluggish and unresponsive.

Sending IC_STATUS_TRANS statuses

In order for INFOConnect to keep an accurate count of transactions, your application needs to notify INFOConnect of the beginning and the end of the transactions. Use IC_TRANSACTION_ON and IC_TRANSACTION_OFF to indicate whether or not the transactions will be flanked by IC_TRANSACTION_BEGIN and IC_TRANSACTION_END status messages.

Closing a Session

Call ic_close_session to end an INFOConnect session. This function returns immediately. Event E_IC_SESSION_CLOSE is passed to the event_handler when the session is actually closed. The basic steps to follow are:

Clear the BOOLEAN indicating 'session establishment'

The sample code uses a variable named *bSessionEst* for this purpose.

Call ic_close_session and deallocate buffers

Use INFOConnect memory management routines to free datacomm buffers. Notice that datacomm buffers can be freed immediately after returning from IcCloseSession rather than waiting for the E_IC_SESSION_CLOSE event. The Manager cancels any outstanding transmit or receive requests during ic_close_session to ensure that the datacomm buffers are idle. It is a good idea to set the buffer handle variables to NULL_IC_BUFHND after releasing the buffers.

Handle the E_IC_SESSION_CLOSE event

This is the best place to clear the session handle with NULL_IC_SESSHND rather than immediately after the ic_close_session function call.

Sample code

```
void do_close()
{
 bSessionEst=FALSE;
 if(hSession!=NULL_IC_SESSHND){
  ic_close_session(hSession);
 }
 if(hXmtBuf!=NULL_IC_BUFHND){
  ic_buf_free(hXmtBuf);
  hXmtBuf=NULL_IC_BUFHND;
 }
 if(hRcvBuf!=NULL_IC_BUFHND){
  ic_buf_free(hRcvBuf);
  hRcvBuf=NULL_IC_BUFHND;
 }
}

long event_handler(WINDOW win, EVENT *ep)
{
 switch (ep->type){
        case E_IC_SESSION_CLOSE:
                hSession=NULL_IC_SESSHND;
                break;
 }
}
```

Terminating your Application

The sequence of events during application termination can vary depending on who initiated the termination: the user, INFOConnect, the system or your application itself. For each scenario, the first event passed to your application is different. See the discussion after the code fragment for specific details about each scenario.

Close any open INFOConnect sessions and free any buffers before terminating your application. The general recommended strategy is to do all this during E_DESTROY processing. Notice the precautions taken in the code fragment below such as setting released resources to NULL.

The following scheme handles termination cleanly no matter how it is initiated.

```
long event_handler(WINDOW win, EVENT *ep)
{
 switch (ep->type) {
 case E_COMMAND:
        switch (ep->v.cmd.tag) {
        case ...
        case M_FILE_QUIT:
                xvt_terminate();
                break;
        }
        break;

 case E_CLOSE:
        xvt_terminate();

 case E_QUIT:
        if (ep->v.query) quit_OK();
        else    xvt_terminate();
        break;

 case E_DESTROY:
        do_ic_cleanup();
        break;

 case E_IC_SESSION_CLOSE:
        /* See discussion below */
        hSession = NULL_IC_SESSHND;
        xvt_terminate();
        break;

 case E_IC_STATUS:
        if (ICRESULT == IC_COMMMGR_QUERYEXIT)
                ic_exit_ok(TRUE);
        if (ICRESULT == IC_COMMMGR_CANCELEXIT)
                ; /* Some other app responded QUERYEXIT/FALSE.
                  INFOConnect won't exit after all.    */
        if (ICRESULT == IC_COMMMGR_EXIT)
                ; /* INFOConnect is really going away.
                  All apps responded QUERYEXIT/TRUE.       */
        break;
 }
}
```

```
void do_ic_cleanup()
{
 bSessionEst=FALSE;
 if(hSession!=NULL_IC_SESSHND){
  ic_close_session(hSession);
 }
 if(hXmtBuf!=NULL_IC_BUFHND){
  ic_buf_free(hXmtBuf);
  hXmtBuf=NULL_IC_BUFHND;
 }
 if(hRcvBuf!=NULL_IC_BUFHND){
  ic_buf_free(hRcvBuf);
  hRcvBuf=NULL_IC_BUFHND;
 }
}
```

User-initiated termination of your application - M_FILE_QUIT

Most applications have a File-menu/Exit-option that allows the user to terminate the application by choosing the Exit option. This results in an E_COMMAND event with a M_FILE_QUIT command tag value. Your application needs to be prepared to close INFOConnect sessions and free any associated buffers. The code fragment shown above accomplishes this by calling xvt_terminate.

INFOConnect-initiated termination - E_IC_SESSION_CLOSE

If the user clears the session associated with your application from the INFOConnect Manager window, INFOConnect sends your application an E_IC_SESSION_CLOSE event. To abort your application at this point, the recommended procedure is to call xvt_terminate from E_IC_SESSION_CLOSE processing. Do not call xvt_terminate from E_IC_SESSION_CLOSE processing if you want to keep your application active in this situation.

User-initiated termination of the INFOConnect Manager

If the user shuts down INFOConnect itself from the INFOConnect Manager window (ALT-F4 keystroke), a series of status messages are sent to all active INFOConnect applications. First your application receives IC_COMMMGR_QUERYEXIT. The application must call ic_exit_ok with either TRUE if it is OK to terminate or FALSE if it is not OK to terminate. If it is OK, then INFOConnect will continue querying the other INFOConnect applications.

If an application refuses to shut down, all applications that had agreed to the shutdown are sent IC_COMMMGR_CANCELEXIT and continue with normal execution. If all applications agree to the shutdown, the Manager then sends IC_COMMMGR_EXIT to confirm that all applications have agreed to close. As a final notice, the application will receive a terminate severity error, IC_ERROR_TERMINATE_EXIT. The application can temporarily delay closing the session if necessary.

System-initiated termination - E_QUIT and E_CLOSE

Termination invoked by the system generates E_CLOSE and E_QUIT events.

Application-initiated termination

Your application can force termination itself by calling xvt_terminate. To do this you should follow the suggestion of releasing all INFOConnect resources during E_DESTROY processing.

Advanced Procedures for XVT Applications

Canceling Pending Requests

Many applications typically stay in receive mode so they're ready to respond to any messages from the partner activity. Occasionally, it may be necessary to cancel this outstanding receive request.

The ic_lcl function selectively cancels pending actions. A parameter indicates what is to be canceled: the pending transmit request, pending receive request, or both. Your application should wait for an E_IC_LCL_RESULT event before accessing the datacomm buffers associated with the canceled actions. By waiting for this event, you are sure to process any E_IC_RCV_DONE or E_IC_XMT_DONE events that were already in your event queue before the ic_lcl was done.

```
/* The following definitions are in ICDEF.H        */
/* You do NOT need to define these in your application */

#define IC_LCL_RCV  1
#define IC_LCL_XMT  2
#define IC_LCL_RCVXMT (IC_LCL_RCV | IC_LCL_XMT)
#define IC_LCL_CLOSESESSION 4
```

Sample code

The following code cancels any pending receive or transmit request.

```
icerror = ic_lcl(hSession, IC_LCL_RCVXMT);
if (icerror != IC_OK) {
  ic_default_error_proc(win, hSession, NULL, icerror);
```

Handling Data Communications Errors

Data communications errors show up as E_IC_XMT_ERROR and E_IC_RCV_ERROR events in the event_hander routine. A good application will act on these events. Often, you may just want to retry the request for some number of times.

The following code fragment will retry datacomm errors five times before displaying an error to the user. Two variables, *nXmtTries* and *nRcvTries*, are the basis of this technique. They are incremented when a request results in an error. They are set to zero after successful completion (E_IC_XMT_DONE and E_IC_RCV_DONE) indicating there is no longer an outstanding request.

```
#define MAXRETRIES 5

/* global variables */
int nXmtTries = 0;
int nRcvTries = 0;

long event_handler(WINDOW win, EVENT *ep)
  switch (ep->type) {
  ...
  case E_IC_XMT_DONE:
    nXmtTries = 0;
    ...
    break;

  case E_IC_RCV_DONE:
    nRcvTries = 0;
    ...
    break;

  case E_IC_XMT_ERROR:
   if (++nXmtTries > MAXRETRIES) {
    ic_default_error_proc(win,
              ep->v.ic.session,
              ep->type,
              ep->v.ic.v.result);
    nXmtTries = 1;
   }
   /* try again */
   icerror = ic_xmt(ep->v.ic.session, hXmtBuf,
          gstrlen(sNoteBuf));
   if (icerror != IC_OK)
    nXmtTries = 0;
    ic_default_error_proc(win,
              ep->v.ic.session,
              NULL,
              icerror);
   break;
```

```
case E_IC_RCV_ERROR:
   if(++nRcvTries>MAXRETRIES){
    ic_default_error_proc(win,
            ep->v.ic.session,
            ep->type,
            ep->v.ic.v.result);
    nRcvTries=1;
   }
   /* try again */
   icerror=ic_rcv(ep->v.ic.session,hRcvBuf,uBufsize);
   if(icerror!=IC_OK)
    nRcvTries=0;
    ic_default_error_proc(win,
            ep->v.ic.session,
            NULL,
            icerror);
   break;

  ...
  }
```

Advanced Status and Error Handling

Before reading this discussion, you should be familiar with the definition of *context* given in Section 2 , "An Introduction to INFOConnect Connectivity Services."

What is typedef IC_RESULT?

INFOConnect statuses and errors are defined with type IC_RESULT. An IC_RESULT type is a 'long' made up of three parts: a context, a type and a value. The formal names for these three are: IC_RESULT_CONTEXT, IC_RESULT_TYPE and IC_RESULT_VALUE.

The following macros are available for building and tearing down IC_RESULT types.

IC_MAKE_RESULT	Builds IC_RESULT from its 3 parts
IC_GET_RESULT_CONTEXT	Extracts 'context' from IC_RESULT
IC_GET_RESULT_TYPE	Extracts 'type' from IC_RESULT
IC_GET_RESULT_VALUE	Extracts 'value' from IC_RESULT

How do I test for library-specific statuses and errors?

If necessary, you can determine where a status or error is defined by extracting the context from the IC_RESULT. This might be useful when you are formatting your own error messages and want to include the name of the library that defines the error.

Note: *The library that defines a message is not necessarily the only library that uses or generates that message, although that is generally the case. Standard INFOConnect statuses and errors have a context of IC_RESULT_CONTEXT_STD and are defined in IcError.h and IcStatus.h.*

Each INFOConnect library or accessory that defines unique statuses and errors must provide a header file defining the types and values for its statuses and errors. The naming convention for the header file is to use the .HIC suffix for the header file. The .HIC file also contains configuration information For example, TTY-specific definitions from IcTTY.dll are defined in IcTTY.hic.

What is the scope or visibility of statuses and errors?

Generally, statuses and errors are only 'seen' by the components in the current path: the application, the service libraries, and the external interface library. Status and error events are not sent across the connection by INFOConnect except for accessories executing locally (invoked via ic_open_accessory). One of the sample programs, CoupleS, is a service library that extends the scope of statuses and errors by encoding them and sending them across the connection.

What are these status messages telling me?

Perhaps you're wondering what the status messages listed in Appendix B of the *IDK Programming Reference Manual* really mean! The following paragraphs give a brief description of several of the status messages.

IC_CONTROL_RCVAVAIL status

The IC_CONTROL_RCVAVAIL status is sent to the application when a message is available but the application isn't in receive mode and ready to get it. Although a terminal emulator generally returns to receive mode as quickly as possible after each host message, there might be times when an IC_CONTROL_RCVAVAIL status is sent to the terminal because another message is available, but the terminal is still in a local state.

IC_CONNECT_... statuses

These statuses originate with the external interface. They can be used by the application to show the user some kind of status about the datacomm connection.

The IC_CONNECT_ACTIVITY and IC_CONNECT_NOACTIVITY pair of statuses indicate the presence of line activity. The exact meaning of a *line* depends on the specific external interface, but generally a line carries much more than the traffic for just one session. Therefore, IC_CONNECT_ACTIVITY means that something is happening on the line, but not necessarily for your session. IC_CONNECT_NOACTIVITY indicates the absence of any line activity for some reasonable period of time (for example,. 10 seconds). External interfaces must choose an appropriate time period so that the application is not flooded with nuisance statuses.

The IC_CONNECT_OPEN and IC_CONNECT_CLOSE pair of statuses indicate activity for a specific INFOConnect session, namely, the current one. Once again, the exact meaning varies with the external interface.

IC_CONTROL_ACTIVATE status

Suppose a session with an INFOConnect terminal is minimized and focus is switched to the main INFOConnect window. The INFOConnect window shows the active sessions. By selecting the Goto button while the session for the INFOConnect terminal is selected, INFOConnect sends an IC_CONTROL_ACTIVATE status to the application requesting it to grab the input focus.

IC_LINESTATE_... statuses

Three line state statuses, IC_LINESTATE_XMT, IC_LINESTATE_RCV, and IC_LINESTATE_LCL, are intended to help terminal emulators tell the user about the current state of the line. Depending on the underlying libraries, calls to ic_rcv and ic_xmt are sometimes followed by IC_LINESTATE_RCV and IC_LINESTATE_XMT statuses. Also, E_IC_RCVDONE events are sometimes followed by IC_LINESTATE_LCL statuses.

Using IC_STATUS_BUFFER extended status

When an application needs to exchange more information with an ICS library than IC_RESULT_VALUE can store, it can send a buffer of information with the IC_STATUS_BUFFER extended status. To accomplish this, a HIC_STATUSBUF buffer handle is assigned to the IC_RESULT_VALUE member of the IC_RESULT structure.

Extended statuses can be exchanged in two ways: synchronously and asynchronously. Refer to Section 7, "Writing INFOConnect Libraries for Windows 3.x" for detailed steps on exchanging extended statuses.

Using Event Hooks with XVT 3.0

INFOConnect uses an XVT event hook routine to pass INFOConnect events to your window event handler. You are still free to use an event hook routine from your application, but you must be careful to pass control to any hook routine already registered with XVT. In other words, all event hook routines must cooperate and pass control to the next routine in the chain.

For example, here is the code in INFOConnect that manages the XVT event hook interface. The hook routine itself, ic_event_hook, is registered with XVT during ic_init_ics processing. Ic_init_ics also saves the address of any preexisting hook routine. Later, when ic_event_hook is actually called, it does its own processing and then passes control to the next hook routine in the chain.

```
typedef BOOLEAN (FAR *EVENTHOOK)(HWND hwnd, \
                unsigned message, \
                unsigned wParam, \
                long lParam, \
                long FAR *rp);
EVENTHOOK old_event_hook;

IC_RESULT ic_init_ics(int version, int revision)
{
  ...
  old_event_hook = (EVENTHOOK)get_value(NULL_WIN,
                  ATTR_EVENT_HOOK);
  set_value(NULL_WIN, ATTR_EVENT_HOOK, (long)ic_event_hook);
  return IC_OK;
}

BOOLEAN FAR ic_event_hook(HWND hwnd,
             unsigned message,
             unsigned wParam,
             long lParam,
             long FAR* rp)
{
  ...
  if ((EVENTHOOK)0L == old_event_hook)
    return TRUE;
  else
    return ((*old_event_hook)(hwnd, message, wParam, lParam, rp));
}
```

Using Keyboard and Event Hooks with XVT 2.0

Note: *The following discussion only applies to developers using XVT 2.x. Event and keyboard hook processing changed between XVT 2.0 and 3.0.*

XVT provides a way for applications to alter the standard keyboard and event behavior. Module whook.c is provided by XVT 2.0 and contains sample code for function key_hook and event_hook. INFOConnect applications still have this flexibility although the mechanics have been slightly altered.

event_hook

If your application requires event_hook functionality, you must follow the standard instructions given in whook.c, however, you must name your function ICevent_hook instead of event_hook because INFOConnect itself uses event_hook. Module icxehook.c is provided as a starting point for ICevent_hook. The Windows version is in \idk\win\icxehook.c.

```
BOOLEAN ICevent_hook(int rid, BOOLEAN modal, MSG far *mp) {
 /* Called for every event from windows */
  return TRUE;
 }
```

key_hook

If your application requires key_hook functionality, you should follow the standard instructions given in whook.c. The name of your function is key_hook as documented in whook.c. INFOConnect does not provide a template for key_hook; use the code in the XVT whook.c module.

Windows 3.x Linking considerations for the hook routines

Icxevent.obj is provided by INFOConnect and must always be included in the link regardless of whether or not you are providing your own ICevent_hook. Your own ICevent_hook module, icxehook.obj, is then added to the linkstream for your application.

On the other hand, icxkey.ob**j** is only a default key_hook routine provided by INFOConnect and is to be dropped from the link if you are providing your own key_hook routine.

Encoding and Decoding

Some transports do not guarantee that all binary data streams can be safely sent across the connection (that is, SINFO.transparent=FALSE). Binary data must be encoded by the sender and decoded by the receiver. This requires close coordination between the workstation and host components; INFOConnect does not currently provide encoding and decoding services. A service library is a good place to implement the workstation side of this functionality.

Data Compression and Error Detection

INFOConnect does not currently provide routines for data compression and error detection. Error detection is generally the responsibility of the communications layers beneath the INFOConnect architecture. Data compression, on the other hand, is an ideal application for an INFOConnect service library.

Running with Old Versions of INFOConnect

The basic philosophy of INFOConnect version control is "Old applications must still run with new versions of the Manager, but new applications are not required to run with old Managers." In fact, the Manager normally refuses to run applications built with a version of the IDK that is newer than the Manager itself. The assumption is that new applications might make new API calls that an old Manager doesn't know about.

If you are upgrading an existing application to a new version of the IDK, and can function properly without making new INFOConnect API calls, this section describes how to build your application with the latest IDK and initialize with different versions of the Manager at run time. Your application must remember which version of the Manager it is executing and only make appropriate API calls known by that version of the Manager.

The version of an application is normally "marked" on the initial call to ic_init_ics:

```
icerror=ic_init_ics(IC_VERSION_3_0,IC_REVISION_3_0);
if(icerror!=IC_OK)
  //error processing. INFOConnect services are not available.
```

IC_VERSION_3_0 and IC_REVISION_3_0 are defined in IcDef.h. IC_VERSION_... is added at major release levels (Release 1.0, 2.0, 3.0). IC_REVISION_... has a finer granularity -- a new one is added each time the INFOConnect APIs are extended or changed.

If you compile your XVT application using the 2.0 values for version and revision (IC_VERSION_2_0 and IC_REVISION_2_0) and attempt to run with INFOConnect 1.0, an error IC_ERROR_NEWVERSION is returned. However, your XVT application can successfully run with INFOConnect 2.0 or 3.0.

The following code sample is suggested for initializing with the Manager:

```
int icversion; /* global variable */
IC_RESULT icerror;

if ((icerror=IcInitIcs(IC_VERSION_CHECK,IC_REVISION_1_0))==IC_OK)
  icversion=IC_VERSION_1_0;
else if ((icerror=IcInitIcs(IC_VERSION_3_0,IC_REVISION_3_0))==IC_OK)
  icversion=IC_VERSION_3_0;
else if ((icerror=IcInitIcs(IC_VERSION_2_0,IC_REVISION_2_0))==IC_OK)
  icversion=IC_VERSION_2_0;
else {
  IcDefaultErrorProc(hWnd,NULL,NULL,icerror);
  /* INFOConnect services are unavailable!! */
}
```

At the end of this code fragment, *icversion* is set to the level of the Manager with which you are running.

CAUTION

Be careful to check icversion before making any API calls undefined in older versions of INFOConnect. Otherwise, you will encounter unexpected behavior.

The recommended way to isolate new API calls in your application follows:

```
if ((icversion >= IC_VERSION_3_0) &&
{
  /* call some new 3.0 INFOConnect API */
}
else {
  /* alternative action using pre 3.0 API calls */
}
```

Procedures for INFOConnect Accessories

INFOConnect applications that can be invoked and controlled by other INFOConnect applications are called accessories. INFOConnect accessories are written so that they can be used to build more sophisticated INFOConnect accessories. You communicate with an accessory through an INFOConnect session. It is easy and very useful to extend your INFOConnect application to become an accessory.

Calling INFOConnect Accessories

Two functions are available to invoke accessories: ic_open_accessory and ic_run_accessory. ic_open_accessory is the more useful of the two; a connection is established between your application and the accessory using the Local external interface.

Sample code

```
#define HSESSION (ep->v.ic.session)
IC_RESULT icerror;
IC_SINFO sinfo;
IC_SESSHND hMyHostConnection, hMyAnsiSession;

/* Retrieve characteristics of the host connection */
icerror = ic_get_session_info(hMyHostConnection, &sinfo);
if (icerror != IC_OK) {
  handle_ic_error(win, HSESSION, NULL, icerror);
  terminate();
}

/* Open an accessory named ANSI using pathname MYPATH */
icerror = ic_open_accessory(win, "ANSI", NULL,
        "MYPATH", &sinfo, &hMyAnsiSession);
if (icerror != IC_OK) {
  handle_ic_error(win, NULL_IC_SESSHND, NULL, icerror);
  terminate();
}
```

Notice the sinfo structure required as one of the parameters on ic_open_accessory. This code fragment assumes that a host connection has already been established. The session attributes of the host connection are retrieved and used to initialize the local connection to the ANSI accessory. For an example of using ic_open_accessory in complete context, see the sample program, IcOpenAc.

Making your Application an INFOConnect Accessory

Note: *See Section 6 of the* IDK Programming Reference Manual *for a complete list of requirements for accessories.*

All INFOConnect accessories are expected to do the following:

Parse the command line

All INFOConnect accessories should accept the -P and -L command line options for pathname and window location.

For a coding example of parsing the pathname, see function cmdline_get_path in the sample program IcXvtApp.

Display the INFOConnect session name

Where relevant, accessories should incorporate the INFOConnect session name in the window's title bar. This will help the user differentiate between multiple, active copies of an accessory.

The following code fragment was taken from the IcXvtApp sample program.

```
/* Display session name */

char sSessionName[IC_MAXSESSIONIDSIZE];

ic_get_session_id(HSESSION,
        sSessionName,
        sizeof(sSessionName));
set_doc_title(win, sSessionName);
```

Handle the IC_CONTROL_ACTIVATE status message

The code necessary to support this status was covered earlier under "Basic status handling." INFOConnect sends this status when the user presses the GoTo button on the INFOConnect window display of active sessions.

Registering and Deregistering as an accessory

Accessories that define statuses or errors must obtain a context by calling ic_register_accessory. Use the context string defined in your .HIC header file.

Provide a header file for accessory callers

Accessories must provide an .HIC header file that defines their identifying context string and any nonstandard, accessory-specific errors or statuses defined by the accessory. The compile-time name and value of the context string must be unique from all other INFOConnect accessories.

```
/****************************************/
/* SAMPLE.HIC                          */
/*                                     */
/****************************************/

#define SAMPLE_CONTEXTSTRING "SAMPLE"

/* status types - none defined */

/* error types - none defined */
```

Compiling

Windows Platform

Note: *Before compiling and linking any INFOConnect applications, review the System Verification Checklist in the Installation section.*

Memory Models

XVT applications can use either the Medium or Large memory models. The Windows SDK and XVT documentation cover the use of the various memory models. The INFOConnect architecture places no additional constraints on memory model usage.

Include files for XVT applications

All INFOConnect events, functions and types are defined in Xvt.h which was modified by IcXvtMod.exe during IDK installation. To access the INFOConnect definitions, you must define the tag ICXVT before including Xvt.h. If your application also includes Windows.h, you must also define the tag ICXVTWIN before including Xvt.h.

```
#define ICXVT /* Allow INFOConnect Interfaces */
#include <xvt.h>
```

C compiler options

Follow the recommendations in the XVT documentation. The sample programs provided with the IDK were built with the following options using the Microsoft C compiler.

```
cl  -c -AM -Gsw -W3 -Oils -Zp -FPc -DCCMSC -DOSDOS -DWSWIN
icxvtapp.c
```

Option	Description
-c	Compile only - don't link
-AM	Medium memory model
-Gsw	(s) remove stack probes (w) compile for Windows
-W3	Generate warnings
-Oils	(i) enable intrinsic functions (l) enable loop optimization (s) favor code size (t) favor execution time (d) optional flag for debugging ***Note:*** *Choosing between (s) and (t) can have a significant impact on your application. You may want to experiment before settling on one of them.* ***Note:*** *Using the (d) flag disables optimization and is recommended when using debuggers like CodeView. Some of the sample programs use the (d) flag in their make files. Don't forget to remove it when building the production version of your code.*
-Zp	(p) pack structures on 1 byte boundaries (i) optional flag for CodeView debugging
-FPc	Generate calls to the emulator floating-point library
-DCCMSC	Microsoft C compiler
-DOSDOS	MS-DOS operating system
-DWSWIN	Windows platform

Compiler Errors

Some common compilation errors and their solutions are listed in this section.

Out of near heap space

If your application includes both <xvt.h> and <windows.h> there may be too many names for the compiler to handle. The newer, DOS-extended compilers should not encounter this problem; upgrade if possible. Otherwise, see the comments in <windows.h> about removing unnecessary names from the compile. You will need to add code similar to the following:

```
#define ICXVT
#include <xvt.h>

#define NOGDICAPMASKS    - CC_*, LC_*, PC_*, CP_*, TC_*, RC_
#define NOWINSTYLES      - WS_*, CS_*, ES_*, LBS_*, SBS_*
#define NOSYSMETRICS     - SM_*
#define NOMENUS          - MF_*
#define NOICONS          - IDI_*
#define NOSYSCOMMANDS    - SC_*
#define NOMINMAX         - Macros min(a,b) and max(a,b)
#define NOSOUND          - Sound driver routines
#define NOWH             - SetWindowsHook and WH_*
#define NOWINOFFSETS     - GWL_*, GCL_*, associated routines
#define NOKANJI          - Kanji support stuff.
#define NOCOMM
#define NOPROFILER       - Profiler interface.
#define NOSCROLL         - SB_* and scrolling routines
#define NODEFERWINDOWPOS - DeferWindowPos routines

#include <windows.h>
```

Resource Files

XVT resource files (*.URL) are processed with XVT's CURL utility. The generated .RC file is then processed with RC.EXE, the standard Windows resource compiler.

XVT 3.0

```
curl -r rcwin -p -DWSWIN -DOSDOS -DCCMSC -DKPURL icxvtapp.url
rc -r icxvtapp.rc
```

XVT 2.0

```
curl -r          -DWSWIN -DOSDOS -DCCMSC -DKPURL icxvtap2.url
rc -r icxvtap2.rc
```

Linking

Windows Platform

Refer to the XVT documentation for linking procedures. Basically, you link as for any XVT/Windows application with one exception; you must additionally reference an INFOConnect library in the list of libraries used. There are several INFOConnect libraries to choose from depending on the memory model, version of XVT, and INFOConnect release level(s):

IcXvtM.lib	Resolves all references to the 3.0 release of ICS functions for XVT 3.0x (Medium memory model)
IcXvtL.lib	Resolves all references to the 3.0 release of ICS functions for XVT 3.0x (Large memory model)
IcXvt2M.lib	Resolves all references to the 3.0 release of ICS functions for XVT 2.0 (Medium memory model)
IcXvt2L.lib	Resolves all references to the 3.0 release of ICS functions for XVT 2.0 (Large memory model)
Ic2XvtM.lib	Resolves all references to the 2.0 and 3.0 releases of ICS functions for XVT 3.0x (Medium memory model)
Ic2XvtL.lib	Resolves all references to the 2.0 and 3.0 releases of ICS functions for XVT 3.0x (Large memory model)
Ic2Xvt2M.lib	Resolves all references to the 2.0 and 3.0 releases of ICS functions for XVT 2.0 (Medium memory model)
Ic2Xvt2L.lib	Resolves all references to the 2.0 and 3.0 releases of ICS functions for XVT 2.0 (Large memory model)

The .EXE file built by the linker is then combined with the .RES resource file built earlier.

The sample XVT application, IcXvtApp, was built with the following statements:

```
link @icxvtapp.lnk
rc icxvtapp.res icxvtapp.exe
```

Linker (LNK) files for XVT 3.0 applications

A typical LNK file (for example, icxvtapp.lnk) contains:

```
/noe/nod/map/line/co icxvtapp.obj
icxvtapp.exe
icxvtapp.map
icxvtM Mmxvtw xvtw xvtwtx Mlibcew libw
icxvtapp.def
```

IcXvtApp is the name of the application object file.

The /NOE option allows you to override an object file built into the libraries with one of your own. This option prevents the linker from searching the extended dictionary, which is an internal list of symbol locations that the linker maintains.

The /NOD option allows you to specify the Windows libraries explicitly. It tells the linker not to search any libraries specified in the object file to resolve external references.

Never use the /NOI option with Windows.

The /CO option is optionally used to prepare for debugging with the Microsoft CodeView debugger. Don't forget to remove it from the production version of your code.

IcXvtM.lib resolves all references to the INFOConnect functions for Medium model applications. Large model applications use IcXvtL.lib instead.

Mmxvtw.lib is the XVT library for medium memory model applications compiled with the Microsoft C compiler. A large model application built with Borland C++ would use LBxvtw.lib instead.

Xvtw.lib and Xvtwtx.lib are also XVT libraries.

Mlibcew.lib is Windows library for medium memory model.

Libw.lib is another Windows library.

Module definition (DEF) files for XVT 3.0 applications

A typical module definition file (for example, IcXvtApp.def) contains:

```
NAME      IcXvtApp
DESCRIPTION 'Sample INFOConnect/XVT Application'
EXETYPE   WINDOWS
STUB      'WINSTUB.EXE'
CODE      MOVEABLE DISCARDABLE
DATA      PRELOAD MOVEABLE MULTIPLE
SEGMENTS
;the entry point of the code (WinMain) must be PRELOAD
_TEXT     MOVEABLE DISCARDABLE PRELOAD

HEAPSIZE  8192
STACKSIZE  5120 ;recommended minimum for Windows apps

;All functions that will be called by any Windows routine
;MUST be exported.

EXPORTS
```

Linker (LNK) files for XVT 2.0 applications

A typical LNK file (for example, IcXvtAp2.lnk) contains:

```
/noe/nod/map/line/co icxvtap2.obj c:\idk\lib\win\icxevent c:\idk\lib\win\icxkey
icxvtap2.exe
icxvtap2.map
icxvt2MMmxvtw     Mlibcew libw
icxvtap2.def
```

IcXvtAp2 is the name of the application object file.

Icxevent is the INFOConnect event_hook routine and must always be included in the linkstream. If you are providing your own ICevent_hook, it must be provided in addition to Icxevent. See page 4 - 28 for more details.

Icxkey is a default key_hook module provided by INFOConnect. If you are providing your own key_hook routine it must replace icxkey. See page 4 - 28 for more details.

The /NOE option allows you to override an object file built into the libraries with one of your own (like whook.obj). This option prevents the linker from searching the extended dictionary, which is an internal list of symbol locations that the linker maintains.

The /NOD option allows you to specify the Windows libraries explicate. It tells the linker not to search any libraries specified in the object file to resolve external references.

Never use the /NOI option with Windows.

The /CO option is optionally used to prepare for debugging with the Microsoft CodeView debugger. Don't forget to remove it from the production version of your code.

IcXvt2M.lib resolves all references to the INFOConnect functions for Medium model applications. Large model applications use IcXvt2L.lib instead.

Mmxvtw.lib is the XVT library for medium memory model applications built with Microsoft C.

Mlibcew.lib is Windows library for medium memory model.

Libw.lib is another Windows library.

Module definition (DEF) files for XVT 2.0 applications

A typical module definition file (for example, IcXvtAp2.def) contains:

```
NAME     IcXvtAp2
DESCRIPTION 'Sample INFOConnect/XVT 2.0 Application'
EXETYPE   WINDOWS
STUB     'WINSTUB.EXE'
CODE     MOVEABLE DISCARDABLE
DATA     PRELOAD MOVEABLE MULTIPLE
SEGMENTS
;the entry point of the code (WinMain) must be PRELOAD
_TEXT    MOVEABLE DISCARDABLE PRELOAD

HEAPSIZE  8192
STACKSIZE  5120 ; recommended minimum for Windows apps

;All functions that will be called by any Windows routine
;MUST be exported.

EXPORTS
   xvt_WndProc       @1
   xvt_dialog        @2
   xvt_AbortDlgProc    @3
   xvt_AbortProc      @4
   xvt_FontFunc       @5
   ControlWndProc      @6
```

IcXvtApp - A sample XVT application

Finally we are ready to look at the complete source for IcXvtApp, an XVT application that uses INFOConnect services. Most of the code fragments used as examples in the preceding discussion were taken from this program.

Note: *The XVT toolkit changed substantially between XVT 2.0 and 3.0. IcXvtApp is written using the XVT 3.0 toolkit. Developers using XVT 2.0 should see sample program IcXvtAp2. It is the same program as IcXvtApp, but uses the XVT 2.0 API.*

What does IcXvtApp do?

IcXvtApp is a simple communications program. It opens an INFOConnect session and allows the user to enter messages to be sent across the communications path using dialog boxes. Received messages are also displayed using dialog boxes. XVT functions are used for all presentation services (for example, windows and dialogs) and INFOConnect functions are used for all communications services. No platform-specific functions (such as Windows or MS-DOS functions) are used.

Source file descriptions

File	Description
IcXvtApp.c	C-language source for XVT 3.0x
IcXvtApp.h	Header file for XVT 3.0x
IcXvtApp.url	Resource file for XVT 3.0x
IcXvtApp.def	Module-definition file used to link for XVT 3.0x
IcXvtApp.ico	Icon file for XVT 3.0x
IcXvtAp2.c	C-language source for XVT 2.0
IcXvtAp2.h	Header file for XVT 2.0
IcXvtAp2.url	Resource file for XVT 2.0
IcXvtAp2.def	Module-definition file used to link for XVT 2.0
IcXvtAp2.ico	Icon file for XVT 2.0

All source files needed to build this application are provided with the IDK in the sample directory.

Windows platform

To build the windows version of this application, do:

```
nmake -f makefile PROGRAM=icxvtapp XVT=y
```

XVT 2.0 users

```
nmake -f makefile PROGRAM=icxvtap2 XVT=y XVTVER=0x0200
```

Couple - An XVT Application that Connects Two INFOConnect Sessions

The Couple application opens two INFOConnect sessions and routes all traffic from one to the other. It is the counterpart to the LOCAL external interface which connects two sessions at the external interface layer. Couple prompts the user for two path names unless they are passed in on the command line like this:

```
couple.exe -ppathABC -ppathDEF
```

What is Couple used for?

Uses for Couple come up in many situations. Here are just a few ideas:

- Couple can be used as the skeleton for a server application that connects pairs of INFOConnect paths.
- Working with INFOConnect accessories (using ic_open_accessory) often involves connecting two sessions as in Couple; one session is for a host connection and the other is for the accessory connection. The IcOpenAc sample program further demonstrates this.

How does Couple work?

Couple opens both sessions, allocates a single receive buffer for each and goes into receive mode for each session. When session A gets an E_IC_RCV_DONE event, Couple transmits the received buffer on session B. When session B gets the E_IC_XMT_DONE event, Couple goes back into receive mode on session A. Similar processing occurs when session B gets a E_IC_RCV_DONE event.

Source file descriptions

Couple.c	C-language source
Couple.h	Header file
Couple.url	Resource file
Couple.def	Module-definition file used to link
Couple.ico	Icon file

All source files needed to build Couple are provided with the IDK in the SAMPLE directory.

Windows platform

To build the windows version, do:

```
nmake -f makefile PROGRAM=couple XVT=y
```

IcOpenAc - An XVT Application that Opens an INFOConnect Accessory

IcOpenAc demonstrates how to connect to an INFOConnect accessory using ic_open_accessory. Two sessions are opened; one to a host and one to an accessory. IcOpenAc routes all datacomm traffic between the two open sessions. Selected messages from the host are intercepted and processed by IcOpenAc instead of passing them on to the accessory. Accessory developers can use IcOpenAc to test the behavior of their accessory when it is actually invoked as an accessory via ic_open_accessory.

The user of IcOpenAc specifies the pathname of a host connection, an INFOConnect accessory name, the desired window state of the invoked accessory, and an initial command to be sent to the host on behalf of the accessory. Here is the format of the options recognized by IcOpenAc:

-p	A pathname
-a	An accessory name
-o	Options recognized by the invoked accessory
-w	Option to control the accessory's initial window state
-c	An initial command or statement to be transmitted across the session

```
icopenac.exe -pMYPATH-aicxvtapp-cmyfirstmessagetotransmit
```

How does IcOpenAc work?

IcOpenAc is based on the model for connecting two sessions demonstrated in the Couple sample application. After opening a path to the host and another path to an accessory, each session is put in receive mode. An initial message is transmitted to both sessions, then all traffic received from each session is transmitted to the other as in Couple. The traffic from the host is monitored for a particular message which is then intercepted and processed by IcOpenAc. IcOpenAc is coded to watch for a special termination message, <ESC>D, after which it terminates. The host program that issues the <ESC>D message is not provided in the IDK.

IcOpenAc is also an example of an XVT application that includes native Windows code, too.

Source file descriptions

IcOpenAc.c	C-language source
IcOpenAc.h	Header file
IcOpenAc.url	Resource file
IcOpenAc.def	Module-definition file used to link
IcOpenAc.ico	Icon file

All source files needed to build IcOpenAc are provided with the IDK in the sample directory.

Windows platform

To build the windows version, do:

```
nmake -f makefile PROGRAM=icopenac XVT=y
```

Section 5
Writing INFOConnect/DosLink Applications

DosLink is an INFOConnect solution that allows DOS applications to make connections to other computers using INFOConnect Connectivity Services. The DOS application requires a DOS window in enhanced mode Windows. This capability is only available on 386 class machines and higher.

What is the DOS application's perspective of INFOConnect DosLink services?

After reviewing Section 2, you may be wondering how events and window handles fit into the world of DOS programming. The event-driven model is usually foreign to traditional DOS programming techniques. DOS programs often use a polling style while waiting for external events (for example, the arrival of a datacomm message) to occur. Somewhere in the main program loop, a routine is polled to determine if an event has occurred.

Event processing

The INFOConnect DosLink API is a subset of the INFOConnect Accessory API and is made up of functions and events. In addition to making INFOConnect function calls, your application must process INFOConnect events returned to it. Typically, this event processing is coded in a routine named IcEventHandler. IcEventHandler itself can be "handled" in two ways. It can be registered as a callback routine during application initialization and called directly by INFOConnect later as events occur. Or, more convenient for DOS programming, you can explicitly poll for INFOConnect events during your main program loop and pass any available events directly to the event handler. These two approaches are referred to as *event callbacks* and *event polling*. The approach used is specified during initialization.

Window handles

Another characteristic of the DosLink API is the use of window handles in some of the function parameter lists. A Windows program naturally has a window handle, but what do window handles have to do with DOS programs? Window handles are in the DosLink API for consistency with the other INFOConnect APIs. When an INFOConnect session is opened, it is associated with a window handle. A special function, IcCreateHwnd, is provided for DosLink applications to obtain a window handle.

Buffer handles

Finally, the DosLink API requires datacomm buffers to be passed to INFOConnect using handles. There are several approaches available to DosLink applications for obtaining buffers and buffer handles. You can use INFOConnect functions to actually allocate your datacomm buffers. These routines return buffer handles right from the start. Or, for existing DOS programs that are using non-INFOConnect memory allocation routines, the DosLink API provides a way to assign handles to far pointers.

Basic Procedures for DosLink Applications

This section shows how to use the INFOConnect DosLink API to accomplish basic data communications tasks in your application.

Initializing INFOConnect Connectivity Services

All INFOConnect events, functions and types are defined in IcDos.h. Include this header file at the beginning of your application.

Call IcInitIcs to initialize with INFOConnect Connectivity Services. IC_VERSION_3_0 and IC_REVISION_3_0 provide version control designating to use the INFOConnect 3.0 API.

Sample code

```
#include <icdos.h>;
#include <assert.h>;

IC_RESULT icerror;

void main(int argc, char** argv)
{
  /* Initialize INFOConnect Interfaces */
  icerror = IcInitIcs(IC_VERSION_3_0, IC_REVISION_3_0);
  if (icerror != IC_OK) {
    assert(FALSE);
  }
  ...
}
```

Opening a Session

Before you can send data with INFOConnect, you must open an INFOConnect session. Here are the steps to open a session and allocate datacomm buffers.

Initialize INFOConnect

Initialize INFOConnect Connectivity Services using IcInitIcs if you haven't done so already.

Obtain a window handle

Call IcCreateHwnd to create a window handle. The DosLink API contains window handles to remain consistent with the other INFOConnect APIs. You will need a window handle to open a session later. Active sessions are associated with window handles. You can use any convenient string with IcCreateHwnd; the application name is a convenient choice. The sample code uses `argv[0]`, which is the command line parameter for the program name.

Create a session handle

Call IcCreateSession to create a session handle. This handle must be passed to IcOpenSession before transmit or receive requests can be sent across the connection.

Call IcOpenSession

Call IcOpenSession to request a path to be opened. For the 'hWnd' and 'lpsession' parameters, use the handles returned from IcCreateHwnd and IcCreateSession. You must supply a path name for the 'host' parameter. If you are familiar with writing INFOConnect/Windows or INFOConnect/XVT applications, you may be used to passing a NULL string for the path name and let INFOConnect prompt the user for a path. This functionality is not available to DosLink applications.

If INFOConnect returns from IcOpenSession with IC_OK, the session establishment process has successfully started, but not completed. INFOConnect passes an IC_SESSIONESTABLISHED message to your event function later when the session is actually established. You must be careful not to use the pending session handle before the IC_SESSIONESTABLISHED message occurs. No IC_SESSIONESTABLISHED message is generated if INFOConnect does not return with IC_OK on the IcOpenSession call.

Establish an INFOConnect event handler

IcOpenSession is the first INFOConnect function that results in an event being sent to your application. As discussed earlier, you need a routine, typically named IcEventHandler, to process these events. If you are a Windows programmer, you may notice that the function prototype for IcEventHandler looks like the Window procedure required in Windows applications.

There are two styles for INFOConnect event handlers: polling and callback. The sample code fragment below uses a "polling style" event handler. Function IcGetNextEvent retrieves new INFOConnect events from the event queue which are then passed to the event handler.

Handle the IC_SESSIONESTABLISHED event

Add a test for IC_SESSIONESTABLISHED to your event handler. It's a good idea to create and set a global boolean variable to indicate the current state of your session. This is primarily needed during the time between the IcOpenSession call and before the IC_SESSIONESTABLISHED message is returned to your application. The sample code uses *bSessionEst* within the session structure and doesn't try to transmit or receive data until *bSessionEst* is TRUE.

If the IC_SESSIONESTABLISHED message does not contain IC_OK (indicating an error of some kind), you must call IcCloseSession to release the pending session handle.

If you are familiar with INFOConnect application development, you may be used to allocating buffers now and initiating transmit and/or receive requests. As described later in this section, DosLink applications are dependent on an INFOConnect server accessory, DosLinkS, to open the requested path on behalf of the application. DosLink applications must wait for some special status messages from the server accessory.

Define an enumeration indicating 'connection state'

It's useful to define an enumeration type to track the state of the DosLink connection. The sample code uses

```
enum{NOTESTABLISHED,BROKEN,JOINED,SERVER]ConnectState
```

for this purpose. The connection state is communicated to the application with IC_STATUS/IC_CONNECT_... events. Connection states are related to the DosLinkS server accessory. For now, suffice it to say that DosLink applications cannot transmit or receive across a connection until the connection state has reached the SERVER state.

Handle the IC_STATUS/IC_CONNECT_SERVER event and optionally allocate buffers

Add a test for IC_STATUS to your INFOConnect event handler to process the IC_CONNECT_SERVER message. You can optionally allocate INFOConnect buffers now (as in the sample code fragment below) for transmitting and receiving data. *If your DOS application is already allocating buffers using standard C runtime library routines, these buffers can be assigned handles using other DosLink API calls. This is discussed later.* When determining what buffer size to use, call IcGetSessionInfo to find the maximum buffer size the underlying communications software can support. Some connections may support very large buffers, so you may want to put a ceiling on the buffer size as shown in the code fragment. Be sure to check for NULL on the allocation requests.

Sample code

The following code fragment illustrates everything covered in the above discussion.

```
#include <icdos.h>;
#include <assert.h>;

struct aSession { /* INFOConnect session structure */
  HWND hWnd;
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
  enum {NOTESTABLISHED, BROKEN, JOINED, SERVER} ConnectState;
  unsigned uBufsize;
} s;

IC_RESULT icerror;
IC_SINFO sinfo;

void main(int argc, char **argv)
{
  /* Initialize INFOConnect Interfaces */
  icerror = IcInitIcs(IC_VERSION_3_0, IC_REVISION_3_0);
  assert(icerror == IC_OK);
  /* Initialize the INFOConnect session structure. */
  s.hWnd = IcCreateHwnd(argv[0]); /* program name */
  s.hSession = NULL_HIC_SESSION;
  s.hXmtBuf = NULL;
  s.hRcvBuf = NULL;
  s.bSessionEst = FALSE;
  s.ConnectState = NOTESTABLISHED;
  s.uBufsize = 0;
```

```
  icerror=IcCreateSession(&s.hSession);
  assert(icerror==IC_OK);
  icerror=IcOpenSession(s.hWnd,"mypath",&s.hSession);
  assert(icerror==IC_OK);

  do {     /* This is the main loop in your application. */
   HANDLE   hWnd;
   unsigned message;
   IC_RESULT icresult;

   IcGetNextEvent(s.hSession,&hWnd,&message,&icresult);
   IcEventHandler(hWnd,message,s.hSession,icresult);
  } while (s.bSessionEst);
}

long FAR PASCAL IcEventHandler(HWND     hWnd,
            unsigned  message,
            HIC_SESSION hSession,
            IC_RESULT  icresult)
{
  switch (message) {
   case IC_SESSIONESTABLISHED:
    if IC_CHECK_RESULT_SEVERE(icresult)
     IcCloseSession(hSession);
    else {
     s.bSessionEst=TRUE;
     s.ConnectState=BROKEN;
    break;
   case IC_STATUS:
    if (icresult==IC_CONNECT_JOINED)
     s.ConnectState=JOINED;
    else if (icresult==IC_CONNECT_SERVER) {
     s.ConnectState=SERVER;
     icerror=IcGetSessionInfo(hSession,&sinfo);
     assert(icerror==IC_OK);
     s.uBufsize=min((unsigned)sinfo.max_size,MAXBUFSIZE);
     s.hXmtBuf=IcAllocBuffer(s.uBufsize);
     s.hRcvBuf=IcAllocBuffer(s.uBufsize);
    }
    else if (icresult==IC_CONNECT_BROKEN)
     s.ConnectState=BROKEN;
    break;
   case IC_NULLEVENT:
    break;
  }
  return(NULL);
}
```

Transmitting a Buffer

The IcXmt function attempts to transmit a buffer of data. This function is asynchronous in nature; it returns immediately to the application before the transmit is completed. A return value of IC_OK means the transmit request has been initiated. When the transmit finishes, one of two messages is passed to your INFOConnect event handler : IC_XMTDONE or IC_XMTERROR. The basic steps to follow are:

Don't transmit prematurely

Don't transmit over a session before two INFOConnect messages have been sent to your application: the IC_SESSIONESTABLISHED message and the IC_CONNECT_SERVER status message. Also, don't transmit while a previous transmit request is still pending for that session. The sample code below uses several variables, *bSessionEst*, *ConnectState* and *nXmtTries*, to manage these conditions.

Allocate and lock a transmit buffer

If you used IcAllocBuffer to allocate the transmit buffer, lock the buffer with IcLockBuffer.

Prepare the transmit buffer

If you used IcAllocBuffer to allocate the transmit buffer, lock the buffer with IcLockBuffer. Fill the transmit buffer and unlock it with IcUnlockBuffer.

If you are not using IcAllocBuffer to allocate the transmit buffer, obtain a handle using IcCreateHandle and/or IcHandleOffset.

Call IcXmt

Pass the buffer to INFOConnect with IcXmt.

You should declare a variable to indicate a transmit request is outstanding. Check this variable to avoid transmitting a second buffer before a previous transmit request has finished.

One approach is to simply use a boolean to indicate outstanding transmit requests. Don't transmit unless the boolean is clear, then set the boolean after calling IcXmt successfully. Clear the boolean when an IC_XMTDONE or IC_XMTERROR message is received.

The sample program, IcDosApp, uses a different approach. A counter named nXmtTries indicates when a transmit request is pending and will also be useful later to manage transmit errors and retries.

Handle the IC_XMTDONE and IC_XMTERROR messages

Add tests for IC_XMTDONE and IC_XMTERROR to your event handler. Set your variable to indicate that transmits are now allowed.

IC_XMTDONE messages contain the buffer handle and buffer length of the transmitted data.

IC_XMTERROR messages contain an IC_RESULT with the reason for the transmit failure. Don't ignore these messages. You may want to display the error or retry the transmit request. See page 5 - 19 for more on handling data communications errors.

Sample code

```
struct aSession {
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
  enum {NOTESTABLISHED, BROKEN, JOINED, SERVER} ConnectState;
  unsigned uBufsize;
  int nXmtTries;
}s;

void DoTransmit(char localbuf[])
{
  if ((s.nXmtTries>0) || (!s.bSessionEst) || (s.ConnectState<SERVER)) {
    ...beep or issue message indicating not ready to transmit
  }
  else {
    if ( (buf=IcLockBuffer(s.hXmtBuf)) == NULL) {
      assert( FALSE);
    }
    else {
      _fstrcpy( buf, localbuf);
      IcUnlockBuffer( s.hXmtBuf);
      icerror = IcXmt( s.hSession, s.hXmtBuf,
              strlen(localbuf));
      if (! IC_CHECK_RESULT_SEVERE(icerror))
        s.nXmtTries = 1;
      else
        assert(FALSE); /* display icerror */
    }
  }
}

long FAR PASCAL IcEventHandler( HWND hWnd,
              unsigned message,
              HIC_SESSION hSession,
              IC_RESULT icresult)
{
  switch (message) {
    case IC_XMTDONE:
      s.nXmtTries = 0; /* no outstanding transmits */
      break;
    case IC_XMTERROR:
      ...see discussion on handling datacomm errors
      break;
  }
  return( NULL);
}
```

Receiving a Buffer

The IcRcv function requests a buffer of data. This function is asynchronous in nature; it returns immediately to the application before the receive is completed. A return value of IC_OK means the receive request was initiated. When the receive request finishes, one of two messages is passed to your event handler: IC_RCVDONE or IC_RCVERROR. The basic steps to follow are:

Don't issue a receive request prematurely

Don't use a session before two INFOConnect messages have been sent to your application: the IC_SESSIONESTABLISHED message and the IC_CONNECT_SERVER status message. Don't make a receive request for a session while a previous receive request is still pending for that session. The sample code below uses several variables, *bSessionEst*, *ConnectState* and *nRcvTries*, to manage these conditions.

Allocate a receive buffer

Allocate a receive buffer if you haven't already done so. IcAllocBuffer is recommended for datacomm buffer allocation.

Call IcRcv

Pass the buffer to INFOConnect with IcRcv.

You should declare a variable to indicate that a receive request is outstanding. Check this variable to avoid making a second receive request while the previous one is still outstanding.

One approach is to use a global boolean to indicate outstanding receive requests. Don't issue a receive request unless the boolean is clear, then set the boolean after calling IcRcv successfully. Clear the boolean when an IC_RCVDONE or IC_RCVERROR message is received.

The sample program, IcDosApp, uses a different approach. A counter named nRcvTries indicates when a receive request is pending and will also be useful later to manage receive errors and retries.

Handle the IC_RCVDONE and IC_RCVERROR messages

Add tests for IC_RCVDONE and IC_RCVERROR messages to your event hander. Set your variable to indicate that new receive requests are now allowed.

IC_RCVDONE messages contain the buffer handle and buffer length of the received data.

IC_RCVERROR messages contain an IC_RESULT with the reason for the receive failure. Don't ignore these messages. You may want to display the error or retry the receive request. See page 5 - 19 for more on handling data communications errors.

Sample code

```
struct aSession {
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
  unsigned uBufsize;
  int nRcvTries;
}s;

void DoReceive(void)
{
  if ((s.nRcvTries>0) || (!s.bSessionEst) || (s.ConnectState<SERVER)) {
    ...beep or issue error message indicating not ready to receive
  }
  else {
    icerror = IcRcv(s.hSession, s.hRcvBuf, s.uBufsize);
    if (!IC_CHECK_RESULT_SEVERE(icerror))
      s.nRcvTries = 1;
    else
      assert(FALSE); /* display icerror */
  }
}

long FAR PASCAL IcEventHandler( HWND hWnd,
            unsigned message,
            HIC_SESSION hSession,
            IC_RESULT icresult)
{
  switch (message) {
    case IC_RCVDONE:
      s.nRcvries = 0; /* no outstanding receive */
      break;
    case IC_RCVERROR:
      ...see discussion on handling datacomm errors
      break;
  }
  return(NULL);
}
```

Allocating and Using Datacomm Buffers

This section contains guidelines to help you better manage your application's datacomm buffers. The IcRcv and IcXmt functions are asynchronous in nature; they return immediately before the datacomm request is actually completed. This means your application must be careful about accessing the datacomm buffers that were passed to these functions. The following list will help you better manage your application's data communications.

Allocate datacomm buffers with IcAllocBuffer if possible

Normally, INFOConnect applications must use IcAllocBuffer and IcFreeBuffer to allocate data communications buffers to ensure that the buffers have the proper system attributes to be shareable across applications. Use these routines if you can because it's simpler. Most of the sample code fragments use IcAllocBuffer.

Using INFOConnect routines for buffer allocation may be inconvenient for existing DOS applications, therefore, there are DosLink functions that convert far pointers to INFOConnect handles. IcCreateHandle and IcDestroyHandle convert paragraph-aligned, normalized far pointers to and from INFOConnect handles. A *paragraph-aligned* address is a segment/offset pair in which the offset is a multiple of 16. A *normalized* address is a segment/offset pair in which the offset is less than 16. You must normalize buffer addresses yourself before calling IcCreateHandle.

If your buffers are not paragraph-aligned you cannot use IcCreateHandle. Instead, use IcHandleOffset immediately before calling IcXmt or IcRcv to temporarily convert non-paragraph-aligned pointers to an INFOConnect-compatible form.

INFOConnect datacomm is asynchronous

Calls to IcRcv and IcXmt will return to your application immediately before the request is actually completed. Later, a message will be sent to your event handler when the request is completed. Until this message is returned, your request is referred to as pending.

Don't use pending buffers

Do not access a buffer that is associated with a pending request. It's a good idea to define and set a variable for each buffer that tracks when the buffer is associated with a pending request.

Don't issue a receive or transmit request until the IC_SESSIONESTABLISHED message and the IC_CONNECT_SERVER status message have been returned.

Don't issue a receive request while a pending receive request exists for the same session. The state of the first receive request is undefined if this happens. Your application cannot assume that all external interfaces will behave similarly for this situation. The DosLink interface happens to queue all requests without verifying whether the request uses a pending buffer or not.

These same warnings hold true for premature transmit requests.

Use one receive buffer and one transmit buffer

You can have as many datacomm buffers as you like, but a session can only have one active transmit buffer and one active receive buffer at a time. Therefore, it's recommended to allocate one receive buffer and one transmit buffer.

You can use one buffer for both transmitting and receiving, but for maximum interoperability use separate buffers. Your application can be more responsive using separate buffers since it can keep a receive request pending while waiting for a pending transmit request to complete.

Cancel pending requests with IcLcl

Pending requests can be canceled by using IcLcl. You must wait until an IC_LCLRESULT message is returned to your application on behalf of the canceled messages before you can safely access any datacomm buffers.

Don't ignore errors

Don't ignore error messages (IC_XMTERROR and IC_RCVERROR). You may want to retry the request at least some number of times. See page 5 - 19 for more on handling data communications errors.

Error Handling

Nearly every INFOConnect function and message returns an IC_RESULT type indicating the success of the function or message. Any value other than IC_OK indicates an error. In addition to the value returned from INFOConnect functions, asynchronous error events can be returned to your event handler with the IC_ERROR message. The four categories of errors and recommended actions for each follow:

IC_ERROR_INFO	Log this error if the application has a log file. Don't bother displaying a message to the user. The requested function was completed.
IC_ERROR_WARNING	The requested function was completed, but something unusual or noteworthy happened. The application can choose to log or display this error.
IC_ERROR_SEVERE	The requested function did not complete successfully. The user should usually see this error.
IC_ERROR_TERMINATE	Display this error and close the session.

The following are some guidelines for error handling. The sample code fragment below follows all these guidelines.

- Don't display anything for IC_OK.
- Support IC_ERROR_TERMINATE_NOMSG errors. This is a special error used by INFOConnect to close sessions without displaying anything to the user.
- Close the session for errors of type IC_ERROR_TERMINATE. IC_GET_RESULT_TYPE is an INFOConnect macro used to extract the error-type from 'icerror'.
- IcDefaultErrorProc, the INFOConnect default error procedure available to standard INFOConnect applications, is not accessible to DosLink applications. You must display errors yourself.
- IcGetString, the INFOConnect function for retrieving error text, is currently not accessible to DosLink applications.

Sample code

Here is a code fragment that makes a transmit request and passes any errors to a general error handling routine called HandleIcError.

```
icerror = IcXmt(hSession, s.hXmtBuf, s.nXmtLen);
if IC_CHECK_RESULT_SEVERE(icerror)
  HandleIcError(hWnd, hSession, NULL, icerror);

void HandleIcError(HWND      hWnd,
          HIC_SESSION session,
          unsigned   message,
          IC_RESULT  icerror)
{
  /* This routine does basic processing of INFOConnect
    errors. It uses standard Console I/O routines to display
    error numbers. Sessions are closed upon
    detection of errors of type IC_ERROR_TERMINATE.
  */
  if ((icerror != IC_OK) &&
    (icerror != IC_ERROR_TERMINATE_NOMSG))
  {
    if (message != NULL)
      _cprintf("INFOConnect error: %lx message type: %x \r\n",
          icerror, message);
    else
      _cprintf("INFOConnect error %lx \r\n", icerror);
  }
  if ((IC_GET_RESULT_TYPE(icerror) & IC_ERROR_MASK) ==
                IC_ERROR_TERMINATE)
    IcCloseSession(session);
}
```

Closing a Session

Call IcCloseSession to end an INFOConnect session. This function returns immediately. Message IC_SESSIONCLOSED is passed to the event handler when the session is actually closed. The basic steps to follow are:

Clear the boolean indicating 'session establishment'

The sample code uses a variable named bSessionEst for this purpose. This variable was originally set when the IC_SESSIONESTABLISHED message was received by the application.

Call IcCloseSession and IcDestroySession and deallocate buffers

Use INFOConnect memory management routines to free datacomm buffers. It's also a good idea to clear the buffer handle variables with NULL.

Destroy the window handle

If no other INFOConnect sessions are associated with the window handle, destroy it with IcDestroyHwnd.

Handle the IC_SESSIONCLOSED message

In addition to the expected IC_SESSIONCLOSED message that is delivered to your application after IcCloseSession, your application must be prepared for unsolicited IC_SESSIONCLOSED messages. If the user clears your application's session from the INFOConnect manager window, an IC_SESSIONCLOSED message is delivered to your event handler. Typically, your application should terminate gracefully.

Sample code

```
struct aSession {
  HWND hWnd;
  HIC_SESSION hSession;
  HANDLE hXmtBuf;
  HANDLE hRcvBuf;
  BOOL bSessionEst;
}s;

void DoIcClose(void)
{
  s.bSessionEst=FALSE;
  if(s.hSession != NULL_HIC_SESSION) {
    IcCloseSession(s.hSession);
    IcDestroySession(s.hSession);
    s.hSession = NULL_HIC_SESSION;
  }
  if(s.hWnd != NULL) {
    IcDestroyHwnd(s.hWnd);
    s.hWnd = NULL;
  }
  if(s.hXmtBuf != NULL) {
    IcFreeBuffer(s.hXmtBuf);
    s.hXmtBuf = NULL;
  }
  if(s.hRcvBuf != NULL) {
    IcFreeBuffer(s.hRcvBuf);
    s.hRcvBuf = NULL;
  }
}
```

Advanced Procedures for DosLink Applications

Canceling Pending Requests

Many applications typically stay in receive mode so they're ready to respond to any messages from the partner activity. Occasionally, it may be necessary to cancel this outstanding receive request.

The IcLcl function selectively cancels pending actions. A parameter indicates what is to be canceled: the pending transmit requests, pending receive requests, or both. Your application must wait for an IC_LCLRESULT message before safely accessing the datacomm buffer associated with the canceled actions.

```
/* The following definitions are in ICDEF.H      */
/* You do NOT need to define these in your application */

#define IC_LCL_RCV  1
#define IC_LCL_XMT  2
#define IC_LCL_RCVXMT (IC_LCL_RCV | IC_LCL_XMT)
#define IC_LCL_CLOSESESSION 4
```

Sample code

The following code cancels any pending receive or transmit requests.

```
icerror = IcLcl(hSession, IC_LCL_RCVXMT);
if IC_CHECK_RESULT_SEVERE(icerror) {
  assert(FALSE);
```

Handling Data Communications Errors

Data communications errors show up as IC_XMTERROR and IC_RCVERROR messages in your event handler. A good application will act on these messages. Often, you may just want to retry the request for some number of times.

The following code fragment will retry datacomm errors five times before displaying an error to the user. Two variables, *nXmtTries* and *nRcvTries*, are the basis of this technique. They are incremented when a request results in an error. They are set to zero after successful completion (IC_XMTDONE and IC_RCVDONE) indicating there is no longer an outstanding request.

```
#define MAXRETRIES 5

struct aSession {
 HWND hWnd;
 HIC_SESSION hSession;
 HANDLE hXmtBuf;
 HANDLE hRcvBuf;
 BOOL bSessionEst;
 unsigned uBufsize;
 int nXmtLen;
 int nXmtTries;
 int nRcvTries;
}s;

long FAR PASCAL IcEventHandler(HWND     hWnd,
           unsigned  message,
           HIC_SESSION hSession,
           IC_RESULT  icresult)
{
 switch (message) {
  case IC_XMTDONE:
   s.nXmtTries = 0; /* no outstanding transmits */
   break;
  case IC_RCVDONE:
   s.nRcvTries = 0; /* no outstanding receives */
   break;
  case IC_XMTERROR:
   if (++(s.nXmtTries) > MAXRETRIES) {
    HandleIcError(hWnd, hSession, message, icresult);
    s.nXmtTries = 1;
   }
   /* try transmit again */
   icerror = IcXmt(hSession, s.hXmtBuf, s.nXmtLen);
   if IC_CHECK_RESULT_SEVERE(icerror) {
    HandleIcError(hWnd, hSession, NULL, icerror);
    s.nXmtTries = 0;
   }
   break;
```

```
    case IC_RCVERROR:
      if(++(s.nRcvTries)>MAXRETRIES){
        HandleIcError(hWnd,hSession,message,icresult);
        s.nRcvTries=1;
      }
      /* try receive again */
      icerror=IcRcv(hSession,s.hRcvBuf,s.uBufsize);
      if IC_CHECK_RESULT_SEVERE(icerror){
        HandleIcError(hWnd,hSession,NULL,icerror);
        s.nRcvTries=0;
      }
      break;
  }
  return(NULL);
}
```

Running with Old Versions of INFOConnect

The basic philosophy of INFOConnect version control is "Old applications must still run with new versions of the Manager, but new applications are not required to run with old Managers." In fact, the Manager normally refuses to run applications built with a version of the IDK that is newer than the Manager itself. The assumption is that new applications might make new API calls that an old Manager doesn't know about.

If you are upgrading an existing application to a new version of the IDK, and can function properly without making new INFOConnect API calls, this section describes how to build your application with the latest IDK and initialize with different versions of the Manager at run time. Your application must remember which version of the Manager it is executing with, and only make appropriate API calls known by that version of the Manager.

The version of an application is normally "marked" on the initial call to IcInitIcs:

```
icerror=IcInitIcs(IC_VERSION_3_0,IC_REVISION_3_0);
if IC_CHECK_RESULT_SEVERE(icerror)
  // error processing. INFOConnect services are not available.
```

IC_VERSION_3_0 and IC_REVISION_3_0 are defined in IcDef.h. A new IC_VERSION_... is updated at major release levels (for example, release 1.0, 2.0, 3.0). IC_REVISION_... has a finer granularity -- it is updated each time the INFOConnect APIs are extended or changed.

However, nothing prevents you from calling IcInitIcs again with values from an earlier version of the IDK. The following code sequence is one possibility for initializing with the Manager.

```
int icversion; /* global variable */
IC_RESULT icerror;

if ((icerror=IcInitIcs(IC_VERSION_CHECK,IC_REVISION_1_0)) == IC_OK)
  icversion = IC_VERSION_1_0;
else if ((icerror=IcInitIcs(IC_VERSION_3_0,IC_REVISION_3_0)) == IC_OK)
  icversion = IC_VERSION_3_0;
else if ((icerror=IcInitIcs(IC_VERSION_2_0,IC_REVISION_2_0)) == IC_OK)
  icversion = IC_VERSION_2_0;
else {
  IcDefaultErrorProc(hWnd, NULL, NULL, icerror);
  /* INFOConnect services are unavailable!! */
}
```

At the end of this code fragment, *icversion* is set to the level of the Manager you are running with.

CAUTION

Be careful to check *icversion* before making any API calls undefined in older versions of INFOConnect. Otherwise, you will encounter unexpected behavior.

Here is the recommended way to isolate new API calls in your application:

```
if (icversion >= IC_VERSION_3_0)
{
  /* call some new 3.0 INFOConnect API */
}
else {
  /* alternative action using pre 3.0 API calls */
}
```

A Closer Look at the DosLink Solution

The DosLink solution is implemented by a collection of components as shown in the following diagram. The goal of DosLink is to allow a DOS application to connect to the host using an INFOConnect *target path*. The DOS application appears to be manipulating the target path directly, but there are actually several other components involved. The dotted line represents the route followed by data on its journey between the application and the host.

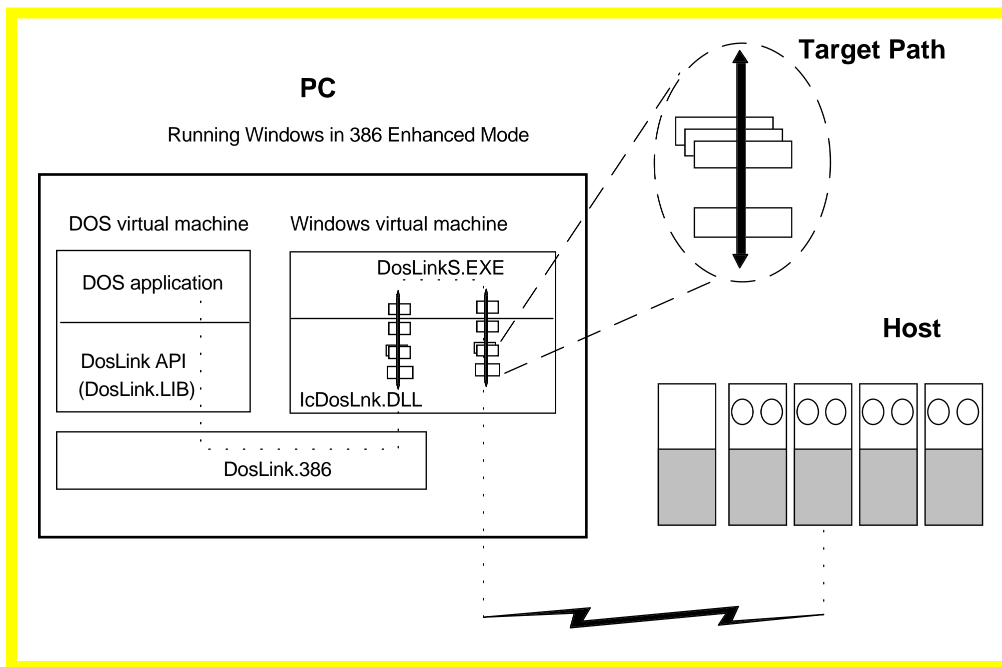


Figure 5–1. The DosLink Solution

Each component along the dotted line, from the DOS application to the host, is described in the following paragraphs.

DOS (or DosLink) application

This is an updated version of a DOS application that uses the DosLink API to open and use the *target path*. Everything in between the DOS application and the target path is transparent to the DOS application. IcDosApp is a sample DOS application provided in the IDK that operates at this position in the architecture.

If you have a collection of DOS applications that uses a common data communications API, perhaps with a DOS Terminate and Stay Resident (TSR), you can modify the TSR to use the DosLink API. IcBDrive is a sample TSR provided in the IDK that maps a subset of the BDrive DOS-based data communications API to the INFOConnect Accessory API.

DosLink API

IcDos.h is a C-header file that contains the function prototypes for the INFOConnect DosLink services available to DOS applications. When you link your application, DosLink.lib resolves references to the DosLink API. The link process binds an object module from DosLink.lib to your application (IcDos.obj). This object module actually interfaces with the DosLink virtual device driver below. You do not need the Windows SDK to link your application. DosLink applications only require a C-compiler and the IDK to build.

DosLink.386

DosLink.386 is a Virtual Device Driver (VxD) distributed with the basic INFOConnect Connectivity Services package. Virtual devices allow communication between different virtual machines when a machine is running in 386 enhanced mode. When Windows runs in enhanced mode, it starts each DOS session in a different virtual machine.

IcDosLnk.dll

IcDosLnk is an INFOConnect External Interface Library that maps the standard INFOConnect Accessory API on to the interface used by DosLink.386 virtual device. IcDosLnk is also distributed with the basic INFOConnect Connectivity Services package.

DosLinkS.exe

DosLinkS is a server application that joins two INFOConnect paths together and routes all datacomm traffic between them. Each DosLink application requires the DosLinkS server to manage a pair of paths in this manner. The two icons in the architecture diagram represent these two paths. One of the paths is the target path that the DOS application thinks it is directly manipulating. The other path is named DosLink and is internally configured and maintained by DosLinkS.

DosLinkS is an INFOConnect accessory distributed with the basic INFOConnect Connectivity Services package. It is based on the CoupleW sample application provided with the IDK. However, where CoupleW only connects a single pair of paths, DosLinkS can manage many pairs of paths. Each DosLink application running on the machine results in a pair of paths for DosLinkS to manage. DosLinkS must be up and running before any DosLink applications try to open paths. This can be done in Windows 3.1 by dragging the DosLinkS icon into the Startup program group or in Windows 3.0 by updating WIN.INI:

```
[windows]
load=c:\infoconn\doslinks.exe
```

Target path

The target path can be any INFOConnect path configured on the PC. Therefore, your DOS application has access to all the transports available through INFOConnect now and in the future.

Compiling and Linking

Note: *Before compiling and linking any INFOConnect applications, please review the System Verification Checklist in the Installation section.*

Compiling and Linking Environment

DosLink applications are Real-mode DOS programs, not Windows programs. Therefore, you do not need the Windows SDK to build DosLink applications. In fact, if you are building DosLink applications on a Windows development machine, you must be careful that no Windows-related libraries or header files are accidentally pulled in during the compile and link process.

Also, some C compilers (for example, Microsoft C 5.1 and 6.0) are capable of building both real-mode DOS programs and protected-mode OS2 programs. The default environment, real mode or protected mode is selected during compiler installation. If the following message is displayed when you run a Doslink application:

This program cannot be run in DOS mode

your default environment is probably OS2/protected mode. Typically, there are different libraries for the two modes. Microsoft compilers use names like SLIBCE**R**.LIB and SLIBCE**P**.LIB for the real and protected mode, small model libraries. Refer to your compiler installation documentation for information pertaining to real mode and protected mode.

Memory Models

The Medium or Small memory models are recommended, but the INFOConnect architecture places no constraints on memory model usage.

Include files

The function prototypes for the DosLink API are declared in the IcDos.h header file.

C compiler options

The sample programs provided with the INFOConnect Development Kit are built with the following options using the Microsoft C compiler.

```
cl -c -AS -Gs -W3 -Oils -Zp -FPc icdosapp.c
```

-c	Compile only - don't link
-AS	Small memory model
-Gs	(s) remove stack probes
-W3	Generate all warnings
-Oils	(i) enable intrinsic functions (l) enable loop optimization (s) favor code size (t) favor execution time (d) optional flag for debugging

Notes:

- *Choosing between (s) and (t) can have a significant impact on your application. You may want to experiment before settling on one of them.*
- *Using the (d) flag disables optimization and is recommended when using debuggers like CodeView. Some of the sample programs use the (d) flag in their make files. Don't forget to remove it when building the production version of your code.*

-Zp	(p) pack structures on 1 byte boundaries (i) optional flag for CodeView debugging

Linker (LNK) files

The sample DosLink application, IcDosApp, was built with the following statements:

```
lnk @icdosapp.lnk
```

A typical LNK file (IcDosApp.lnk) contains:

```
/noi/map/line/co/stack:0x2000 icdosapp.obj
icdosapp.exe
icdosapp.map
doslink;
```

IcDosApp is the name of the application object file.

The /NOI option (noignorecase) causes the linker to distinguish between uppercase and lowercase letters.

The /CO option is optionally used to prepare for debugging with the Microsoft CodeView debugger. Don't forget to remove it from the production version of your code.

The /MAP and /LINE options are optional for debugging purposes.

DosLink is an object module library that resolves references to the INFOConnect functions.

IcDosApp - a Sample DosLink Application

Finally we are ready to look at the source for IcDosApp, a DosLink application that uses INFOConnect services. Most of the code fragments used as examples in the preceding discussion were taken from this program.

What does IcDosApp do?

IcDosApp is a simple, TTY-style communications program run from the DOS window on a machine running in Windows 386 enhanced mode. IcDosApp opens an INFOConnect path and sends keystrokes across the connection. The host is expected to echo the received character as in a full-duplex connection to a UNIX host. Microsoft console IO routines (_getch and _cputs) are used to access the keyboard and screen. You will have to modify the source if your compiler runtime library does not support these routines.

When invoked, IcDosApp expects a single command line parameter, the INFOConnect path to be opened.

```
c>icdosapp myunixpath
```

Source file descriptions

IcDosApp.c	C-language source

All source files needed to build this application are provided with the IDK in the SAMPLE directory. To build this application, do:

```
nmake -f makedos. PROGRAM=icdosapp
```

IcBDrive - a Sample DosLink TSR

BDrive is a data communications API used by many older DOS programs. BDrive was implemented as a DOS TSR program. Replacing the BDrive TSR with a TSR that maps the BDrive API to the DosLink API enables any program that uses BDrive has access to the INFOConnect transports.

Note: *The IcBDrive sample program only maps a subset of the BDrive API to INFOConnect.*

Source file descriptions

IcBDrive.c	C-language source
IcBDrive.msg	Message source file

All source files needed to build this application are provided with the IDK in the SAMPLE directory. To build this application, do:

```
nmake -f makedos. PROGRAM=icbdrive
```

To run a BDrive program, first install the IcBDrive TSR, then run the DOS program:

```
c>icbdrive
c>myprogram
```

Section 6
A Closer Look at the INFOConnect Architecture

This section is primarily directed at developers of INFOConnect libraries: both service libraries and external interface libraries. Applications developers may benefit from this information, but should not need it for development.

Manager Components

The following table lists each Manager Component, its dynamic link library (DLL) or executable (EXE) name, and the API it provides to applications, accessories and libraries:

Manager Component	DLL/EXE	API
INFOConnect Manager	INFOConn.exe	-
Communication Manager	IcMgr.dll	Library API and Manager API
Configuration Manager	IcMgrCfg.dll	Configuration Accessory API
Installation Manager	InstMgr.exe	-
Quick Configuration Manager	IcQCfg.exe	-
Database Manager	IcDb.dll	-
Utilities	IcUtil.dll and IcAbout.dll	-

Table 6–1. Manager Component, DLL/EXE, and API

Notes:

- *The session related interfaces of the ICS Accessory Application Programming Interface (AAPI) are provided by the Application Interface Library (IcAAPI16.dll).*
- *A CommMgr.dll stub provides support for applications built with the 1.0 and 2.0 INFOConnect Development Kits.*

The following diagram illustrates the manager components of INFOConnect Connectivity Services and the Application Programming Interfaces (APIs) they provide:

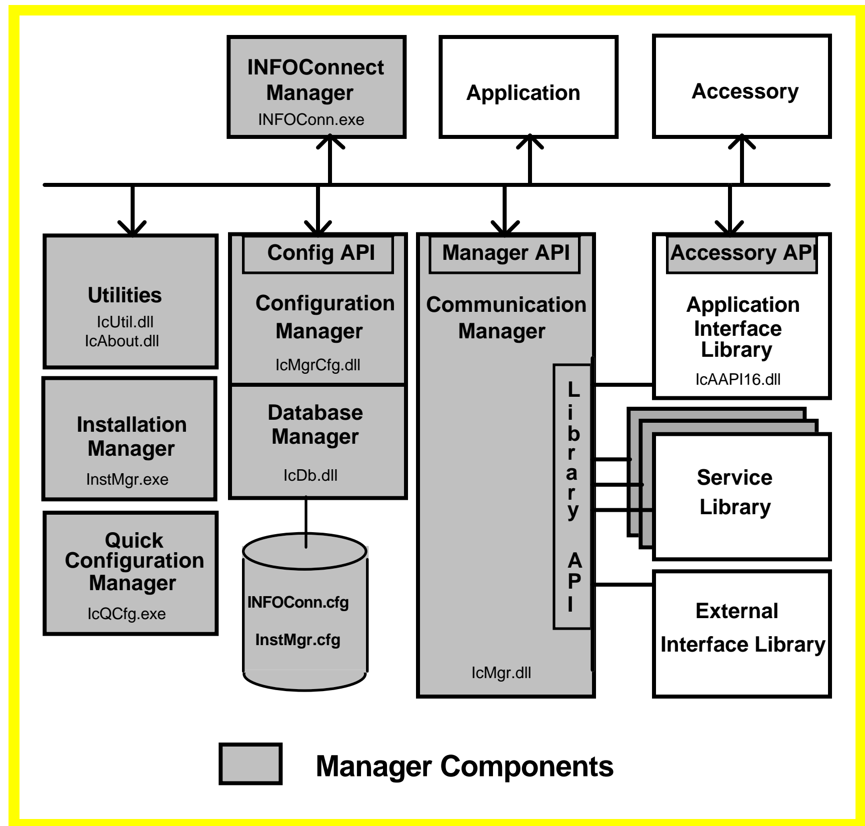


Figure 6–1. Manager Components and APIs

Structure of Service and External Interface Libraries

For brevity, service libraries may sometimes be referred to as *services* and external interface libraries as *interfaces or EILs.*

Each service and interface defines and exports a list of functions called by the manager.

The following table describes the functions that must be provided by an INFOConnect library:

IcLibUpdateConfig	Called to present a dialog box to user for configuration
IcLibVerifyConfig	Called to verify a configuration
IcLibPrintConfig	Called to format library-specific configuration information
IcLibInstall	Called once to perform initialization
IcLibTerminate	Called once to do cleanup before the library is unloaded
IcLibOpenChannel	Called to initialize a library channel
IcLibCloseChannel	Called to terminate a library channel
IcLibOpenSession	Called when application opens a session
IcLibCloseSession	Called when application closes a session
IcLibXmt	Called to transmit a buffer
IcLibRcv	Called to receive a buffer
IcLibLcl	Called to cancel pending transmits and receives
IcLibSetResult	Called to process status and error events
IcLibEvent	Called to preprocess events destined for the application
IcLibGetString	Called to supply error message text
IcLibIdentifySession	Called to retrieve a unique session identifier
IcLibGetSessionInfo	Called to provide the application with session information

ICS Control Flow

Processing an Open Session Request

The following diagram shows the flow of control through the INFOConnect system when an application calls function IcOpenSession.

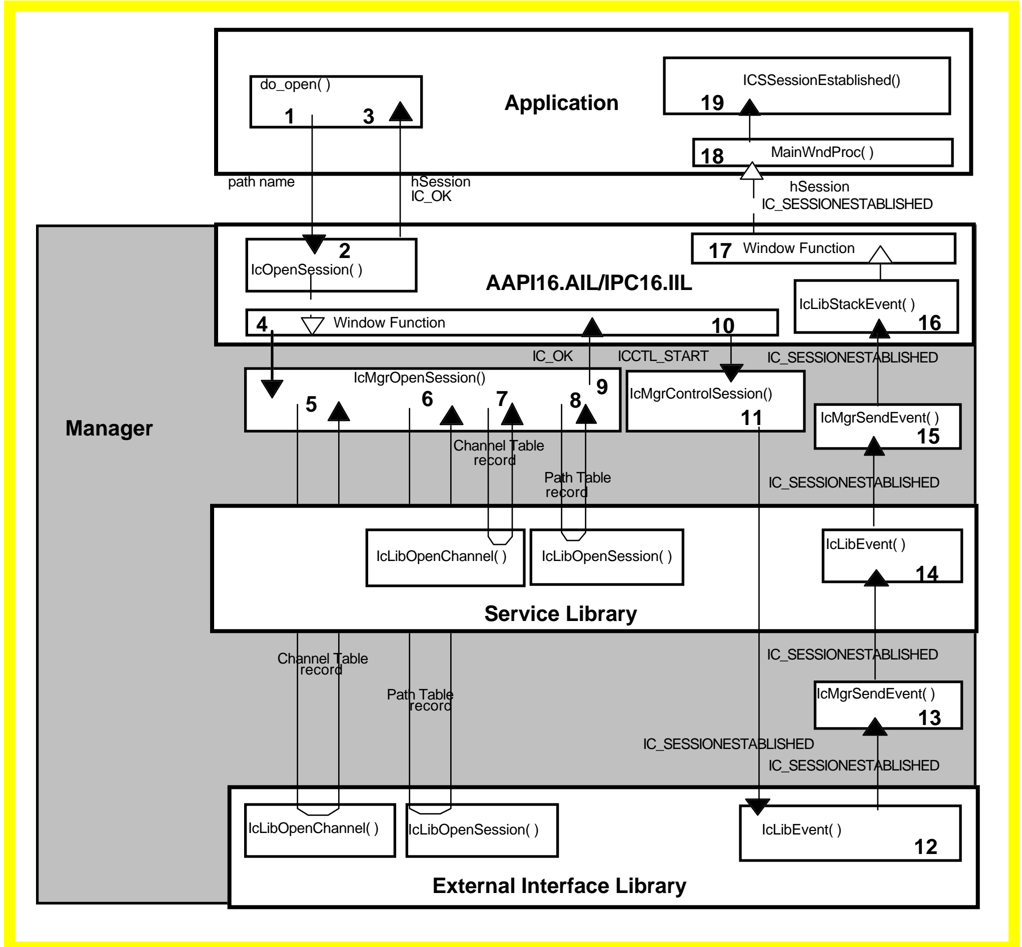


Figure 6–2. Processing During Session Establishment

The following steps relate to Figure 6-1:

1. The application has already initialized with INFOConnect and now calls IcOpenSession passing in the path name to be opened.

2. The application Interface Library/Interprocess Interface Library (AIL/IIL) does preliminary verification of the open request, looks up the path name, posts a message to an internal window on the INFOConnect message queue and returns a pending session handle, hSession, to the application. The dotted line indicates the posted message.

3. The application resumes processing until the session established event is returned.

4. The GUI system gives control to the AIL/IIL's Window function to process the message posted in step 2. This new execution sequence continues from step 4 through step 16. The Window function calls a Manager function called IcMgrOpenSession.

5. The Manager determines what service libraries and external interface are used by the path. Libraries are loaded, as necessary.

   If the external interface defines channels and this is the first session to be opened on the channel, IcLibOpenChannel is called. If the channel is already open because an earlier session used the same channel, IcLibOpenChannel is not called. If IcLibOpenChannel is called, your library is passed a channel table record and must initiate the process of opening the channel. Remember, though, that you cannot just wait for some external event to occur because Windows is a *cooperative* multitasking environment. If the "open" process cannot complete until some outside event has occurred, you must go ahead and return IC_OK from IcLibOpenChannel. Later, when the session established event is bubbled up to the IcLibEvent routine, you can refuse to bubble that event on up the stack until the awaited external event has occurred.

   If the external interface doesn't use channels, IcLibOpenChannel is still called one time during session establishment for the first session. Simply return IC_OK.

6. Next, the Manager calls IcLibOpenSession to open a specific session. If your library uses channels, the corresponding channel handle is passed in along with the appropriate path table record to IcLibOpenSession. After initializing session related data, return IC_OK.

The Manager immediately calls IcLibOpenSession a second time with the same parameters except for *Options,* which is set to IC_OPEN_VERIFY. This allows the Manager to determine if the path being opened can be opened again. If not, the Manager will prune the path from the list of available paths presented to the user in future session establishment dialogs. Returning anything but IC_VERIFY_OK will result in removing the path from the available list.

7. Service library processing occurs next. If any service libraries are configured in the path, the bottom most one (closest to the external interface) is called at IcLibOpenChannel. The conditions under which IcLibOpenChannel is called are the same as in step 5. service libraries that define or don't define channels are treated the same as the corresponding external interfaces described in step 5.

8. The service library's IcLibOpenSession routine is called to open a specific session. After initializing session related data, return IC_OK.

   As in step 6, the Manager immediately calls IcLibOpenSession a second time with the *Options* set to IC_OPEN_VERIFY. As discussed in step 6, your library must return IC_VERIFY_OK if the same path can continue to be opened by the user.

   If multiple service libraries are configured in the path, steps 7 and 8 are repeated for each library up the stack.

9. If all libraries in the stack have returned IC_VERIFY_OK on the secondary IcLibOpenSession calls, the path will remain in the list of available paths, and the user can choose it again in a future session establishment dialog. However, the Manager must have a unique suffix string to attach to the name of each session opened using this same path. Therefore, the Manager begins working its way down the stack of libraries (from the service library closest to the application to the external interface at the bottom) calling IcLibIdentifySession in each library. As soon as one of the libraries returns a non-NULL string, this probing activity stops and the lower libraries are not called. The Manager returns an IC_OK to the AIL/IIL.

10. The AIL/IIL calls IcMgrControlSession with ICCTL_START to establish the session.

11. The Manager begins the process of notifying all the libraries in the stack about the newly opened session. A session-established event is bubbled up through the architecture starting with the external interface.

12. At this point, the EIL has complete control of the session establishment process. The next library in the stack won't get the session established event until the EIL passes it on. Generally, the EIL does some bookkeeping and immediately bubbles the event up the library stack by calling the Manager at IcMgrSendEvent. However, if the EIL is waiting for some external event to occur, it can temporarily block the session establishment process by simply not calling IcMgrSendEvent for now. Later, perhaps during the EIL's timer processing, the EIL can resume the session establishment process by calling IcMgrSendEvent.

Note: *Remember that for cooperative multitasking platforms like Windows, you must "wait" via some kind of timer or callback mechanism. You can't just take over control of the CPU.*

13. The Manager determines if there are more service libraries to call and passes the session established event up the stack. If there were no service libraries configured in the path, the Manager would post the event to the AIL/IIL at this point.
14. Each service library in the stack gets called at IcLibEvent and calls IcMgrSendEvent in turn. As noted in step 12, each library can temporarily block the process and wait to call IcMgrSendEvent at some later time.
15. Finally, the Manager runs out of service libraries to call, so then calls the AIL/IIL's IcLibStackEvent function with the session established event (Ic_SessionEstablished).
16. The AIL/IIL posts a session established event to an internal window on the application message queue . The dotted line indicates the posted message.
17. The GUI system gives control to the AIL/IIL to process the message posted in step 16.
18. The AIL/IIL posts an IC_SESSIONESTABLISHED message to the application's message queue. The application now has a valid session handle (hSession).
19. The application can allocate buffers and begin sending data across the session.

Processing a Transmit (or Receive) Request

The following diagram shows the flow of control through the INFOConnect architecture when an application calls function IcXmt to transmit some data across a session. The same flow control also applies to receive requests (IcRcv).

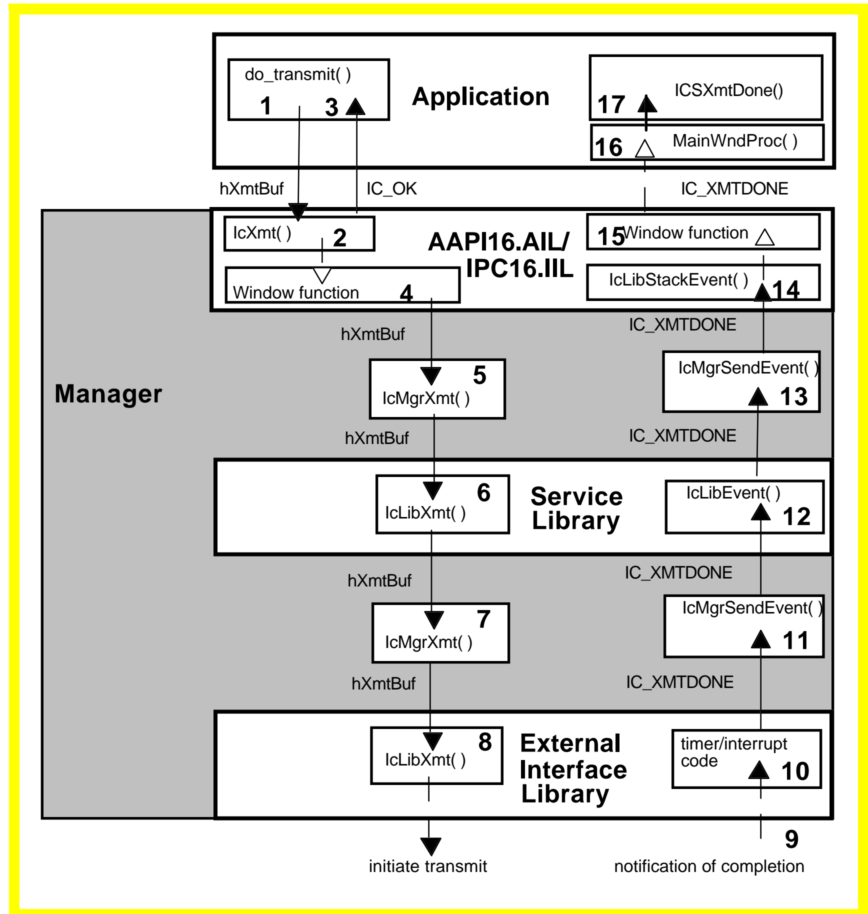


Figure 6–3. Processing During Transmit

The following steps relate to Figure 6-2:

1. The application fills a buffer and passes its handle to IcXmt.
2. The Application Interface Library/Interprocess Interface Library (AIL/IIL) does preliminary verification of the transmit request. If it looks like a valid request, the AIL/IIL posts a message to an internal window on the INFOConnect message queue and returns IC_OK to the application. The dotted line indicates the posted message.
3. The application resumes processing until the transmit finishes. The data buffer being transmitted, hXmtBuf, is still unavailable to the application until the transmit done event is returned.
4. The GUI system gives control to the AIL/IIL's Window function to process the message posted in step 2. The Window function calls a Manager function called IcMgrXmt.
5. IcMgrXmt determines the first service library in the stack and calls service library function IcLibXmt passing the application's buffer handle. If there are no service libraries in the path configuration, IcMgrXmt calls the external interface library and skips to step 8.
6. The service library can change the buffer, if desired, and then pass it on to the Manager using IcMgrXmt. The service library can even allocate and pass a new buffer to the Manager.
7. The Manager determines the next library in the stack. It could be another service library, but eventually the buffer is passed to the external interface Library.
8. The EIL initiates a transmit across the lower communications layers. When each library returns in turn, the execution sequence that began at step 4 is complete.
9. The lower communications layer completes the transmit request and notifies the external interface. Exactly how this is done will depend on the implementation of the particular interface. A timer routine registered earlier by the interface during IcLibOpenChannel is one possibility. The timer routine could periodically poll the status of all datacomm requests. An interrupt routine is another possibility.
10. The EIL determines that the transmit is done and sends a transmit done event (IC_XMTDONE) to the manager at IcMgrSendEvent.

11. The Manager determines the next service library in the stack and passes the transmit done event to the library, by calling the service library function IcLibEvent.
12. The service library processes the transmit done event and passes it on up the stack to the Manager by calling IcMgrSendEvent.
13. When the Manager exhausts the stack of service libraries, it passes the transmit done event to the AIL/IIL's IcLibStackEvent function.
14. The AIL/IIL posts a transmit done message (IC_XMTDONE) to an internal window on the application message queue. The dotted line indicates the posted event. When each library returns in turn, the execution sequence that started at step 9 is complete.
15. As in step 4, the GUI system gives control to the AIL/IIL to process the posted message.
16. The AIL/IIL posts an IC_XMTDONE message to the application's message queue.
17. The application resumes processing the transmit done message.

Processing Status and Error Events

Status and error events go in one of two directions depending on who initiates them, the application or the external interface library.

The processing that occurs when the application requests to go into the local state (IcLcl) is handled similarly to the processing done for status events initiated by the application.

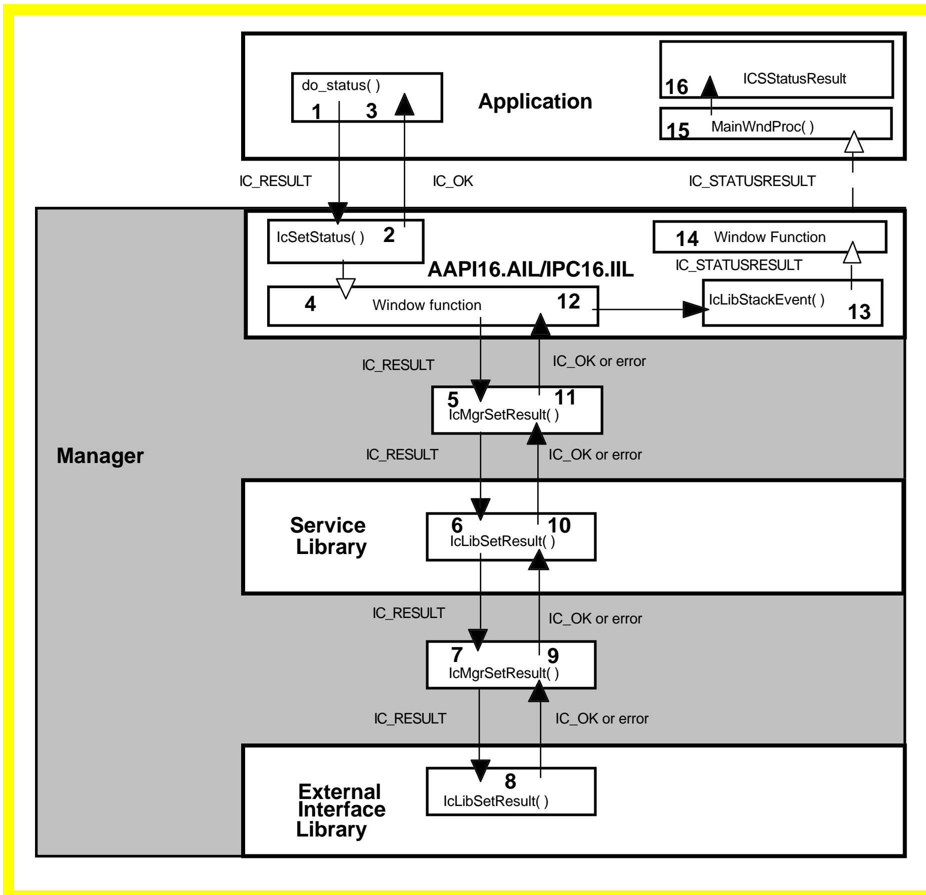


Figure 6–4. Status Initiated by the Application

The following steps relate to Figure 6-3:

1. The application calls function Ic_SetStatus (or Ic_SetError) and passes in an IC_RESULT.
2. The AIL/IIL does preliminary verification of the call. If it looks OK, the AIL/IIL posts a message to an internal window on the INFOConnect message queue and returns IC_OK to the application.
3. The application resumes processing.
4. The GUI system gives control to the AIL/IIL's Window function to process the message posted in step 2.
5. The Window function calls a manager routine named IcMgrSetResult with the IC_RESULT.
6. IcMgrSetResult determines the first service library in the stack and calls service library function IcLibSetResult. IcMgrSetResult passes the application's IC_RESULT parameter and indicates whether this is a status or error. If there are no service libraries in the path configuration, IcMgrSetResult calls the external interface library and skips to step 8.
7. The service library can recognize or ignore the status, but it must pass it on down the stack by calling IcMgrSetResult.
8. The Manager determines the next library in the stack. It could be another service library, but eventually the IC_RESULT is passed to the external interface Library.

   The interface can recognize or ignore the status. If the interface recognizes the status, it should return an IC_OK or error to the caller.

   If the interface doesn't recognize the status, it should continue to pass the IC_RESULT down to IcMgrSetResult for processing by the Manager. Then, whatever the Manager returns should be bubbled back up the chain.
9. The Manager determines the next service library in the stack and returns to it.
10. Each service library returns to its original caller.
11. When the Manager determines that all libraries in the stack have returned, it returns the IC_RESULT to the window function.
12. The window function calls IcLibStackEvent with the associated result.

13. The AIL/IIL posts a status result message to an internal window on the application message queue. The dotted line indicates the posted message. This completes the execution sequence that started at step 4.
14. As in step 4, the GUI system gives control to the AIL/IIL to process the posted message.
15. The AIL/IIL posts an IC_STATUSRESULT message to the application's message queue.
16. The application resumes processing the status result message.

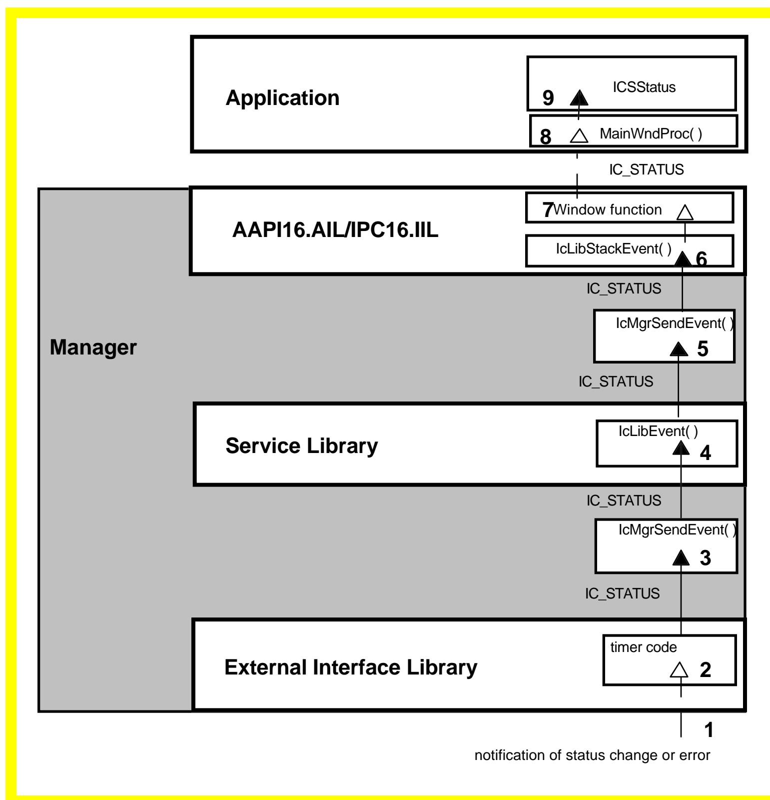


Figure 6–5. Status Initiated by a Library

The following steps relate to Figure 6-4:

1. The lower communications layer notifies the external interface of some status change or error. Exactly how this is done will depend on the implementation of the particular interface. The timer code shown in the diagram is just one possible implementation. This routine might have been registered earlier by the interface (perhaps during IcLibInstall) and it periodically polls the status of all datacomm requests.
2. The external interface determines that a status change or error has occurred. It builds an IC_RESULT with the appropriate context, type and value and calls the manager at function IcMgrSendEvent.
3. The Manager bubbles the IC_STATUS event up through the stack of service libraries starting with the bottom most one first.
4. Each service library gets an opportunity to see the IC_RESULT headed for the application.
5. The Manager eventually runs out of service libraries and calls the AIL/IIL.
6. The AIL/IIL posts a status message to an internal window on the application message queue. The dotted line indicates the posted event. This completes the sequence that started in step 1.
7. The GUI system gives control to the AIL/IIL to process the message posted in step 6.
8. The AIL/IIL posts an IC_STATUS or IC_ERROR message to the application's message queue.
9. The application resumes processing the status or error message.

Section 7
Writing INFOConnect Libraries for Windows 3.x

This section leads you through all phases of service and external interface library development: designing, coding, compiling and linking. Several sample libraries are presented at the end of the section. All source files necessary to build the samples are provided with the development kit.

Skeleton files are also provided with the development kit (Service.* and Intrface.*) as a starting point for your own library development.

Basically, writing an INFOConnect library consists of creating a DLL that exports the required list of functions introduced in Section 6, "A Closer Look of the INFOConnect Architecture". Before explaining each of the required functions, there are some broader design issues that must be explained.

Design Issues

This section covers issues and questions to be considered before you begin to write your library.

Choosing Between Accessories, Services and Interfaces

It is not always obvious whether a particular solution is best implemented as an accessory or service or interface, or some combination. In the C language, the choice of a for statement, do statement, or while statement is often a matter of style or convenience; any of them can be made to work, but one is likely to be more appropriate. The same is often true of the choice between ICS components.

Don't favor writing applications and overlook the use of service and external interface libraries. Before assuming that you need to write an application, think about your solution first as a service library, then as an external interface library, and finally as an accessory or application. You may be able to implement at least part of your solution as a library. Libraries are generally smaller and more structured than applications and are more easily reused by other INFOConnect applications.

Tasks suited for service libraries are usually the easiest to identify. Do you need to filter the data stream in some way? For example, suppose you need to convert incoming EBCDIC data to ASCII. This could easily be accomplished within the application itself, but if implemented as a service, the code can easily be reused by other applications. Tasks typically assigned to the Session and Presentation layers of the OSI model are very appropriate for service libraries. This includes things like dialog management, data compression, data representation, and data encryption. Furthermore, don't lower your sights on what a service library can accomplish. They can be quite sophisticated; even doing transmits and receives "behind the applications' back."

Tasks suited for external interface libraries are not limited to the obvious ones like support for a new data communications transport. If you need to interface to a Windows DLL from your application, consider writing an external interface library for that piece of the implementation. Once again, your code is more likely to be reusable by other ICS applications.

INFOConnect libraries do not generally "talk" to the user except for the configuration dialog. If your task requires interaction with a user it might be better to split the task, implementing the user interaction portion as an application and other the parts of the task as a library (or multiple libraries).

Session Attributes

The IC_SINFO data structure contains session attributes that describe to the application the characteristics and limitations of a connection. Less ambitious applications may choose to only work across some subset of "preferred" connection-types that minimizes the effort required by the application to get data across the session.

Session attributes are primarily in the domain of external interface libraries. Interfaces initialize the IC_SINFO structure and services generally leave them alone. However, it is well within the capabilities of a service to alter one or more session attributes to enhance the basic characteristics of the connection.

The "preferred" values for session attributes from the application's perspective follow. *Transparent* and *block_mode* probably have the greatest impact on the application.

Max_size

Generally, the bigger the block that can be transported across the connection, the better.

Transparent

Transparent indicates whether all binary data streams can be sent across the connection. *Transparent=TRUE* connections are preferable and more convenient to the application. Non-transparent connections require the application to use some type of data encoding mechanism or be content to only use the displayable ASCII character set.

Block_mode

Block_mode indicates whether data is sent and received as messages or as a stream of characters. *Block_mode=TRUE* connections are preferable. Otherwise, the application must scan the receive buffers and do its own blocking/unblocking of data into logical messages.

Reliable

Reliable indicates whether undelivered messages are signaled to the application. Naturally, *reliable=TRUE* is preferable.

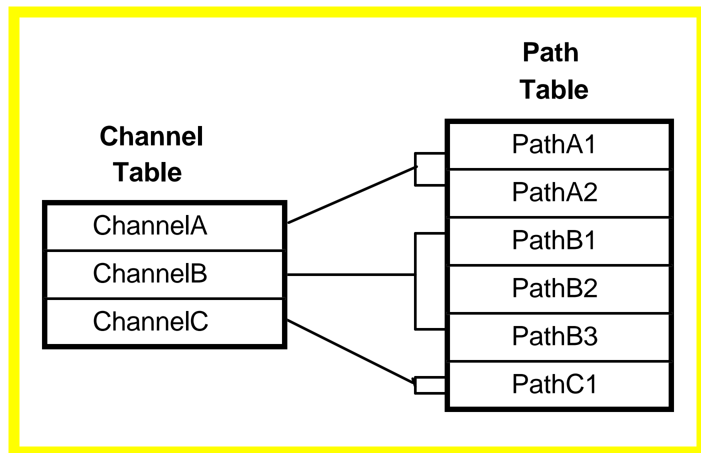
Focus_notify

Focus_notify indicates whether or not the application should call the Set Status procedure with IC_REACTIVATE_ON or IC_REACTIVATE_OFF each time it gains or loses focus. *Focus_notify=FALSE* is preferred, however, it is easy to support focus_notify and most applications should support it.

Configuration Management

Library configuration is based on tables. Configuration information for a library is organized into tables of rows and columns (records and fields). Tables can reference each other through link fields. The goal is to eliminate data redundancy. Rather than duplicating information in two places, information is stored in one place and links are established between tables.

The INFOConnect configuration architecture centers around two tables: the path and channel tables. Multiple path records in the path table are linked to one channel record in the channel table. Path and channel records are passed to the library from the Manager during the session establishment process.



Don't confuse the above diagram with multiplexing. The diagram shows the static relationship between paths and channels. Non-multiplexing libraries can often use channels and paths in their organization of configuration information.

For external interface libraries that define both channels and paths, the INFOConnect Manager provides a user interface in order to link a path to a channel. Service libraries typically do not require channel tables. A service library that defines a channel table must also provide an interface (a control on the path dialog) which links a path to a channel.

A library can use several combinations of tables. The most common combinations are: no tables at all, only a path table, or both a path and channel table. Libraries can also define additional tables, such as custom tables or invisible tables, as needed. All tables, except invisible tables, can be accessed by using the INFOConnect Library Configuration window. Your library manages the user interface for invisible tables with minimal help from the Manager.

Table definitions and default values are contained in your library's resource (.RC) file, but the actual table data resides in the INFOConnect configuration file, INFOConn.cfg. Your library provides callback functions to help the Manager process your tables. IcLibUpdateConfig, IcLibVerifyConfig and IcLibPrintConfig are callback functions that process tables at configuration time. IcLibUpdateConfig typically provides dialog boxes for each table defined by your library. IcLibOpenChannel and IcLibOpenSession are callback functions that process tables at run time. During session establishment, the Manager passes the appropriate channel and path table records to IcLibOpenChannel and IcLibOpenSession. See the control flow diagram in Section 6, "A Closer Look at the INFOConnect Architecture," for more information about session establishment processing.

As mentioned earlier, some libraries won't need any tables. When a library does the same processing on all data that passes through it (for example, there is no difference in processing between one session and another), there is no need for configuration tables that differentiate paths from each other. Other libraries will only need a path table. This is very common. The Stack library in the IDK samples only uses a path table. Finally, when many paths share some common information, that information should be moved into another table such as a channel or custom table; especially if this information is shared at runtime. The Reflect sample library uses both a path and channel table. The UTS and Poll/Select external interface libraries also use both tables. Channel tables are often used by EILs to store shared, hardware-related information like COM port number, interrupt number, and so forth.

The work required to implement library configuration is broken into the following steps:

- **Table Design**
- **Table Description**
- **Table Processing**

Table Design

The following is a walk through of the table design for a hypothetical TTY External Interface Library that drives a modem attached to one of the PC's COM ports. The path table contains most of the information needed for a connection: baud rate, parity, phone number, and so forth. Rather than "hard-code" a COM port and IRQ into each path, though, we will use a channel table and define the COM port there.

Here is the layout of our two tables along with sample records for each.

Path table

PathID	ChannelID	Baud	Data	Stop	Parity	Phone
CompuSrv	ModemA	9600	7	1	E	555-1212
UnixBox	ModemA	19200	8	1	N	555-1234

Channel table

ChannelID	COM port	IRQ
ModemA	2	3

If the modem is moved to a different COM port, only the channel table record for ModemA needs to be updated. The two path table records remain unchanged.

Dialog boxes for path and channel tables

To keep one foot in the real world, here are the associated dialog boxes to be completed by the user during path and channel configuration. In this example, all fields in the path and channel tables are included in the dialog box. This is not required. There may very well be fields in a table that the user never sees.

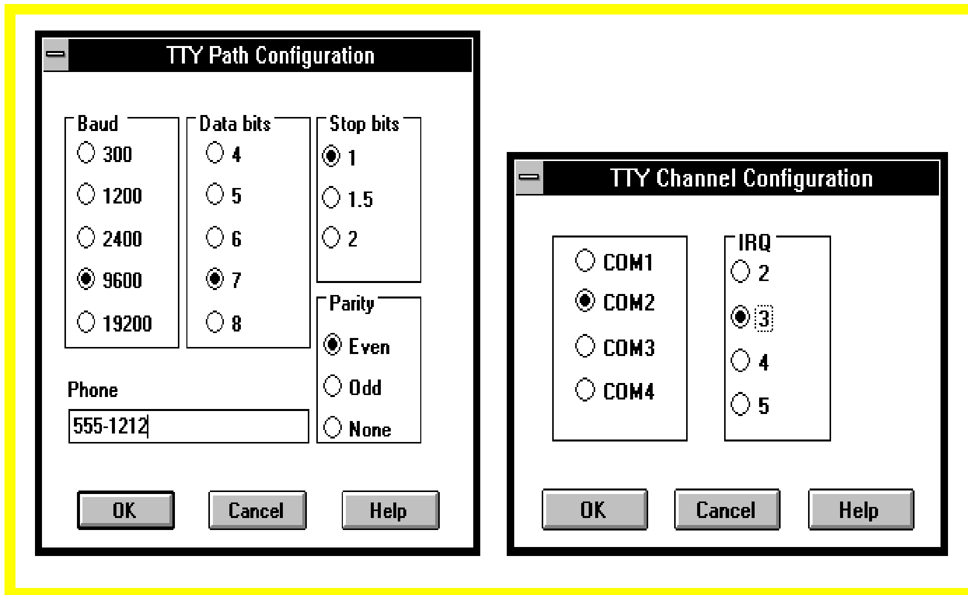


Table Description

Now we must define each field in each table before encoding the table definitions into the .RC library resource file. Each field is described by a key type, field type, starting location and length. Here are some of the available types listed in the IcDict.h header file:

```
/* Key types (Field Flags) */
#define IC_FF_NO_KEY      0
#define IC_FF_PRIMARY_KEY 1
#define IC_FF_LINK_KEY    2
#define IC_FF_ALTERNATE_KEY 3

/* Field types */
#define IC_FT_BINARY    0
#define IC_FT_INT       1
#define IC_FT_BOOL      2
#define IC_FT_CHAR      3
#define IC_FT_UNSIGNED  4
#define IC_FT_STRUCTURE 5
#define IC_FT_STRING    6
#define IC_FT_STRINGI   7
```

Path table field-descriptions

The first two fields in the path table, *PathID* and *ChannelID*, exist by default in all path tables. The remaining fields, starting with *Baud* in this example, are different for each specific library. No start and length information is needed for PathID and ChannelID because the Manager provides that for you.

field name	**key type** (use IC_FF_ data types)	**field type** (use IC_FT_ data types)	**start bit**	**length in bits**
PathID	IC_FF_PRIMARY_KEY	string - IC_FT_STRINGI	***	***
ChannelID	IC_FF_LINK_KEY	string - IC_FT_STRINGI	***	***
Baud	IC_FF_NO_KEY	integer - IC_FT_UNSIGNED	-1	16
Databit	IC_FF_NO_KEY	integer - IC_FT_UNSIGNED	-1	16
Stopbit	IC_FF_NO_KEY	integer - IC_FT_UNSIGNED	-1	16
Parity	IC_FF_NO_KEY	integer - IC_FT_UNSIGNED	-1	16
Phone	IC_FF_NO_KEY	string - IC_FT_STRINGI	-1	160 (20 bytes)

Note: *A* -1 *is used for the start-bit to indicate that the field immediately follows the previous field.*

Channel table field-descriptions

The first field in the channel table, *ChannelID*, exists by default in all channel tables. The remaining fields, starting with *COM port* in this example, are specific to each library. No start and length information is needed for ChannelID because the Manager provides that for you.

field name	**key type** (use IC_FF_ data types)	**data type** (use IC_FT_ data types)	**start bit**	**length in bits**
ChannelID	IC_FF_PRIMARY_KEY	string - IC_FT_STRINGI	***	***
COM port	IC_FF_NO_KEY	integer - IC_FT_UNSIGNED	-1	16
IRQ	IC_FF_NO_KEY	integer - IC_FT_UNSIGNED	-1	16 (2 bytes)

*** These fields are always present, but are defined and managed by the Manager. Do not actually define them in your table description.

Table descriptions in .RC format

The table descriptions can now be encoded into an INFOConnect-specific resource to be included in the library's .RC resource file. This is just one section of several that are required in the .RC file. See page 7 - 74 for a complete description of the resource file and how all the sections interrelate. Notice how the field names are also defined in the STRINGTABLE.

```
#define IC_PATHTABLE_BASENO   1000
#define IC_CHANNELTABLE_BASENO 1001

/* Field names */

#define BAUD  100
#define DATABIT 101
#define STOPBIT 102
#define PARITY 103
#define PHONE  104
#define COMPORT 105
#define IRQ   106

IC_PATHTABLE_BASENO IC_DICTIONARY_RCTYPE
BEGIN
 BAUD,  IC_FF_NO_KEY, IC_FT_UNSIGNED,  -1, 16,
 DATABIT, IC_FF_NO_KEY, IC_FT_UNSIGNED,  -1, 16,
 STOPBIT, IC_FF_NO_KEY, IC_FT_UNSIGNED,  -1, 16,
 PARITY, IC_FF_NO_KEY, IC_FT_UNSIGNED,  -1, 16,
 PHONE, IC_FF_NO_KEY, IC_FT_STRING,   -1, 160,
 0
END

IC_CHANNELTABLE_BASENO IC_DICTIONARY_RCTYPE
BEGIN
 COMPORT, IC_FF_NO_KEY, IC_FT_UNSIGNED,-1, 16,
 IRQ,   IC_FF_NO_KEY, IC_FT_UNSIGNED,-1, 16
 0
END

STRINGTABLE
BEGIN
 BAUD,  "Baud"
 DATABIT, "Data bits"
 STOPBIT, "Stop bits"
 PARITY, "Parity"
 PHONE,  "Phone number"
 COMPORT, "COM Port"
 IRQ,   "IRQ"
END
```

BAUD, DATABIT, STOPBIT, PARITY, PHONE, COMPORT and IRQ are numeric IDs that correspond to strings in the STRINGTABLE defining the field names in the configuration tables.

Modifying library table fields

At times library updates or new features may require modifications to the library table information. During the installation process, configuration data is moved from the old records to new records by matching field numbers between the old and the new dictionaries. If the new field is longer, it is padded on the right with zeros. If it is shorter, the data is truncated on the right. New fields receive default data from the default data resource.

Note: *The serial number of the table should be incremented to identify that there have been modifications to the table.*

The following rules should be followed when making changes to the fields that existed in prior versions of the library:

- When information needs to be added, the new fields must be added to the end of the table.
- When information is no longer required, the field should not be deleted. Rather than delete the field, *obsolete* the field by setting its length to 0 in the table description in the .RC file. Also, the field definition in the .HIC file should be commented out.
- When information needs to be modified, change the key type, data type, start bit or length in the table description in the .RC file. However, never change an IC_FF_NO_KEY to an IC_FF_ALTERNATE_KEY when other alternate keys follow in the table definition. This would renumber the alternate keys. In this case, obsolete the IC_FF_NO_KEY and add the new IC_FF_ALTERNATE_KEY to the end of the table.

Note: *Special conditions in upgrading configuration records of previously released levels, such as reformatting data, can be automatically handled during the installation process by providing upgrade code in IcLibUpdateConfig and IcLibVerifyConfig. Reflect has been coded to demonstrate the library upgrade feature.*

Table Processing

There are five callback functions that manipulate tables. IcLibUpdateConfig, IcLibVerifyConfig and IcLibPrintConfig build and update tables during configuration. IcLibOpenChannel and IcLibOpenSession use them at run time during session establishment.

IcLibUpdateConfig

This is the primary callback used during channel and path configuration. The Manager passes your library a record (or row) from either the path, channel or custom table and indicates the type of action being performed: add, modify, examine, and so forth. New records (during add operations) are initialized with values from the default record defined in your library's resource file. The library is expected to present the appropriate dialog box to the user and return an updated table record to the Manager. There is often a one to one correlation between tables and dialog boxes, but that is not required.

IcLibVerifyConfig

This function is called by the Manager to perform semantic checking on a record from one of your library tables. No dialog box or user interaction is to be done. Any time a library is configured, this routine is called to accept the data before it is stored, for example during Quick Configuration of a package.

IcLibPrintConfig

This function is expected to format and return a displayable text string summarizing a record from one of your library tables. The description text displayed in the INFOConnect Library Configuration is obtained from this function.

IcLibOpenChannel

This function is called during session activation when a path is being opened and the path is associated with an unopened channel. A channel table record is passed as input along with a channel handle to identify the channel in future references. Your library does what needs to be done to open the channel and saves the channel handle and any relevant information from the channel table record.

IcLibOpenSession

This function is called during session activation. IcLibOpenChannel has already been called, if necessary. A path table record is passed as input along with the handle of the channel associated with this path. Your library does what needs to be done to open the session and saves the session handle and any relevant information from the path table record.

Session and Channel Runtime-record Layout

The next data structures you should design are channel and session runtime-records describing each active channel and session. These structures are basically containers for the path and channel table records passed to your library during session establishment plus any additional information required to manage the channel or session while it is open. Carefully thought-out record structures make designing the different required functions very straight forward.

Here are possible channel and session runtime-records for the hypothetical TTY EIL we have been designing. Typedef structs channel_config and path_config define all the information from the channel and path tables.

```
typedef struct aChannelConfig { // save info from channel table
  int COMport;
  int IRQ;
} channel_config;

typedef struct aPathConfig { // save info from path table
  int baud;
  int databit;
  int stopbit;
  int parity;
  char phone[20];
} path_config;

typedef struct aChannelRec {
  struct aChannelRec *pNextChannelRec;
  HIC_CHANNEL hChannel;
  HIC_SESSION   hSession; // first child session for this channel
  channel_config cr;
} CHANNELREC;

typedef struct aSessionRec {
  struct aSessionRec *pNextSessionRec;
  HIC_SESSION hSession;
  HIC_CHANNEL hChannel; // parent channel for this session
  path_config cr;
} SESSIONREC;
```

Error Management

Almost all the required IcLib functions return an error value using typedef IC_RESULT. Many of the INFOConnect events also incorporate an error value. These are the two mechanisms available to your library for reporting errors. Except for the configuration dialogs, you should never issue dialog or message boxes from your library. This allows the application to have complete control over error presentation to the user.

INFOConnect provides a collection of standard errors that should satisfy most of your library's requirements. Standard errors are defined in IcError.h and are further described in the *IDK Programming Reference Manual*. You should make every effort to only use standard errors, but there is also a mechanism available for defining library-specific errors using the .HIC header file and the IcLibGetString function.

Both standard and library-specific errors are classified into four types. Library-specific errors can be defined so as to be further subgrouped within each of these types as necessary. Here are the four types of errors and the recommended action to be taken by the application:

IC_ERROR_INFO	The requested function was completed. The application should log this error if it has a log file. Don't bother displaying a message to the user. The default error procedure, available to the application, will only display these errors if INFOConnect is running in debug mode.
IC_ERROR_WARNING	The requested function was completed, but something unusual or noteworthy happened. The application can choose to log or display this error. The default error procedure, available to the application, will display these errors.
IC_ERROR_SEVERE	The requested function did not complete successfully. The application should display this error to the user. The default error procedure, available to the application, will display these errors.
IC_ERROR_TERMINATE	The application should display this error and then close the session. The default error procedure, available to the application, will display these errors.

Generally, errors are not transmitted across the connection; the error is only seen by the components associated with the current session: the application, Manager, service libraries and external interface.

Note: *For errors of type IC_ERROR_INFO and IC_ERROR_WARNING, applications can assume that the requested function was completed.*

Status Management

Status messages are used to communicate between the different components of the current session: the application, Manager, service libraries and external interface. Generally, statuses are not transmitted across the connection; the status is only seen by the components associated with the current session.

Statuses can travel in one of two directions: from the application down to the external interface or from the external interface up to the application. Statuses from the application are passed to the IcLibSetResult function and statuses headed to the application are "bubbled up" through the IcLibEvent function.

INFOConnect provides a collection of standard statuses that should satisfy most of your library's requirements. Standard statuses are defined in IcStatus.h and further described in the *IDK Programming Reference Manual*. You should make every effort to only use standard statuses, but there is also a mechanism available for defining library-specific statuses using the .HIC header file. Unlike errors, statuses have no associated text and therefore are not handled by the IcLibGetString function.

Here are the standard status types defined in IcStatus.h that external interfaces should support:

- all IC_STATUS_LINESTATE statuses
- all IC_STATUS_CONNECT statuses except IC_CONNECT_EOF
- IC_CONTROL_RCVAVAIL
- IC_STATUS_TRANS

Section 3, "Writing INFOConnect/Windows Applications," provides a complete discussion on these statuses.

Using IC_STATUS_BUFFER Extended Status

When an application needs to exchange more information with an ICS library than IC_RESULT_VALUE can store, it can send a buffer of information with the IC_STATUS_BUFFER extended status. To accomplish this, a HIC_STATUSBUF buffer handle is assigned to the IC_RESULT_VALUE member of the IC_RESULT structure.

In the following discussion, these members of the IC_STATUSBUF structure will be referenced:

- **icstatus**, the actual status that is associated with the status information as defined by the library
- **icerror**, the IC_RESULT of the status request
- **uBufSize**, the actual size of the IC_STATUSBUF buffer
- **uDataSize**, the size of the valid data
- **data**, the data buffer. This buffer should not contain pointers, but may contain offsets within the structure.

Extended statuses can be exchanged in two ways: synchronously and asynchronously. The following steps are involved in exchanging a synchronous extended status:

1. The application allocates a buffer for the IC_STATUSBUF data structure. It then updates icstatus, icerror, uBufSize. uDataSize and data (data is optional, the status data buffer may be used for the library to supply the application with status data). Be sure to store the handle for IC_STATUSBUF for use in step 5.
2. The application creates an IC_RESULT by calling IC_MAKE_RESULT with the following three parameters: IC_RESULT_CONTEXT_STD, IC_STATUS_BUFFER, and the handle to the IC_STATUSBUF data structure. The extended status is sent to the library by calling IcSetStatus.
3. The library receives the status in IcLibSetResult and calls IC_GET_RESULT_VALUE to obtain the handle of the IC_STATUSBUF. The library performs whatever tasks are required depending on the icstatus field. It may just read the information in the data buffer and take appropriate actions. If the library writes information to the data buffer, it must also update uDataSize.

4. The library sets the icerror field to IC_OK or an appropriate error and also returns icerror from IcLibSetResult.
5. The application receives an IC_StatusResult message. It now accesses the information stored in IC_STATUSBUF by using the handle stored in step 1.

Note: *The application must remember to release the memory allocated for IC_STATUSBUF.*

In the asynchronous case:

1. The application allocates a buffer for the IC_STATUSBUF data structure. It then updates icstatus, icerror, uBufSize. uDataSize and data. Data is optional, the data buffer may be used for the library to supply the application with status data.
2. The application creates an IC_RESULT by calling IC_MAKE_RESULT with the following three parameters: IC_RESULT_CONTEXT_STD, IC_STATUS_BUFFER, and the handle to the IC_STATUSBUF data structure. The extended status is sent to the library by calling IcSetStatus with this IC_RESULT.
3. The library receives the status in IcLibSetResult and calls IC_GET_RESULT_VALUE to obtain the handle to IC_STATUSBUF. The library sets icerror to IC_INCOMPLETE and also returns IC_INCOMPLETE.
4. The application receives an IC_StatusResult message. Since the result is IC_INCOMPLETE, the application waits until it receives an IC_STATUS message.
5. The library performs whatever tasks are required depending on the icstatus field. It may just read the information in the data buffer and take appropriate actions. If the library writes information to the data buffer, it must also update uDataSize.
6. The library sets the icerror field to IC_COMPLETE (or an appropriate error) and calls IcMgrSendEvent to send the IC_STATUS_BUFFER extended status back to the application.
7. The application receives an IC_Status message. It now accesses the information stored in IC_STATUSBUF by calling IC_GET_RESULT_VALUE to obtain the handle to IC_STATUSBUF.

Note: *The application must remember to release the memory allocated for IC_STATUSBUF.*

Version Control

There are four places to identify various version control information for your library:

- The package section of an installation script file (.INF), which is used to install the library, contains INFOConnect Connectivity Services version numbers. This represents the version of the package and the version range of ICS that the package was built for.
- The version information section of the library's resource file (.RC) contains library version information which is used by the INFOConnect Installation Manager. This is the version of each file.
- The INFOConnect RCDATA section of the library's resource file (.RC) contains INFOConnect Connectivity Services version numbers. This is the version range of ICS that the library was built for.
- Every library table has a serial number which is incremented to identify a newer version of the table. This is the version of the table.

INFOConnect Connectivity Services version numbers

INFOConnect Connectivity Services version numbers contains four fields: major version, minor version, EMU level, and build revision. IC_VERSION_FILE and IC_VERSION_PRODUCT are defined in IcDef.h as follows:

```
#define IC_VERSION_FILE    IC_MAJOR_VERSION,IC_MINOR_VERSION,IC_EMU_LEVEL
           IC_BUILD_REVISION
#define IC_VERSION_PRODUCT IC_MAJOR_VERSION,IC_MINOR_VERSION,IC_EMU_LEVEL,
           IC_BUILD_REvISION
```

The File Version and Product Version can be viewed as follows:

```
/* File/Product Version Information            */
/* image format 3.00 0 (000)                   */
/*     *                               Major version */
/*      **                             Minor version */
/*        *                            EMU Level     */
/*          ***                     Build revision */
```

Using major release 3.0 as an example: IC_MAJOR_VERSION is 3 and IC_MINOR_VERSION, IC_EMU_LEVEL, and IC_BUILD_REVISION are 0.

Another way to look at ICS version numbers is to group the major version and minor version fields as "Version" information and the EMU level and build revision fields as "Revision" information. The IC_VER_INFO data structure allows programmers to access ICS version information as four BYTE fields or two WORD fields:

```
typedef LONG IC_VER;
typedef union {
  IC_VER IcVer;
  struct {
    WORD Rev;
    WORD Ver;
  } w;
  struct {
    BYTE Revision;
    BYTE EmuLevel;
    BYTE MinorVersion;
    BYTE MajorVersion;
  } b;
} IC_VER_INFO;
```

To continue the 3.0 example, Ver consists of major and minor version and is equivalent to IC_VERSION_3_0 (defined as 0x0300). Rev, consists of EMU level and revision and is equivalent to IC_REVISION_3_0 (defined as 0x0000) IC_VERSION_... and IC_REVISION_... defines are also found in IcDef.h.

Installation script file version information

There are three parameters containing version information in the [package] section of the installation script file (*.INF): version, lowicver and highicver:

- The **version** parameter is an informational field which assigns a version identifier to the package.
- The **lowicver** parameter defines the minimum INFOConnect API level that the package requires.
- The **highicver** parameter defines the highest INFOConnect API level that the package utilizes.

Each parameter contains four version number fields: major version, minor version, EMU level, and build revision:

```
[package]
;'name' cannot contain blank characters
  name     = "Reflect"
  description = "Sample External Interface"
  version=  2, 0,0, 0
  lowicver= 2, 0,0, 0
  highicver= 3,00,0,000
/*      *                                    Major version */
/*        **                                       Minor version */
/*          *                                EMU Level    */
/*            ***                     Build revision */
```

All three parameters set fields of type IC_VER within the package table: pkg_version, low_icver, and high_icver. Installation scripts are discussed in detail in Section 9, "Packaging an INFOConnect Application."

Notes:

- *When highicver is set to "3, 0, 0, 0", which is equivalent to IC_VERSION_3_0 (0x0300) and IC_REVISION_3_0 (0x0000) or higher, the 3.0 Quick Configuration model is activated during installation.*
- *The Configuration Manager will also check the minimum and maximum version/revision numbers specified in the library's resource file (.RC) to verify that the library can handle the 3.0 Quick Configuration model.*

Windows 3.1 version information resource

The INFOConnect Installation Manager uses the version checking capabilities provided by the Windows 3.1 file installation library, VER.DLL, to determine when an existing file should be replaced by a newer version during installation. The version information is supplied in the library's resource file (.RC).

```
#ifdef WINSDKVER
#if (WINSDKVER >= 0x030a)
/* VER.DLL is only available in the Windows 3.1 SDK */
#include <ver.h>
#define VER_FILETYPE       VFT_DLL
#define VER_FILESUBTYPE     VFT_UNKNOWN
#define VER_FILEDESCRIPTION_STR QMARKETINGNAME
#define VER_INTERNALNAME_STR  QMODULEID
#define VER_FILEVERSION     NFILEVERSION
#define VER_PRODUCTVERSION   NPRODUCTVERSION
#define IC_FILEVERSION_STR   QVERSION
#define IC_PRODUCTVERSION_STR QVERSION
...
#include <icdef.rh>
#endif
#endif /* WINSDKVER */
```

The version information can be viewed by examining the library. To examine the library, open the INFOConnect Library Installation window by selecting Libraries from the Install menu, select the library you want to view, and then select the Examine button.

For a complete description of the version information resource section, refer to page 7 - 81.

INFOConnect RCDATA version information

The INFOConnect RCDATA section of the library's resource file contains a minimum ICS version/revision level and a maximum ICS version/revision level.

```
INFOConnect RCDATA
BEGIN
 IC_VERSION_2_0,
 IC_REVISION_2_0,

...

 /* The following fields are new for 2.02 */
 IC_VERSION_3_0,
 IC_REVISION_3_0,

...

END
```

IC_VERSION_2_0 and IC_REVISION_2_0 refer to the minimum (or oldest) level of Connectivity Services that the library requires for proper operation. Older levels of Connectivity Services will refuse to load the library.

IC_VERSION_3_0 and IC_REVISION_3_0 specify the maximum (or latest) level of Connectivity Services that the library was developed with; therefore, taking advantage of that level of ICS features.

The version information can be viewed by examining the library. To examine the library, open the INFOConnect Library Installation window by selecting Libraries from the Install menu, select the library you want to view, and then select the Examine button.

For more complete information on resource files, see "Resource Files" on page 7 - 74.

Library table serial numbers

Library table serial numbers are defined in the IC_RC_DICTIONARY RCDATA section of the library's resource file. If you need to add, change, or obsolete fields in a library table, you must identify the change by incrementing the table serial number. Incrementing the table serial number and providing upgrade code, if necessary, in IcLibUpdateConfig and IcLibVerifyConfig allows end-users to upgrade to your new library version preserving the existing configuration information. Refer to the "Modifying library tables" section on page 7 - 10.

Filtering Service Libraries

This section contains some general warnings that affect most service libraries that are performing some kind of filtering function.

Modifying the application's transmit buffers

This must be done at IcLibXmt.

Modifying the application's receive buffers

This must be done in IcLibEvent after intercepting IC_RCVDONE events. You may be tempted to do this at IcLibRcv, but the receive buffer is empty at that point.

Total buffer size vs. amount of data in the buffer

If you are inserting data into the buffer, be careful about exceeding the buffer's capacity. There are two different buffer sizes you are working with:

- The total size of the datacomm buffer
- The amount of data within the datacomm buffer.

Function IcGetBufferSize retrieves the total size of the global memory block. The *length* or *size* parameters on functions IcLibXmt, IcLibRcv, and IcLibEvent either specify the amount of data in or to be put in the buffer.

Breaking up the application's original buffer

Be careful about converting an application's transmit or receive request into multiple buffers and requests, particularly for sessions defined with block_mode=TRUE.

For example, suppose you are developing a service library that prefixes each transmitted buffer from the application with a header packet. Rather than allocate a new buffer and copy the header packet followed by the application's buffer, it seems advantageous to simply transmit the header packet, then transmit the application's buffer. However, two IC_XMTDONE events will be returned to the application. Your service must intercept and "swallow" one of them. Also, the receiving component of the distributed application may be expecting a single block rather than two pieces. Both of these problems can be addressed, but must be taken into account in your design.

Windows 3.x Issues

What is a DLL?

INFOConnect libraries are implemented as Windows Dynamic-Link Libraries (DLL) using native Windows function calls. The *Windows 3.x SDK Guide to Programming* contains a chapter on DLLs. You should study that chapter before writing an INFOConnect library.

How do tasks and stacks affect DLLs?

A task is the fundamental unit of scheduling in Windows. Applications run as tasks, and the INFOConnect Manager's window is running as a task. Services and interfaces are DLL libraries that are called: sometimes to run as part of application's task and sometimes to run as part of the Manager's task.

Unlike a task module, a DLL library does not have its own stack. Instead, it uses the stack segment of the task that called the DLL library. This situation is also succinctly stated as "the DS != SS issue."

What does this mean to the INFOConnect library developer?

Some system resources are associated with a particular task. Most of the required library routines (for example, IcLibOpenChannel) are running as part of the Manager's task. The exceptions are: IcLibGetString, IcLibGetSessionInfo, IcLibUpdateConfig, IcLibVerifyConfig and IcLibPrintConfig. For example, in Windows, a file handle returned to the Manager is meaningless to the application. If your library opens a file in IcLibInstall (running as part of the Manager task), it could not write to the file from IcLibGetSessionInfo (running as part of the application task) without reopening the file to obtain a new file handle.

Also be aware of how much stack space you are using for variable declarations, especially for recursive functions like IcLibXmt, IcLibRcv, and IcLibEvent that call each other. Since you don't have control of the INFOConnect Manager's stack size, you may need to use the local or global heap for some data items instead of the stack.

How do I write a device driver?

Some external interfaces may interface to, and require the development of, a lower device driver layer. There are several types of drivers: DOS device drivers, Windows device drivers and Virtual device drivers. See the *Windows 3.1 SDK Guide to Programming*, Section 20.2.4 , for Device Driver considerations. The development kit does not cover device driver development.

Writing the Required IcLib Functions

Each service and each interface defines and exports a list of required functions called by the manager. See the *IDK Programmer's Reference Manual* for the ordinal values to use when exporting the required functions.

IcLibInstall

```
IC_RESULT FAR PASCAL IcLibInstall(IC_RESULT_CONTEXT context)
```

This procedure is called once by the INFOConnect Connectivity Manager when the library is loaded. You should do the bulk of your initialization here rather than in LIBMAIN. Don't forget to save the context passed to you as a parameter. It is used later during error and status processing.

IcLibInstall can return any type of error except one: library-specific errors of type IC_ERROR_TERMINATE. This is because terminate errors cause your library to be immediately unloaded and therefore your library can't be called at IcLibGetString. Severe errors will still allow your library to be loaded and you can expect IcLibTerminate to be called later in those cases.

IcLibInstall is called whenever your library is loaded whether for configuration or session establishment.

Return Value

Return IC_OK if installation completes successfully. Return a standard IC_RESULT error otherwise.

Note: *See Appendix C of the IDK Programming Reference Manual for possible errors.*

Sample code

```
/* Global variables */
IC_RESULT_CONTEXT LibContext;

IC_RESULT FAR PASCAL IcLibInstall(IC_RESULT_CONTEXT context)
{
 /*
   This procedure is called once by INFOConnect
   when the library is loaded.

   Return IC_OK if initialization is successful.
 */
 int i;

 LibContext = context;
 ...
 return IC_OK;
}
```

IcLibTerminate

```
IC_RESULT FAR PASCAL IcLibTerminate (void)
```

This procedure is called once by INFOConnect when the INFOConnect Communication Services are being closed and the library is no longer needed. It is called after IcLibCloseSession and IcLibCloseChannel. You should do the bulk of your termination here rather than in WEP. Free all resources allocated by your library. Don't forget to remove or kill any timers you may have started in IcLibInstall.

Return Value

Return IC_OK if termination completes successfully. Only standard IC_RESULT errors can be returned from this procedure.

IcLibUpdateConfig

```
IC_RESULT FAR PASCAL IcLibUpdateConfig(HIC_CONFIG hConfig,
            UINT    TableNumber,
            void FAR* buffer,
            UINT    len,
            IC_COMMAND Command)
```

This function is called when your library is expected build or update one of its configuration tables. Normally this involves presenting a dialog box to the user.

The *hConfig* parameter is a handle to a configuration session.

The *TableNumber* parameter indicates which of your library's tables is being manipulated.

The *Command* parameter indicates what type of action the user is taking: add, modify or examine. For IC_CMD_EXAMINE, be careful about showing editable fields, otherwise, the user is led to believe that changes can be made when in fact they are discarded. The sample libraries in the development kit, for example, have taken liberties and do not follow this practice.

Two new IC_COMMAND types have been added in INFOConnect 3.0: IC_CMD_SAVE and IC_CMD_DISCARD. IC_CMD_SAVE is received immediately before the data is saved to the data base. IC_CMD_DISCARD is received when data from a previous call to IcLibUpdateConfig or IcLibVerifyConfig is being discarded. For example, IC_CMD_DISCARD is returned to IcLibUpdateConfig when a user cancels during a dialog associated with IcLibUpadateConfig.

Use IcDialogConfig instead of the Windows API for dialog box manipulation since it uses HIC_CONFIG as a parameter.

Return Value

IC_OK is returned if successful. If the user canceled from the dialog, return IC_CANCELED. IC_ERROR_UNKNOWN_COMMAND must be returned for any unknown *Command*. Otherwise, return a standard or library specific error.

Sample Code

```
IC_RESULT FAR PASCAL IcLibUpdateConfig (HIC_CONFIG hConfig,
                    UINT    TableNumber,
                    void FAR *buffer,
                    UINT    len,
                    IC_COMMAND Command)
{
 /*
   The user is updating something in the configuration.
   If appropriate, present a dialog box.
  */

 IC_RESULT icerror = IC_OK;

 switch (TableNumber) {
   case IC_PATHTABLE_BASENO:
    switch (Command) {
     case IC_CMD_ADD:
     case IC_CMD_MODIFY:
     case IC_CMD_EXAMINE:
      assert (len == sizeof(path_config));
      icerror = IcDialogConfig(hConfig, hLibInstance,
                    IDD_PATH_CONFIG,
                    (FARPROC)dbPathConfigDlg,
                    (DWORD)buffer);
      break;

     case IC_CMD_COPY:
     case IC_CMD_DELETE:
     case IC_CMD_SAVE:
     case IC_CMD_DISCARD:
      break;

     case IC_CMD_ABOUT:
     default:
      icerror = IcSetSessionError(NULL, LibContext,
                     IC_ERROR_UNKNOWN_COMMAND,
                     NULL, NULL, NULL);
     } /* end of swtich(command) */
    break; /* end of case IC_PATHTABLE_BASENO */

   case IC_CHANNELTABLE_BASENO:
    /* insert code for the channel table */

    ...

    break; /* end of case IC_CHANNELTABLE_BASENO */

   default:
    icerror = IcSetSessionError(NULL, LibContext,
                  IC_ERROR_UNKNOWN_TABLE,
                  NULL, NULL, NULL);
    break;
   } /* end of switch(TableNumber) */
 return icerror;
}
```

IcLibVerifyConfig

```
IC_RESULT FAR PASCAL IcLibVerifyConfig(HIC_CONFIG hConfig
            UINT    TableNumber,
            void FAR* buffer,
            UINT    len,
            IC_VERIFY Command)
```

This function is called to perform semantic checking on a record from one of your library tables.

The IC_VER_UPGRADE command tells the library to perform special upgrade processing and data conversions on the given buffer of data.

The IC_VER_DELETE command tells the library that the given configuration data is about to be deleted. If the library returns a sever error, the data will not be deleted.

The IC_VER_SAVE command (called IC_VER_NODISPLAY in ICS Release 2.0) tells the library that the configuration data is about to be saved. If the library returns a severe error, the data will not be saved.

Return Value

Return IC_OK if successful. IC_ERROR_UNKNOWN_COMMAND must be returned for any unknown *Command*. Otherwise, return a standard or library specific error.

Sample Code

```
IC_RESULT FAR PASCAL IcLibVerifyConfig (HIC_CONFIG hConfig,
                    UINT    TableNumber,
                    void FAR *buffer,
                    UINT    len,
                    IC_VERIFY Command)
{
 IC_RESULT icerror = IC_OK;

 NOREF(hConfig);
 NOREF(TableNumber);
 NOREF(buffer);
 NOREF(len);

 if ((TableNumber != IC_PATHTABLE_BASENO) ||
    (TableNumber != IC_CHANNELTABLE_BASENO))
   return IcSetSessionError(NULL, LibContext, IC_ERROR_UNKNOWN_TABLE,
              NULL, NULL, NULL);

 switch (Command) {
   case IC_VER_DISPLAY:
   case IC_VER_MODIFY:
   case IC_VER_SAVE:
   case IC_VER_UPGRADE:
   case IC_VER_DELETE:
     break;

   default:
     icerror = IcSetSessionError(NULL, LibContext,
                IC_ERROR_UNKNOWN_COMMAND,
                NULL, NULL, NULL);
     break;
   }
 return icerror;
}
```

IcLibPrintConfig

```
IC_RESULT FAR PASCAL IcLibPrintConfig (UINT        TableNumber,
                  IC_PRINT_DETAIL  detail,
                  void FAR *    buffer,
                  UINT        len,
                  LPSTR        print,
                  UINT        prtlen)
```

This function is expected to format and return a displayable text string summarizing a record from one of your library tables. Alternate ways of viewing INFOConnect configurations can be developed using this function. The INFOConnect Manager uses this function to obtain the description which is displayed in the INFOConnect Library Object Configuration window.

Return Value

IC_OK is returned if successful. IC_ERROR_UNKNOWN_COMMAND must be returned for any unknown d*etail*. Otherwise, return a standard or library specific error.

IcLibOpenChannel

```
IC_RESULT FAR PASCAL IcLibOpenChannel(HIC_CHANNEL    hIcChannel,
            void FAR*    buffer,
            UINT        len,
            IC_OPEN_OPTIONS Options,
            LPHIC_SESSION  lphLibChannel)
```

This function is called to initialize a library channel. It is called once before any sessions (for example, paths) that use the channel are opened. Since a session is being established, IcLibOpenChannel is a good place to perform additional session related initialization not performed in IcLibInstall. For control flow, see *Processing an open session request* in Section 6, "A Closer Look at the ICS Architecture."

IcLibOpenChannel should return promptly to the caller. Displaying a dialog box here and waiting for the user's response before returning to the Manager must be avoided. That goes against the spirit of Windows-style cooperative multi-tasking.

Libraries are not required to define library channels. If a library doesn't define a channel table, this function is still called once with a NULL *buffer* and zero *length*. The library should perform any session related global initialization and return IC_OK.

The lphLibChannel parameter is used with channel aliasing which is explained on page 7 - 53.

Return Value

IC_OK is returned if successful, otherwise an IC_RESULT error. The open process will continue for errors of type IC_ERROR_WARNING and IC_ERROR_INFO. Errors of type IC_ERROR_TERMINATE and IC_ERROR_SEVERE will cause the open process to fail.

IcLibCloseChannel

```
IC_RESULT FAR PASCAL IcLibCloseChannel(HIC_CHANNEL hLibChannel)
```

This function is called to terminate a channel after all sessions associated with this channel have been closed.

IcLibCloseChannel is the opposite of IcLibOpenChannel and is therefore very similar. Normally, IcLibCloseChannel just undoes everything done by IcLibOpenChannel. When you are looking for things to do in IcLibCloseChannel, look in:

- IcLibOpenChannel
- IcLibOpenSession for items not handled by IcLibCloseSession
- IcLibEvent/IC_SESSIONESTABLISHED for items not handled by IcLibEvent/IC_SESSIONCLOSED or IcLibCloseSession.

Return Value

Return IC_OK if successful. Otherwise, return a standard or library specific error.

IcLibOpenSession

```
IC_RESULT FAR PASCAL IcLibOpenSession (HIC_SESSION   hIcSession,
            HIC_CHANNEL   hLibChannel,
            void FAR*   Buffer
            UINT   len,
            IC_OPEN_OPTIONS  Options,
            LPHIC_SESSION   phLibSession)
```

This function is called to initialize a session.

When an application makes an open session request, it eventually results in a call to IcLibOpenSession. For control flow, see "Processing an Open Session Request" in Section 6, "A Closer Look at the INFOConnect Architecture."

If IcLibOpenSession returns IC_OK, but you later determine (perhaps in a timer routine or during IcLibEvent) that the session can't be opened, do not call IcCloseSession to close the session. Instead, you must request the application to close the session by generating an error of type IC_ERROR_TERMINATE using an IC_ERROR message sent with IcMgrSendEvent. Do not call IcMgrSendEvent until IcLibEvent has processed the IC_SESSIONESTABLISHED event.

IcLibOpenSession should return promptly to the caller. Displaying a dialog box here and waiting for the user's response before returning to the Manager should not be done. That goes against the philosophy of Windows-style cooperative multi-tasking.

Notes for External Interfaces

The external interface's IcLibOpenSession is called before the service library, therefore, your external interface should not make any service library calls during IcLibOpenSession processing. Simply set the session to the idle state and wait for the IC_SESSIONESTABLISHED event to be passed to IcLibEvent.

If your interface starts a timer routine, make sure your timer code doesn't start making calls to the service library until the IC_SESSIONESTABLISHED event has been received by IcLibEvent or after the IC_SESSIONCLOSED event has been received by IcLibEvent.

Return Value

IC_OK, IC_ERROR_INFO or IC_ERROR_WARNING result type is returned if the open was successful. IC_VERIFY_OK if the verify was successful. Otherwise, return a standard or library specific error.

IcLibCloseSession

```
IC_RESULT FAR PASCAL IcLibCloseSession(HIC_SESSION   hLibSession,
              HIC_CHANNEL   hLibChannel)
```

This function is called to terminate a session. All session-related data should be cleaned up.

IcLibCloseSession is the opposite of IcLibOpenSession and is therefore very similar. Normally, IcLibCloseSession just undoes everything done by IcLibOpenSession. When you are looking for things to do in IcLibCloseSession, look in IcLibOpenSession.

Libraries running with ICS 3.0 that need to delay the session close processing may want to delay in IcLibLcl/IC_LCL_CLOSESESSION and/or IcLibEvent/IC_SESSIONCLOSED, rather than delay in IcLibCloseSession.

For libraries running with ICS 2.0, an IC_SESSIONCLOSED message does not get passed to your library at IcLibEvent, as does an IC_SESSIONESTABLISHED message during session open processing. Therefore, any resources or processing done by IcLibEvent/IC_SESSIONESTABLISHED may need to be released in IcLibCloseSession.

Return Value

If cleanup is successful, return IC_OK. Otherwise, return a standard or library specific error.

IcLibXmt / IcLibRcv

```
IC_RESULT FAR PASCAL IcLibXmt(HIC_SESSION  hLibSession,
        HANDLE    buffer,
        UINT      length)

IC_RESULT FAR PASCAL IcLibRcv(HIC_SESSION  hLibSession,
        HANDLE    buffer,
        UINT      length)
```

These functions are called to receive or transmit data from the specified buffer.

A successful return value (IC_OK or one of type IC_ERROR_INFO or IC_ERROR_WARNING) does not mean the request is complete, but that it has been initiated. The application expects to later receive an ICS event indicating the outcome of the request.

A return value of type IC_ERROR_SEVERE or IC_ERROR_TERMINATE means no further action will be taken by the library for this request. A bad return value is usually caused by an invalid session handle. The application does not expect to receive an ICS event in this case.

Generally, libraries do not "swallow" receive or transmit requests from the application. Transmit requests must eventually return an IC_XMTDONE or IC_XMTERROR event and receive requests must return an IC_RCVDONE or IC_RCVERROR event. An example of an exception to this is a service library that initiates transmits "behind the back" of the application. In this case, the resulting IC_XMTDONE event must be intercepted and "swallowed" or the application will probably get confused.

IcLibXmt and IcLibRcv are very closely tied to the IcLibEvent function and are often in a recursive calling arrangement. For example, a service library calls IcMgrXmt to push a transmit request down the stack. The EIL calls IcMgrSendEvent to pass an IC_XMTERROR event back up the stack before returning in its IcLibEvent function. This causes the service library to be called at IcLibEvent with the IC_XMTERROR. Any service library in the session could retry the transmit request by call IcMgrXmt again. It doesn't even have to be an error condition, a service library could call IcMgrXmt upon receiving an IC_XMTDONE event. This means you must be very careful about changing the state of your library after calling IcMgrXmt, because of the possible recursion. One technique is to update your internal structures as if the IcMgrXmt call was successful immediately before calling IcMgrXmt. Another way to state this warning is: when IcLibXmt calls IcMgrXmt it must be prepared to be called again itself before the IcMgrXmt call returns.

Here is another typical scenario illustrating the indirect recursion possible between IcLibXmt/IcLibRcv and IcLibEvent. Library A sits below Library B in the library stack. Library A's IcLibEvent routine calls IcMgrSendEvent. This results in Library B's IcLibEvent routine getting called. If Library B does a transmit ("behind the application's back") Library A's IcLibXmt will get called before the original IcMgrSendEvent call has ever returned.

The delivery of IC_XMTDONE and IC_RCVDONE events to the application are generally independent of each other. However, there are situations where the application can rightly expect a certain sequence of delivery for these two events. Consider the following scenario. Many applications typically maintain an outstanding receive request in preparation for receipt of messages from their partner component. If an application then does a transmit, it will logically expect the IC_XMTDONE event to be returned to it before the IC_RCVDONE event containing the partner's reply to the transmit.

Another stipulation: if an event is generated by one of these functions (for example IcLibXmt) with IcMgrSendEvent, the library is committed to returning IC_OK for that function (IcLibXmt) when the function does finally return. Otherwise, the earlier rule is broken that states that a return value not equal to IC_OK means the request is dead.

Be sure to use IcMgrXmt/IcMgrRcv (or IcLibraryXmt/IcLibraryRcv if running with 2.0 and 3.0) to post the transmit/receive request down to the next library in the library stack.

Notes for External Interfaces

An EIL must be aware that an application can use separate datacomm buffers in order to keep a receive request pending while waiting for a pending transmit request to complete. Even if the transport below is synchronous in nature, the EIL must be prepared to queue the requests from applications which are asynchronous in nature.

This does not mean that an EIL has to handle more than one transmit request (or more than one receive request) at a time from an application. The application should only have one pending receive request and one pending transmit request.

Return Value

IC_OK if the session is valid and the command can be processed. Otherwise, return a standard or library specific error.

IcLibLcl

```
IC_RESULT FAR PASCAL IcLibLcl(HIC_SESSION hLibSession,
          IC_LCL_FLAGS which)
```

This function is called to cancel pending transmit and/or receive requests. Unlike IcLibRcv and IcLibXmt, no event needs to be generated and passed to the application with IcMgrSendEvent. Since the application call to IcLcl is returned before any library is called, the Manager sends an IC_LCLRESULT event to the application on behalf of the libraries.

Be sure to use IcMgrLcl (or IcLibraryLcl if running with 2.0 and 3.0) in order to pass the message down to the next library in the library stack.

Note: *If the IC_LCL_CLOSESESSION is not passed down the stack the session will not close. If a library with a maximum version less than IC_VERSION_3_0 in the INFOConnect RCDATA resource does not call IcMgrLcl, a close session will be passed down the stack when the library returns.*

Return Value

IC_OK is returned if successful. Otherwise, return a standard or library specific error.

IcLibSetResult

```
IC_RESULT FAR PASCAL IcLibSetResult(HIC_SESSION hLibSession,
            UINT     uType,
            IC_RESULT  result)
```

This function is used to receive statuses and errors from the higher ICS components that were initiated by the application. For control flow, see "Status Initiated by the Application" in Section 6, "A Closer Look at the ICS Architecture."

Notes for Service Libraries

After any necessary processing, services must make a call to IcMgrSetResult (or IcSetResult if running with 2.0 and 3.0) unless the function of the library is to intercept the status call.

Notes for External Interfaces

Any unprocessed IC_RESULTs should be passed on to IcMgrSetResult (or IcSetResult if running with 2.0 and 3.0) rather than treated as errors.

Return Value

Often, your library will just pass up the IC_RESULT value returned from calling IcMgrSetResult itself. Otherwise, return a standard or library specific error.

Sample code

```
IC_RESULT FAR PASCAL IcLibSetResult(HIC_SESSION  hLibSession,
            UINT         uType,
            IC_RESULT        result)
{
 return IcMgrSetResult(hLibSession, uType, result);
}
```

IcLibEvent

```
IC_RESULT FAR PASCAL IcLibEvent(UINT      uType,
              HIC_SESSION  hLibSession,
              GLOBALHANDLE hBuff,
              UINT      uSize)
```

Parameters		Description
uType	IN	One of the following event types: **IC_RCVDONE** **IC_RCVERROR** **IC_XMTDONE** **IC_XMTERROR** **IC_ERROR** **IC_STATUS** **IC_SESSIONESTABLISHED** **IC_SESSIONCLOSED** **IC_SENDSTATUS**
hLibSession	IN	A session handle.
hBuff	IN	A handle to a global buffer or the HIWORD of an IC_RESULT, depending on uType.
uSize	IN	The buffer size in bytes or the LOWORD of an IC_RESULT, depending on uType.

This function allows libraries to process events passed from a lower layer in the INFOConnect architecture. After processing the event, it must be passed up to the next layer in the ICS architecture. See the control flow diagrams in Section 6, "A Closer Look at ICS Architecture."

The processing to be done in IcLibEvent is tied very closely with some of the other IcLib functions. For example, the code in the IC_SESSIONESTABLISHED case is closely related to the IcLibOpenSession processing.

A library must never generate or "bubble up" an IC_SESSIONESTABLISHED event for a session until it has first received the IC_SESSIONESTABLISHED event for that session. The Manager initiates this process by sending the first IC_SESSIONESTABLISHED event to the EIL at the bottom of the stack. Until then, all of the libraries in the stack are not initialized.

Libraries that support ICS 3.0 must be prepared for events with hLibSession set to NULL_HIC_SESSION. These are global events of possible interest to the library. The library should not pass these events up the stack by calling IcMgrSendEvent.

There are two instances where an IC_STATUS event with hLibSession set to NULL_HIC_SESSION is received by a library. The first is an IC_STATUS message of IC_COMMMGR_INITIALIZED and is received by all libraries which have the library entry field ICMGR_LIB_LOADFLAGS set to 1. This flag is used to control autoload. The second is an IC_STATUS message of IC_COMMMGR_TERMINATED and is received by a library if the library is still loaded when INFOConn.exe terminates. IC_COMMMGR_... status values are described in the *IDK Programming Reference Manual.*

Libraries that do not support ICS 3.0 or higher do not receive an IC_SESSIONCLOSED event. Libraries that do support ICS 3.0 will receive an IC_SESSIONCLOSED event in IcLibEvent. The IC_SESSIONCLOSED event must be passed up the stack in order for the session to close.

Note: *If the IC_SESSIONCLOSED is not passed up the stack the session will not close. If a library with a maximum version less than IC_VERSION_3_0 in the INFOConnect RCDATA resource does not call IcMgrSendEvent, a session closed event will be passed up the stack when the library returns.*

For all events that do not have hLibSession set to NULL_HIC_SESSION, be sure to use IcMgrSendEvent (or IcSendEvent if running with 2.0 and 3.0) to pass the event to the next higher layer in the library stack, including unknown events.

Note: *Only increasing the maximum level of ICS that your library supports in the INFOConnect resource section of the library's resource file causes the new status messages to be sent to your library. If a library specifies that it supports a minimum level of ICS 2.0 and a maximum level of ICS 3.0, it will receive ICS 3.0 status events. If your library only supports the 2.0 status events, use the 2.0 values to set the minimum and maximum level of ICS. See the Resource Files section for designating the minimum and maximum level of Connectivity Services that the library supports.*

Generating IC_STATUS and IC_ERROR events can be done using the macros HIWORD and LOWORD.

Using HIWORD and LOWORD with hBuff and uSize

The *hBuff* and *uSize* parameters on IcLibEvent are somewhat misleading for those event types that are actually passing in a single IC_RESULT type (which is a 'long' value) like IC_SESSIONESTABLISHED, IC_STATUS, and IC_ERROR. For this reason, you may occasionally need to use the HIWORD and LOWORD macros.

Here is a fragment of code from a library requesting the application to close the session. An IC_ERROR_TERMINATE_NOMSG error (an IC_RESULT defined in IcError.h) is bubbled up to the application. Notice the use of the HIWORD and LOWORD macros to pass a 'long' value through the *hBuff* and *uSize* parameters which are a single word each.

```
icerror=IcMgrSendEvent(hLibSession,hIC_ERROR,
          HIWORD(IC_ERROR_TERMINATE_NOMSG),
          LOWORD(IC_ERROR_TERMINATE_NOMSG));
if IC_CHECK_RESULT_SEVERE(icerror)
 ReturnInternalError(hLibSession,INTERNAL_ERROR_10);
```

Return Value

Return IC_OK if the message is valid and can be processed for the given session. Otherwise, return a standard or library specific error.

Sample code

```
IC_RESULT FAR PASCAL IcLibEvent(UINT       uType,
            HIC_SESSION  hLibSession,
            GLOBALHANDLE  hBuff,
            UINT       uSize)
{
 /*
  Event types that your library doesn't handle
  MUST be passed on to the application.
 */

 switch (uType) {
  case IC_SESSIONESTABLISHED:
  case IC_SESSIONCLOSED:
  case IC_XMITDONE:
  case IC_RCVDONE:
  case IC_XMTERROR:
  case IC_RCVERROR:
  case IC_STATUS:
  case IC_ERROR:
  case IC_SENDSTATUS:
  default:
   return IcMgrSendEvent( hLibSession, uType, hBuff, uSize );
   break;
 }
}
```

IcLibGetSessionInfo

```
IC_RESULT FAR PASCAL IcLibGetSessionInfo(HIC_SESSION   hLibSession,
                 LPIC_SINFO   sinfo)
```

IcLibGetSessionInfo is called to build an IC_SINFO record that describes the current session's attributes. The external interface sets the IC_SINFO structure which has been initialized to zeros by the Manager. The IC_SINFO structure and the meaning of each session attribute is described in the *IDK Programming Reference Manual*.

Notes for Service Libraries

Service libraries should only modify those attributes that pertain to the task of the service. Some services may not need to change any IC_SINFO fields.

Notes for External Interfaces

Interfaces should initialize all defined fields in the IC_SINFO structure. Reserved fields are set to zero by the Manager.

Return Value

IC_OK is returned if successful. Otherwise, return a standard or library specific error.

Sample code for service library

```
IC_RESULT FAR PASCAL IcLibGetSessionInfo(HIC_SESSION hLibSession,
                 LPIC_SINFO sinfo)
{
 /*
   The external interface must initialize all SINFO fields.
   The service library should only change ones it is
   specifically going to manage.

   See the INFOConnect Reference Manual for descriptions
   of the different attributes.
 */

 return IC_OK;
}
```

Sample code for external interface

```
IC_RESULT FAR PASCAL IcLibGetSessionInfo(HIC_SESSION hLibSession,
                        LPIC_SINFO sinfo)
{
 /*
   The external interface must initialize all SINFO fields.

   See the INFOConnect Reference Manual for descriptions
   of the different attributes.
 */

 sinfo->max_size=MAXRECORDSIZE;
 sinfo->transparent=TRUE;
 sinfo->block_mode=TRUE;
 sinfo->reliable=TRUE;
 sinfo->focus_notify=FALSE;
 return IC_OK;
}
```

IcLibGetString

```
IC_RESULT FAR PASCAL IcLibGetString(HANDLE    hData,
            IC_RESULT  result,
            LPSTR    buffer,
            UINT    length)
```

This function retrieves a null-terminated text string associated with a library specific error for your library. Status messages do not have text strings associated with them.

hData is the library handle of the communication session on which the error occurred or NULL_HIC_SESSION if the error is not associated with any session.

Try to maximize your use of standard errors before defining unique errors. Application developers are more likely to code for standard errors and possibly take some corrective action without bothering the user with an error message. Upon receiving a non-standard error, an application's most likely course of action is to simply display it.

IcLibGetString is called after your library has returned a unique error and the application calls IcGetString to construct an error message to display. If an error message contains an insert, the text of the insert must be saved when the error is generated (perhaps in the library's session record) so that the IcLibGetString function can later retrieve it.

The message text should not contain either your library name or an error number; that information is retrieved by the caller through other means. To prepare for internationalization, the text for the error should be retrieved from a resource file.

Return value

Return IC_OK if successful. Otherwise, return a standard or library specific error.

Sample code

This code fragment was taken from Reflect.c, a sample interface library. Errors are defined in Reflect.hic and the error text is in Reflect.rc. Notice how the *value* portion of the error is used as the stringtable index of the error message text.

```
IC_RESULT FAR PASCAL IcLibGetString(HANDLE    hData,
                    IC_RESULT  icresult,
                    LPSTR     buffer,
                    UINT      length)
{
  /*
    Return text for library specific errors.
  */

  IC_RESULT_CONTEXT iccontext;
  IC_RESULT_TYPE  ictype;
  IC_RESULT_VALUE  icvalue;

  iccontext = IC_GET_RESULT_CONTEXT(icresult);
  ictype = IC_GET_RESULT_TYPE(icresult);
  icvalue = IC_GET_RESULT_VALUE(icresult);

  if (LoadString(hLibInstance,icvalue,buffer,length) != 0)
    return IC_OK;
  else
    return(IcSetSessionError(hData,LibContext,IC_ERROR_INTERNAL,
        "GetString/LoadString",NULL,NULL));
}
```

Header file containing error definitions

```
/*************************************************************/
/* REFLECT.HIC                        */
/*************************************************************/

#define REFLECT_CONTEXTSTRING "REFLECT"

...

/* status types */
#define REFLECT_STATUS_PING 1

/* error types */
#define REFLECT_ERROR_SEVERE IC_ERROR_SEVERE

/* values and string numbers correspond */
#define REFLECT_ERROR_XMTERROR    1
#define REFLECT_ERROR_RCVERROR    2
```

Resource file containing error text

```
/**************************************************/
/* REFLECT.RC                                     */
/**************************************************/

#include <windows.h>
#include <icdef.h>
#include <icdict.h>
#include "reflect.h"

...

STRINGTABLE
BEGIN
...
  REFLECT_ERROR_XMTERROR, "XmtError"
  REFLECT_ERROR_RCVERROR, "RcvError"
...
END
...
```

IcLibIdentifySession

HANDLE FAR PASCAL IcLibIdentifySession (HIC_SESSION hLibSession)

This function is called for each session to retrieve a unique session identifier if multiple instances of a path can be active. Multiple instances of a path can be active when all libraries in a session return IC_VERIFY_OK in IcLibOpenSession/IC_OPEN_VERIFY.

Return Value

For sessions that can be opened over a single path, the library can return a handle to a global buffer (allocated through IcAllocBuffer) containing a unique alphanumeric identification string up to IC_MAXSESSIONIDLEN characters. Otherwise, return (HANDLE)NULL.

Sample code

```
HANDLE FAR PASCAL IcLibIdentifySession (HANDLE hSession)
{
 return (HANDLE)NULL;
}
```

Windows DLL requirements

Windows requires all DLLs to provide an initialization routine, LIBMAIN, and a termination routine, WEP. The *Windows 3.x SDK Guide to Programming* covers these routines in more detail.

LIBMAIN

LIBMAIN is called directly by Windows when the DLL is loaded. You should only do Windows-related tasks here. Don't do any INFOConnect-related tasks here.

```
BOOL FAR PASCAL LIBMAIN(HANDLE hInstance,
            WORD wDataSeg,
            WORD wHeapSize,
            LPSTR lpCmdLine)
{
  /*
    This is a Windows-specific function required for all
    DLL libraries. It is called by Windows when the
    library is initially loaded.
  */

  NOREF(wDataSeg);
  NOREF(wHeapSize);
  NOREF(lpCmdLine);

  LibInstance = hInstance;
  return TRUE;
}
```

WEP

WEP is called directly by Windows when the DLL is unloaded. You should only do Windows related tasks here. Don't do any INFOConnect-related tasks here. The following code should usually be sufficient for your use.

```
int FAR PASCAL WEP (int nParameter)
{
 /*
   This is a Windows-specific function required for all
   DLL libraries. It is called by Windows when the
   library is being unloaded.
 */

 if (nParameter == WEP_SYSTEM_EXIT)
   return (1);
 else if (nParameter == WEP_FREE_DLL)
   return (1);
 else
   return (1);
}
```

Other Procedures and Guidelines

Session and Channel Aliasing

Aliasing is a way for INFOConnect and your library to exchange "nicknames" for session and channel handles.

Why should I use aliases?

Aliases are not required, but are provided as a convenience and performance boost to library execution. To see the advantage of aliases, compare the two code fragments that follow. Both code fragments are from a library that uses a linked list of session records. Whenever the library is called, the proper session record must be located. Notice the code in IcLibRcv in the first code fragment that calls GetSessionRec to scan all session records for *hSession*. In the second code fragment, aliases are used. INFOConnect passes the library a session handle that the library just locks to access the proper session record; no searching is necessary.

Aliases are assigned by your library in IcLibOpenChannel and IcLibOpenSession. They allow you to avoid a lot of list-searching. Most of the sample libraries in the development kit use aliasing.

Sample code fragment without aliases

```
typedef struct aSessionRec {
  struct    aSessionRec *pNextSessionRec;
  HIC_SESSION hSession; /* INFOConnect session handle */
  ...
} SESSIONREC;

typedef struct aSessionRec *PSESSIONREC;

IC_RESULT FAR PASCAL IcLibRcv (HIC_SESSION hLibSession,
              HANDLE    buffer,
              UINT      length )
{
  PSESSIONREC pSession;

  /* First, search for the list entry containing hLibSession */
  pSession = GetSessionRec (hSession);
  /* Now do my normal processing */
  ...
  return IcMgrRcv(hLibSession, buffer, length );
}

PSESSIONREC GetSessionRec  (HIC_SESSION hLibSession)
{
  PSESSIONREC p;

  p = pSessionNode;
  while ((p != NULL) && (p->hSession != hLibSession))
    p = p->pNextSessionRec;
  if ((p == NULL) || (p->hSession != hLibSession))
    return NULL;
  else
    return p;
}
```

The memory for the session record created in AddSessionRec is allocated using LocalAlloc (LPTR, len). The LPTR flag is used to allocate fixed memory. This allows the sample to define LockLibSession which actually just typecasts a handle to a PSESSIONREC and to define an UnlockLibSession which actually just returns. This explains why the code fragment above does not lock or unlock memory. The next sample, which uses aliases, shows the use of the LockLibSession and UnlockLibSession.

Sample code fragment using aliases

```
#define   LockMem(handle) (handle)
#define   UnlockMem(handle)
#define   LockLibSession(h) ((PSESSIONREC)LockMem(h))
#define   UnlockLibSession(h) UnlockMem(h)

typedef struct aSessionRec *PSESSIONREC;

typedef struct aSessionRec {
  struct aSessionRec *pNextSessionRec;
      /* Here are two handles for the same session */
  HIC_SESSION hIcSession; /* INFOConnect session handle */
  HIC_SESSION hLibSession; /* My library's session handle */
  ...
} SESSIONREC;

IC_RESULT FAR PASCAL IcLibRcv (HIC_SESSION  hLibSession,
               HANDLE     buffer,
               unsigned   length )
{
  HIC_SESSION hIcSession;
  PSESSIONREC pSession;

  /* INFOConnect passes me my own session handle that I */
  /* can immediately lock. No need to search.        */
  pSession = LockLibSession(hLibSession);
  /* Before I call IcMgrRcv(), I must retrieve the */
  /* INFOConnect session handle.              */
  hIcSession = pSession->hIcSession;
  ...
  UnlockLibSession(hLibSession);
  return IcMgrRcv( hIcSession, buffer, length );
}
```

Establishing aliases

Here is the code necessary to establish a session alias in IcLibOpenSession. Similar processing is used in IcLibOpenChannel to establish a channel alias.

Notice how the local pointer, pSession, is cast and returned to INFOConnect as the session alias. This scheme only works for local pointers. Far pointers are two words in size; HIC_SESSION is one word.

```
IC_RESULT FAR PASCAL IcLibOpenSession(HIC_SESSION   hIcSession,
                HIC_CHANNEL   hLibChannel,
                void FAR *   lpConfigBuf,
                UINT      len,
                IC_OPEN_OPTIONS Options,
                LPHIC_SESSION  lphLibSession)
{
 PSESSIONREC pSession;
 IC_RESULT icerror=IC_OK;

 pSession=AddSessionRec(hLibChannel,hIcSession,
          lpConfigBuf,len);

 ...
 *lphLibSession=MakeSessionHnd(pSession);
 return icerror;
}
```

Using aliases

Once an alias is established, all calls from INFOConnect to your library will use either *hLibSession* or *hLibChannel* and all calls that your library makes to INFOConnect must use *hIcSession* or *hIcChannel*. Its a good idea for your session and channel records to contain both names. The following sample libraries in the development kit use aliasing: Service, Intrface, and IcStack2.

Generating Errors from your Library

Your library can generate standard errors or define and issue library-specific errors. There are two methods of reporting errors: sometimes with the function's return value and sometimes with IC_ERROR events.

Generating standard errors

Standard errors are defined in IcError.h. Messages can have from zero to three inserts. Appendix C of the *IDK Programming Reference Manual* documents which standard errors use text inserts. Function IcSetSessionError must be used when returning standard errors to enable INFOConnect to capture which library generated the error. The following code fragment returns one error with a single insert and a second error without inserts. *LibContext* is the context saved earlier from the IcLibInstall call to your library.

```
IC_RESULT_CONTEXT LibContext;/* set in IcLibInstall */

IC_RESULT FAR PASCAL IcLibOpenSession(HIC_SESSION  hSession,
                HIC_CHANNEL  hChannel,
                void FAR *  lpConfigBuf,
                unsigned    len,
                BOOL       bVerify,
                LPHIC_SESSION lphSession)
{
 /*
   This function is called to initialize a library session.
   Return IC_OK or IC_VERIFY_OK for successful completion.
 */

 IC_RESULT icerror = IC_OK;

 if (!lpConfigBuf)
   return(IcSetSessionError(hSession,
              LibContext,
              IC_ERROR_INTERNAL,
              "IcLibOpenSession",
              NULL,NULL));
 if (!OpenSession(hSession, hChannel, lpConfigBuf))
   return(IcSetSessionError(hSession, LibContext,
        IC_ERROR_PICHANNELINUSE,
        NULL, NULL, NULL));
 ...
 return icerror;
}
```

Generating library-specific errors

Errors unique to your library must be defined in your .HIC header file and must include your library's context passed to you at function IcLibInstall. Errors are defined with type IC_RESULT which is a 'long' made up of three parts: a context, a type and a value.

```
IC_RESULT_CONTEXT LibContext; /* set in IcLibInstall */

IC_RESULT FAR PASCAL IcLibRcv (HIC_SESSION  hSession,
              HANDLE  hBuffer,
              unsigned length  )
{
 if (length < ENCODEDBUFSIZE) {
   return (IC_MAKE_RESULT(LibContext,
             REFLECT_ERROR_SEVERE,
             REFLECT_ERROR_XMTERROR);
 }
 ...
}
```

Defining library-specific errors

Although statuses and errors use the same structure (typedef IC_RESULT) they are never used in the same context. Therefore, a status and error can have the same type and value without being confused.

Here is the HIC header file defining the error used in the above code fragment:

```
/**********************************************************/
/* REFLECT.HIC                              */
/**********************************************************/

#define REFLECT_CONTEXTSTRING "REFLECT"

...

/* status types */
#define REFLECT_STATUS_PING 1

/* error types */
#define REFLECT_ERROR_SEVERE IC_ERROR_SEVERE

/* values and string numbers correspond */
#define REFLECT_ERROR_XMTERROR    1
#define REFLECT_ERROR_RCVERROR    2
```

Displaying library-specific errors

Never display errors directly from your library. After you have returned an error to the application, your library will be called at IcLibGetString and is expected to return the associated text for an error message. See the section on IcLibGetString for more information.

Generating IC_ERROR messages

Sometimes you may find it necessary to use HIWORD and LOWORD when generating IC_ERROR events. This is how a double-word 'long' value can be passed in using the two single-word parameters of IcMgrSendEvent.

```
IC_RESULT icresult, icerror;
...
icresult = ...
icerror = IcMgrSendEvent(hLibSession, IC_ERROR,
            HIWORD(icresult),
            LOWORD(icresult));
if IC_CHECK_RESULT_SEVERE(icerror)
  return icerror;
```

Requesting Session Termination

Occasionally, a situation may require the session to be closed. INFOConnect libraries do not close sessions directly themselves. Instead, you can request the application to close the session by issuing any error with a type of IC_ERROR_TERMINATE.

A standard error, IC_ERROR_TERMINATE_NOMSG, is also available for situations when the session should go away silently without any message to the user. Here is some code taken from a service modifier that is sending a "silent" termination request to the application. Notice the use of HIWORD and LOWORD to break the 'long' IC_RESULT data type down into two single words.

```
icerror=IcMgrSendEvent(hLibSession,IC_ERROR,
        HIWORD(IC_ERROR_TERMINATE_NOMSG),
        LOWORD(IC_ERROR_TERMINATE_NOMSG));
if IC_CHECK_RESULT_SEVERE(icerror)
  ReturnInternalError(hLibSession,INTERNAL_ERROR_10);
```

On-line Help

Your configuration dialog should contain a Help button that invokes the Windows help system with a specific on-line help file for your library. The following topics should be covered in the on-line help (.HLP) :

- The purpose and capabilities of your library.
- The configuration dialog for your library.
- Error help text, if your library sets IC_LF_ERROR_HELP in the library_flags field of its INFOConnect RCDATA resource.

The following files are used to build the .HLP file:

File	Description
.DOC	Microsoft Word document (optional)
.RTF	Rich Text Format file
.HLP	final on-line help file used by Windows
HELP.STY	MSWord style sheet (optional, WORD for DOS only)
.HPJ	Help Project File - Help Compiler control file
RTF_DOS.E XE	MSWord utility to convert .DOC and .STY -> .RTF (optional, WORD for DOS only)
.HH	.H header file defining context-sensitive IDs (optional)

The *Tools Reference Manual* of the Windows SDK covers the generation of on-line help files in detail, but here is a summary of the steps involved.

Build a .RTF file containing your on-line help text

The Windows help system uses .RTF (Rich Text Format) files. You can use any word processor that supports the .RTF format.

Create the Windows-compatible on-line help file (.HLP)

Create an .HPJ file and then run the Windows SDK Help Compiler to convert the .RTF file into a .HLP file.

If you use Microsoft Word for DOS 5.0 or later, you may find the RTF_DOS.EXE utility useful to convert standard MSWord documents into .RTF files.

Call IcRunHelp from your configuration dialog callback function

By using IcRunHelp, your library doesn't have to know where your help file is installed. Here is a sample callback function that uses IcRunHelp.

```
BOOL FAR PASCAL doConfigDlg(HWND   hDlg,
              unsigned message,
              WORD   wParam,
              LONG   lParam)
{
 IC_RESULT icerror;

 switch (message) {
  case WM_COMMAND: {
   switch (wParam) {
    case ID_HELP:
     icerror = IcRunHelp("ModuleName.HLP",
           (DWORD)LibContextID);
     break;
    case IDOK:
     EndDialog( hDlg, TRUE );
     break;
    case IDCANCEL:
     EndDialog( hDlg, FALSE );
     break;
    default:
     return FALSE;
     break;
   }
   return TRUE;
  }
  break;
  case WM_INITDIALOG:
   return TRUE;
   break;
  default:
   return FALSE;
 }
 return FALSE;
}
```

Note: *The module name (IC_MODULEID as defined in the library's resource file) with a .HLP extension should be used for the help file name. The Manager uses the file name of a loaded library to construct the fully qualified help file name by changing the extension to .HLP. If the library is not loaded, the Manager uses the filename in the configuration database with a .HLP extension.*

See the sample INFOConnect libraries on the IDK disk for a complete example of providing on-line help support.

Modifying Global Variables

Be careful about modifying global variables outside of your IcLibInstall and IcLibTerminate routines. Will multiple session threads "step on each other"?

Communicating with Applications Using Status Messages

Generally, applications and libraries should avoid nonstandard statuses. However, sometimes it is quite appropriate for an application and library to communicate using status messages. Although status messages are small, large blocks of information can be exchanged by storing the handle of an INFOConnect status buffer (HIC_STATUSBUF) in an IC_STATUS_BUFFER extended status. This section illustrates and gives some guidelines for defining an IC_STATUS_BUFFER extended status that incorporates a library specific status.

The code fragments shown in this section are taken from the IcWinApp sample application and the Reflect sample external interface library. These samples can be built to communicate with an extended status which contains a library specific status, REFLECT_STATUS_PING.

IcWinApp's modified menu bar contains a special menu item for initiating a REFLECT_STATUS_PING message. If running over a path configured with the Reflect library, Reflect will modify the string stored in the IC_STATUSBUF structure and sends the extended status back to IcWinApp. IcWinApp displays the string returned from REFLECT.

To activate the PING code, you must add

```
#define PING
```

to the header files IcWinApp.h and Reflect.h and then rebuild both samples.

Define the status message in a .HIC file

Either the accessory or the library can define the status type, but typically the library defines them. Both components must then include the defining HIC file.

In the PING sample, the REFLECT_STATUS_PING status message is defined in Reflect.hic. The ICVALUE portion (low word) of an IC_STATUS_BUFFER extended status contains the handle of an IC_STATUSBUF buffer. The icstatus field of the IC_STATUSBUF structure contains a library specific status with a status type of REFLECT_STATUS_PING.

```
/**********************************************************/
/* REFLECT.HIC                    */
/**********************************************************/

#define REFLECT_CONTEXTSTRING "REFLECT"

...

/* status types */
#define REFLECT_STATUS_PING 1

/* error types */
#define REFLECT_ERROR_SEVERE IC_ERROR_SEVERE

/* values and string numbers correspond */
#define REFLECT_ERROR_XMTERROR    1
#define REFLECT_ERROR_RCVERROR    2
```

Obtaining Contexts

If the new status message is defined in the application's HIC file, the application must call IcRegisterAccessory to obtain a context. This implies that the application satisfies all the requirements of an INFOConnect accessory. This is one reason to define the status in the library's HIC file instead.

The partner process uses IcGetContext and the context string from the .HIC file to obtain the runtime context. IcGetContext normally returns IC_OK or IC_CONTEXTSTRING_NOT_FOUND. You should decide if the "no find" situation is an error that should be displayed or not.

In the PING sample, the REFLECT_STATUS_PING status is defined in the Reflect library's .HIC file. IcWinApp obtains the context of Reflect using REFLECT_CONTEXTSTRING which is also defined Reflect.hic. Notice that the "no find" situation is handled specifically by IcWinApp, but all other errors are passed to the default error procedure.

```
IC_RESULT icerror;
IC_RESULT_CONTEXT LibContext = IC_RESULT_CONTEXT_INVALID;

icerror = IcGetContext(REFLECT_CONTEXTSTRING, &LibContext);
if (icerror != IC_OK) {
  MessageBox(hWnd, "Reflect library not active", NULL, MB_OK);
  if (icerror != IC_CONTEXTSTRING_NOT_FOUND)
    IcDefaultErrorProc(hWnd, hSession, NULL, icerror);
}
```

Don't save Contexts across messages

The component defining the status message always knows its own context, but the partner component must use IcGetContext and/or IcGetContextString each time it uses the context. The context returned by IcGetContext is only valid during the processing of the current message. Don't expect to retrieve the context once and save it globally. Contexts are assigned dynamically by the Manager as accessories register themselves and libraries are loaded. A context can become invalid if the defining accessory or library terminates and the Manager reassigns the context to someone else.

The PING sample follows this guideline in IcWinApp. Notice that IcWinApp always requests Reflect's context each time a status message is sent or received.

Use IC_MAKE_RESULT to build a library specific status messages

The IC_MAKE_RESULT macro is useful for building the status message from its three parts: context, status type and status value.

The following code fragment uses IC_MAKE_RESULT to make the library specific status. This status contains a context of the Reflect library and the REFLECT_STATUS_PING status type.

```
LPIC_STATUSBUF lpStatusBuf=NULL;
IC_RESULT icerror;
IC_RESULT_CONTEXT LibContext=IC_RESULT_CONTEXT_INVALID;

  icerror=IcGetContext(REFLECT_CONTEXTSTRING,&LibContext);
  /* add appropriate error handling */
  /* allocate and lock the extended status buffer */
  lpStatusBuf->icstatus=IC_MAKE_RESULT(LibContext,REFLECT_STATUS_PING,0);
```

Building an IC_STATUS_BUFFER extended status message

In the PING example, IcWinApp allocates a buffer large enough to contain the IC_STATUSBUF followed by the "PING" data. The icstatus field of the IC_STATUSBUF structure contains a library specific status with a status type of REFLECT_STATUS_PING. All fields of the IC_STATUSBUF structure are initialized. The "PING" null terminated string immediately follows the IC_STATUSBUF header.

The PING sample then uses IC_MAKE_RESULT to make the IC_STATUS_BUFFER extended status. This status contains a context of IC_RESULT_CONTEXT_STD, a type of IC_STATUS_BUFFER, and a value of the handle to the IC_STATUSBUF buffer. The following code fragment is from IcWinApp.c:

```
void PingReflect(HWND hWnd)
{
 LPIC_STATUSBUF lpStatusBuf=NULL;
 LPSTR lpData=NULL;
 IC_RESULT_CONTEXT LibContext=IC_RESULT_CONTEXT_INVALID;

 icerror=IcGetContext(REFLECT_CONTEXTSTRING,
          &LibContext);
 if IC_CHECK_RESULT_SEVERE(icerror) {
  MessageBox(hWnd,"Reflect library not active",
       QAPPNAME,MB_OK);
  if (icerror!=IC_CONTEXTSTRING_NOT_FOUND)
   HandleIcError(hWnd,s.hSession,NULL,icerror);
 }
 else {
  s.hStatusBuf=IcAllocBuffer(sizeof(IC_STATUSBUF)+256);
  assert(s.hStatusBuf);
  lpStatusBuf=(LPIC_STATUSBUF)IcLockBuffer(s.hStatusBuf);
  assert(lpStatusBuf);
  lpStatusBuf->icstatus=IC_MAKE_RESULT(LibContext,
                 REFLECT_STATUS_PING,
                 0);
  lpStatusBuf->icerror=IC_OK;
  lpStatusBuf->uBufSize=256;
  lpData=(LPSTR)(lpStatusBuf+1);
  lstrcpy(lpData,"Hello?");
  lpStatusBuf->uDataSize=lstrlen(lpData);
  IcUnlockBuffer(s.hStatusBuf);
  s.icstatus=IC_MAKE_RESULT(IC_RESULT_CONTEXT_STD,
            IC_STATUS_BUFFER,
            s.hStatusBuf);
  icerror=IcSetStatus(s.hSession,s.icstatus);
  if IC_CHECK_RESULT_SEVERE(icerror)
   HandleIcError(hWnd,s.hSession,NULL,icerror);
 }
}
```

Libraries send status messages with IcMgrSendEvent

IcMgrSendEvent expects two single-word parameters rather than a IC_RESULT 'long' value. Use the LOWORD and HIWORD macros to break down the 'long' value.

```
IC_RESULT icstatus
HIC_STATUSBUF hStatusBuf;

icstatus=IC_MAKE_RESULT(IC_RESULT_CONTEXT_STD,
          IC_STATUS_BUFFER,
          hStatusBuf);

IcMgrSendEvent(hLibSession,IC_STATUS,
      HIWORD(icstatus),LOWORD(icstatus));
```

Don't push library-specific statuses on down the library stack

After processing a library-specific status, your library should "swallow" the status by not calling IcMgrSetResult to continue passing the status down the stack. Then, if a status makes it all the way down the library stack and the application is returned an IC_StatusResult event containing an error, the application knows the library didn't properly process the status.

Defining a message protocol between the application and library

When global buffers are exchanged between the application and library, be sure to agree who's responsibility it is to free the buffer. Typically, the sender allocates the buffer and the receiver frees it, however, senders must allow for the error case when the message is not successfully processed by the receiver. In the PING sample, IcWinApp both allocates and frees the global buffer. Also notice how IcWinApp frees the buffer after detecting an error in IC_StatusResult processing.

System Timers

INFOConnect libraries that need timers must access them directly from Windows. INFOConnect does not currently provide any kind of framework or API for libraries to share an INFOConnect timer.

There are several techniques for allocating Windows system timers. Generally, the preferable way for INFOConnect libraries is the "no-WndProc" approach (the HWND parameter on the SetTimer call is set to NULL), because libraries generally don't have Window functions and message loops.

Sample code

The REFLECT sample library registers a timer routine using this technique. The library's local callback function, TimerRoutine, is registered during IcLibInstall processing and deregistered during IcLibTerminate processing.

```
#define TIMERRESOLUTION 2000
int nTimerID=0;

IC_RESULT FAR PASCAL IcLibInstall(IC_RESULT_CONTEXT context)
{
 nTimerID=SetTimer(NULL,0,TIMERRESOLUTION,(FARPROC)TimerRoutine);
 if (nTimerID == 0)
   return IcSetSessionError(NULL, LibContext,
                IC_ERROR_TIMERS,
                NULL,NULL,NULL);

 return IC_OK;
}

IC_RESULT FAR PASCAL IcLibTerminate(void)
{
 IC_RESULT icerror = IC_OK;

 if (nTimerID)
   KillTimer(NULL,nTimerID);
 return icerror;
}

WORD FAR PASCAL TimerRoutine(HWND hWnd,
              WORD wMsg,
              int nIDEvent,
              DWORD dwTime)
{
 /*
   This callback function must appear in the .DEF file.
 */
 ...
 }
  return (0);
}
```

Tracing

There are two functions, IcMgrTraceBuffer and IcMgrTraceResult, which allow libraries to write information to the trace.log debug file when tracing has been enabled for the session.

The IcMgrTraceResult function can be called by a library to append a trace entry containing an IC_RESULT in the trace log file. IcMgrTraceBuffer can be used by a library to add data or state information to the trace log. For example, a library may need to trace transmit and receive buffers when buffers appear to be "lost." Another useful possibility is to log pertinent session information after receiving an error result or an unexpected event.

IC_OK is returned for both procedures if the request is valid, even if debug was not enabled for the session and the information was not traced.

Sample code and Trace log entry

The following code sample demonstrating the usage of IcMgrTraceResult and IcMgrTraceBuffer is from IcStack2:

```
icerror = IcLd(pSession->hIcSessionBelow, which);
if IC_CHECK_RESULT_SEVERE(icerror) {
  /* Log the result and the session record in the trace log */
  IcMgrTraceResult(LibContext, pSession->hIcSession,
          IC_NULLEVENT,
          QMODULEID".IcLibLd: Error from IcLd call",
          icerror);
  IcMgrTraceBuffer(LibContext, pSession->hIcSession,
          IC_NULLEVENT,
          QMODULEID".IcLibLd: Session Record",
          pSession, sizeof(SESSIONREC));
```

The following trace log entries occurred when an INFOConnect closed while a session that included IcStack2 as an EIL was active:

```
Session:0003 ICSTACK2.IcLibLd: Severe Error from IcLd call IC_ERROR_BADPARAMETER
Session:0003 ICSTACK2.IcLibLd: Contents of Session Record IC_OK
 00 00 03 00 80 07 44 04 01 00 01 00 53 59 53 54  ......D.....SYST
 45 4D 58 5F 58 58 58 58 00 20 20 20              EMX_XXXX.
```

Running with Multiple Versions of INFOConnect

Service and external interface libraries compiled with the 2.0 release of the IDK will run with INFOConnect release 3.0 without recompiling. Libraries can be written to run with multiple versions of INFOConnect by following one or both of these guidelines.

Follow step 1 if the library sets the minimum and maximum levels of ICS to 2.0 in the resource file. Follow step 1 and 2 if the minimum level is 2.0 and the maximum level is 3.0 in the resource file. The INFOConnect RCDATA section of the library's resource file identifies the minimum ICS version/revision level and a maximum ICS version/revision level.

1. To compile a 2.0 library using the 3.0 IDK, continue to use the 2.0 Library API routines and IcLib... function prototypes. Stubs that call the new API routines have been provided in CommMgr.dll.
2. Determine the level of INFOConnect that the library is running with by using successive calls to IcInitIcs. Provide code that checks the version of INFOConnect prior to using any of the new 3.0 functionality. Typecast the parameters in the IcLib... functions that have changed between ICS 2.0 and 3.0. For Example, the first parameter of IcLibUpdateConfig and IcLibVerifyConfig has been changed from a HWND to a HIC_CONFIG.

Notes:

- *A library that runs with INFOConnect 2.0 and 3.0 must be linked using IcWin20.lib.*
- *If the library runs with INFOConnect 3.0 only, always use 3.0 calls and link with IcWin.lib.*
- *Section 10, "Converting from Previous Releases", provides detailed information on upgrading from Release 2.0 to 3.0. Be sure the updates have been completed prior to setting the maximum version in the library's resource file to reflect INFOConnect 3.0.*

The following code fragment can be added to IcLibInstall to determine the release level of INFOConnect that the library is running with:

```
int icversion; /* global variable */
IC_RESULT icerror;

if ((icerror=IcInitIcs(IC_VERSION_3_0,IC_REVISION_3_0))==IC_OK)
  icversion = IC_VERSION_3_0;
else
  icversion = IC_VERSION_2_0;
```

The recommended way to isolate new API calls in your library follows:

```
if (icversion >= IC_VERSION_3_0) {
  /* call some new 3.0 INFOConnect API */
}
else {
  /* alternative action using pre 3.0 API calls */
}
```

Library Checklist

To assure consistency in library behavior, this section gives general guidelines and covers some common situations and how libraries should respond to those situations. If libraries don't respond consistently, applications become library-dependent and one of the major thrusts of the INFOConnect program is defeated.

Assume applications will ignore status messages

Applications should generally be allowed to ignore a status message. Design your library so that it behaves reasonably if an application ignores a status message emitted by the library. This is especially true for library-specific statuses.

Compiling

Note: *Before compiling and linking, review the System Verification Checklist in Section 1, Installation.*

Memory Models

INFOConnect libraries must adhere to the requirements of Dynamic Link Libraries.

The Windows SDK documentation covers the use of the various memory models. The INFOConnect architecture places no additional constraints on memory model usage.

Include files

INFOConnect libraries include the following headers in their source files.

```
#include <windows.h>
#include <idlib.h>
#include <iqproto.h> /* optional */
```

Compiler options

The generic service library, SERVICE, included with the INFOConnect Development Kit uses the following options with the Microsoft C compiler.

```
cl  -c -ASw -Gsw -Os -Zdp service.c
```

-c	Compile only - don't link
-ASw	(S) near code pointers and near data pointers (w) SS != DS DS not loaded on function entry
-Gsw	(s) remove stack probes (w) compile for Windows
-Os	(s) favor code size (d) optional flag for debugging ***Note:*** *Using the (d) flag disables optimization and is recommended when using debuggers like CodeView. Some of the sample programs use the (d) flag in their make files. Don't forget to remove it when building the production version of your code.*
-Zp	(p) pack structures on 1 byte boundaries (i) optional flag for CodeView debugging ***Note:*** *Don't forget to remove the (i) flag when building the production version of your code.*

Resource Files

Windows applications and DLLs use resource files (.RC) to define things like menus, icons, stringtables, bitmaps, and so forth. In addition to these conventional resources, INFOConnect makes use of two less common types supported by the Resource Compiler: RCDATA and User-Defined resources.

There are five INFOConnect-related resources in the .RC file. Let's walk through several fragments from a resource file for the Intrface sample library and look at each of these INFOConnect sections.

Here is the beginning of Intrface.rc:

```
/************************************************************/
/* INTRFACE.RC                          */
/************************************************************/

#include <windows.h>
#include <icdef.h>
#include <icdict.h>
#include <icsample.hic>
#include "intrface.h"
```

IcDef.h and IcDict.h are INFOConnect header files containing general definitions needed in all INFOConnect resource files. IcSample.hic contains component number information. Intrface.h contains local definitions specific to this library. Definitions in Intrface.h will be specifically pointed out because you will have to provide a similar header file for your own resource file.

INFOConnect resource

The INFOConnect resource is a RCDATA resource that points to all other INFOConnect-related resources. It specifies whether the component is an application or library, the vendor of the component, and so forth. See data type IC_RC_NODE in the *IDK Programming Reference Manual* for a detailed description of this resource.

```
INFOConnect RCDATA
BEGIN
 IC_VERSION_3_0,
 IC_REVISION_3_0,
 IC_INTERFACE,    //IC_SERVICE vs IC_INTERFACE vs IC_ACCESSORY
 IC_HEADER_3_0,   //size of the INFOConnect RCDATA section
 IC_RC_DICTIONARY, //link to dictionary tables
 IC_LIBRARYID,    //STRINGTABLE link
 IC_LIBRARYDESC,  //STRINGTABLE link
 IC_VENDOR,       //STRINGTABLE link
 IC_MODULEID,     //STRINGTABLE link, Windows DLL name
 0,0,             //session and library flags
 CONFIGRCID,      //link to path template resource
 /* The following fields are new for 2.02 */
 IC_VERSION_3_0,
 IC_REVISION_3_0,
 0,               //reserved, must be zero
 0,               //reserved, must be zero
 0,0,             //no generic component value
 SAMPLE_INTRFACE, UIS_SAMPLE  //Supplier number for Intrface
END
```

IC_VERSION_2_0 and IC_REVISION_2_0 refer to the minimum (or oldest) level of Connectivity Services that the library requires for proper operation. Older levels of Connectivity Services will refuse to load the library.

IC_INTERFACE in the type field specifies that this component is an interface library.

IC_HEADER_3_0 designates the size of the header.

IC_RC_DICTIONARY is the link to dictionary tables. It is only needed by INFOConnect libraries that define configuration tables. A 0 is used if the library doesn't define any tables.

IC_LIBRARYID identifies the context string in the string table for your library. This is the same string that is defined in your .HIC file and used by other INFOConnect applications and libraries to detect library-specific statuses and errors generated by your library. Unlike a ModuleID, which must be a unique Windows DLL name, a LibraryID is only relevant within the domain of the INFOConnect Manager. LibraryIDs can be longer and more descriptive than ModuleIDs; they can be IC_MAXLIBRARYIDLEN characters long. The user sees LibraryIDs in various Manager status lists and windows.

IC_LIBRARYDESC identifies the default description in the string table. It is used when the library is installed.

IC_VENDOR identifies the vendor identification string in the string table.

IC_MODULEID identifies the string table entry of the Windows DLL name of your library. The string must match the library name in your library's module definition file (.DEF). This DLL name must not conflict with any other Windows DLL.

The next two fields are set to 0 if not relevant. Use IC_SF_SESSIONSTATUS for the session flag if the library can respond to the IC_CONNECT_STATUS status type. Use IC_LF_ERRORHELP for the library flag if the library has context sensitive help topics for every library-defined error value.

CONFIGRCID is only needed for libraries that define path templates. It is zero if no path templates are used by the library. See the *IDK Programmer's Reference Manual*, under IC_TemplateInit data type for an example of the format for CONFIGRCID.

IC_VERSION_3_0 and IC_REVISION_3_0 specify the maximum (or latest) level of Connectivity Services that the library was developed with; therefore, taking advantage of that level of ICS features.

Note: *Specify IC_VERSION_3_0 and IC_REVISION_3_0 only after all requirements in Chapter 10, "Converting from Release 2.0 to 3.0 have been completed. Specify IC_VERSION_2_0 and IC_REVISION_2_0 until the enhancements to the library have been completed.*

The next two fields are reserved fields and must be set to 0.

The next two entries are the LO, HI values of the IC_COMPONENT data type which defines the generic component number for the library. Components that specify a non-zero generic IC_COMPONENT perform a specific function and must conform to the interface defined by the specific component's .HIC file. Generic component numbers are assigned by the Malvern Development Group.

The last two entries are the LO, HI values of the IC_COMPONENT data type which defines the branded (supplier specific) component number for the library. The branded IC_COMPONENT uniquely identifies the component

Note: *If the branded component is defined as 0,0 a unique value will be assigned when the library ID is added to the INFOConnect configuration database. Refer to "Component Numbers" in Appendix A of the IDK Programming Reference Manual for more information on Component numbers.*

IC_RC_DICTIONARY

The dictionary resource is only required for libraries; it is an RCDATA resource that points to the various table definitions (path, channel, custom and invisible) for the library. See data types IC_DICT_NODE and IC_DICT_TABLE in the *IDK Programming Reference Manual* for a detailed description of this resource.

```
IC_RC_DICTIONARY RCDATA
BEGIN
 IC_DICTIONARY_COUNT, //number of tables
 IC_DICTIONARY_BASE, //link to first table
 IC_DICTIONARY_RCTYPE, //User-defined RC resource
 IC_DEFAULT_RCTYPE, //User-defined RC resource
 0,0,0,0,

 IC_TF_PATHTABLE,   //start of path table info
 ID_PATHTABLE,
 IC_PATHTABLE_SERIALNO,

 IC_TF_CHANNELTABLE, //start of channel table info
 ID_CHANNELTABLE
 IC_CHANNELTABLE_SERIALNO
END
```

IC_DICTIONARY_RCTYPE defines the *user-defined* resource used in your resource file to define configuration tables.

IC_DEFAULT_RCTYPE defines the *user-defined* resource used in your resource file to define default records for your path, channel and invisible tables.

The end of the IC_RC_DICTIONARY resource section contains a sequence of triplets, one triplet for each table defined in your library. These triplets are just a sort of table of contents for each table; the tables themselves are not completely defined here. The example above references two tables: a path table and channel table.

Table definitions

Table definitions are only required for libraries. They describe the configuration tables such as path, channel, custom and invisible tables for the library. Each table definition contains names, types and positions of the fields that make up the table. See data type IC_DICT_FIELD in the *IDK Programming Reference Manual* for a detailed description of this resource.

```
IC_PATHTABLE_BASENO IC_DICTIONARY_RCTYPE
BEGIN
  ID_PATHTABLE_DESC,IC_FF_NO_KEY,IC_FT_CHAR,0,160,
  0
END

IC_CHANNELTABLE_BASENO IC_DICTIONARY_RCTYPE
BEGIN
  ID_CHANNELTABLE_DESC,IC_FF_NO_KEY,IC_FT_CHAR,0,160
  0
END
```

The Intrface sample has very rudimentary tables. Both tables only have one field, a 20 character description field. Notice that no key fields are present. Keys are not defined in the table definitions for the path and channel tables; INFOConnect provides them by default. Invisible tables, however, do require that you define key fields.

Table defaults

Each table definition requires an accompanying default record. When the user adds a new path or new channel, the default data passed to your library will come from this section of the resource file. See data type IC_DICT_FIELD in the *IDK Programming Reference Manual* for a detailed description of this resource.

```
IC_PATHTABLE_BASENO IC_DEFAULT_RCTYPE
BEGIN
 "Path Description \0"
END

IC_CHANNELTABLE_BASENO IC_DEFAULT_RCTYPE
BEGIN
 "Channel Description\0"
END
```

CONFIGRCID

The configuration resource defines path templates. Path templates identify a group of libraries that function together. Any INFOConnect component can define templates, but they are most often used by External Interface Libraries. Service libraries that are always used with the same External Interface Library can also make use of templates. For more information on the CONFIGRCID RCDATA resource, see IC_TemplateInit in the *IDK Programming Reference Manual*.

This template shows the Service sample library stacked over the Intrface sample library:

```
CONFIGRCID RCDATA
BEGIN
 IC_TemplateInit
 IC_TemplateBegin     "Intrface"
 IC_TemplateDescription "Sample Interface Library"
 IC_TemplateLibrary   "Service"
 IC_TemplateLibrary   "Intrface"
 IC_TemplateEnd
 IC_TemplateTerm
END
```

STRINGTABLE

You should already have a STRINGTABLE in your resource file. INFOConnect requires the STRINGTABLE to contain some additional entries with text for things like the application or library name, the vendor name, error messages, and so forth.

```
STRINGTABLE
BEGIN
  IC_MODULEID,  QMODULEID
  IC_LIBRARYID, INTRFACE_CONTEXTSTRING
  IC_LIBRARYDESC,"Sample INFOConnect External Interface Library"
  IC_VENDOR,    QVENDOR
  /* Strings naming each field in the path and channel tables */
  ID_PATHTABLE,      "Path Configuration"
  ID_PATHTABLE_DESC,  "Path Description Field"
  ID_CHANNELTABLE,    "Channel Configuration"
  ID_CHANNELTABLE_DESC,"Channel Description Field"
END
```

Version information resource

Windows 3.1 provides version checking capabilities to installation programs in its file installation library, VER.DLL. The INFOConnect Installation Manager does version checking when installing files that have version information in their resource section. The sample programs and makefiles use the technique shown below. Version information is compiled into the resource file if the WINSDKVER environment variable is defined and if WINSDKVER indicates that the Windows 3.1 SDK is available. The VER.H header file is not available in the Windows 3.0 SDK. The constants used on the right side of the #define statements (for example, QMARKETINGNAME, QMODULEID) are generally defined in the .H file and are specifically tailored for your component.

```
#ifdef WINSDKVER
#if (WINSDKVER >= 0x030a)
/* VER.DLL is only available in the Windows 3.1 SDK */
#include <ver.h>
#define VER_FILETYPE        VFT_DLL
/*                 VFT_APP for applications */
/*                 VFT_DLL for libraries  */

#define VER_FILESUBTYPE     VFT_UNKNOWN
#define VER_FILEDESCRIPTION_STR QMARKETINGNAME
#define VER_INTERNALNAME_STR  QMODULEID
#define VER_FILEVERSION     NFILEVERSION
#define VER_PRODUCTVERSION   NPRODUCTVERSION
#define IC_FILEVERSION_STR   QVERSION
#define IC_PRODUCTVERSION_STR  QVERSION
#define IC_LEGALCOPYRIGHT_STR  QCOPYRIGHT
#define IC_LEGALTRADEMARKS_STR QTRADEMARK
#define IC_PRODUCTNAME_STR   QPRODUCTNAME
#define IC_COMPANYNAME_STR   QVENDOR
#include <icdef.rh>
#endif
#endif /* WINSDKVER */
```

INFOConnect resource file (.RC) layout

INFOConnect RCDATA

INFOConnect resource header

version information
vendor
application or library name

IC_RC_NODE
data type

IC_RC_DICTIONARY RCDATA

Library dictionary header

table name
table type
table serial number
links to table descriptions

IC_DICT_NODE &
IC_DICT_TABLE
data types

CONFIGRCID RCDATA

Template descriptions

IC_RC_NODE
data type

IC_DICT_FIELD
data type

Default path record

Path table description

field name
field type
field position

Default channel record

Channel table description

field name
field type
field position

Default invisible record

Invisible table description

field name
field type
field position

STRINGTABLE

String table entries

text for library name
text fo vendor name
text for table names
text for field names

Compiling the resource file

Resource files (.RC) for INFOConnect DLLs are compiled just like any other resource file. The generic interface library, Intrface, was built with the following statement:

```
rc -r intrface.rc
```

INFOConnect Header Files

An INFOConnect header file (.HIC) is supplied by each library so that the information can be used not only by the library's source files, but also as needed by other INFOConnect applications or libraries.

Each library must have a unique context string. Any INFOConnect component can call IcGetContext with the defined context string to obtain the components context.

A table number must be assigned for each library table definition declared in the library's resource file (.RC). The fields in each table must also be defined. These table numbers and their field numbers are needed when using Configuration API calls.

The tables are numbered sequentially, starting with one. The fields within each table are also numbered sequentially starting with one, including PathID and ChannelID. Adding comments about the key and field types provides useful information to developer's using the .HIC file.

Note: *If a field is obsolete, see Modifying Library Tables on page 7 - 10, comment out the field number, do not delete it and renumber the fields.*

Library specific errors and statuses are also defined in the INFOConnect header file. Refer to "Generating library-specific errors' on page 7 - 57 and "Communicating with applications with status messages' on page 7 - 63.

```
#define REFLECT_CONTEXTSTRING "REFLECT"
#define REFLECT_TEMPLATEID   "Reflect"

#define REFLECT_ID      1
#define REFLECT_VERSION   1

#define UIS_SAMPLE_REFLECT  0x00010001 /* ComponentNum == 1,1 */

/* Define Reflect's table numbers as 1 relative.
  Also define the table's field numbers.     */

#define REFLECT_PATH_TABLENUM    1
 /* The path and channel fields are defined here
   but managed by the backplane. */
#define  REFLECT_PATH_PATHID      1
#define  REFLECT_PATH_CHANNELID    2

#define REFLECT_CHANNEL_TABLENUM  2
#define  REFLECT_CHANNEL_CHANNELID  1
#define  REFLECT_CHANNEL_PREFIX    2 /* 40 char IC_FT_STRINGI */
#define  REFLECT_CHANNEL_ERRORFACTOR  3 /* IC_FT_INT field.
      Can force Xmt/Rcv errors.
        value=1 all Xmt/Rcvs are successful
        value=2 every other Xmt/Rcv is successful
        value=3 every third Xmt/Rcv is successful, etc. */
#define  REFLECT_CHANNEL_SUFFIX    4 /* 40 char IC_FT_STRINGI */

#define REFLECT_ADMIN_TABLENUM    3
#define  REFLECT_ADMIN_KEY       1 /* IC_FT_UNSIGNED */
       /* The following arbitrary value is used as the
        key when writing to the administrabive table. */
#define    IC_ADMINTABLE_KEY   0x1234
#define  REFLECT_ADMIN_PATHID      2 /* IC_FT_INT */


/* status types */
#define REFLECT_STATUS_PING 1

/* error types */
#define REFLECT_ERROR_SEVERE IC_ERROR_SEVERE

/* values and string numbers correspond */
#define REFLECT_ERROR_XMTERROR    1
#define REFLECT_ERROR_RCVERROR    2
```

Linking

INFOConnect libraries are linked like normal Windows DLLs. Service.dll, the Dynamic-Link Library for the generic service library skeleton included with the development kit was linked as follows:

```
link @service.lnk
rc service.res service.dll
```

The Resource Compiler is run after the linker to combine the .RES file created earlier with the new .DLL file.

Linker (LNK) files for service libraries

A typical LNK file contains:

```
/noe/nod/map/line service.obj+c:\windev\lib\libentry
service.dll
service.map
icwin Sdllcew libw
service.def
```

The /NOD option allows you to specify the Windows libraries explicitly. It tells the linker not to search any libraries specified in the object file to resolve external references.

The /NOE option allows you to override an object file built into the libraries with one of your own (like whook.obj). This option prevents the linker from searching the extended dictionary, which is an internal list of symbol locations that the linker maintains.

LIBENTRY.OBJ is provided with the Windows SDK and contains initialization code required for all Windows DLLs.

IcWin.lib resolves all references to INFOConnect functions. Use IcWin20.lib when compiling a library that has been coded to run with INFOConnect releases 2.0 and 3.0 .

SDLLCEW.LIB and LIBW.LIB are Windows libraries.

Refer to the Windows SDK documentation for detailed information about linking requirements of DLLs.

Module definition (.DEF) files for Service Libraries

A typical DEF file (for example, Service.def) is shown below. The EXPORTS section must include the IcLib... entries as shown. The ordinal numbers on the IcLib functions are important. Any functions that are called by Windows (for example, dialog box callback functions) must be listed after the IcLib functions. The IDK samples start the ordinal numbers at fifty for the functions called by Windows to allow easy insertion of new IcLib functions in future ICS releases.

```
;*********************************************/
;* SERVICE.DEF                              */
;*********************************************/

LIBRARY Service
DESCRIPTION 'Sample INFOConnect Service Library'
EXETYPE WINDOWS
STUB 'WINSTUB.EXE'
CODE MOVEABLE DISCARDABLE PRELOAD

;DLLs require DATA SINGLE because there is only one instance
DATA SINGLE PRELOAD MOVEABLE
HEAPSIZE 4096
EXPORTS
  IcLibUpdateConfig    @1
  IcLibCloseSession    @2
  IcLibEvent           @3
  IcLibGetString       @4
  IcLibIdentifySession @5
  IcLibInstall         @6
  IcLibLd              @7
  IcLibRcv             @8
  IcLibOpenSession     @9
  IcLibGetSessionInfo  @10
  IcLibSetResult       @11
  IcLibTerminate       @12
  IcLibXmt             @13
  IcLibVerifyConfig    @14
  IcLibPrintConfig     @15
  IcLibOpenChannel     @16
  IcLibCloseChannel    @17

;  Don't forget to include callback functions in this list
  cbChannelConfigDlg   @50
  cbPathConfigDlg      @51
  cbAboutDlg           @52

  WEP                  @53 RESIDENTNAME
```

Module definition (.DEF) files for External Interface Libraries

A typical DEF file (for example, Intrface.def) is shown below. The EXPORTS section must include the IcLib... entries as shown. The ordinal numbers on the IcLib functions are important. Any functions that are called by Windows (for example, dialog box callback functions) must be listed after the IcLib... entries.

```
;**************************************************/
;* INTRFACE.DEF                  */
;**************************************************/

LIBRARY Intrface
DESCRIPTION 'Sample INFOConnect External-Interface Library'
EXETYPE WINDOWS
STUB WINSTUB.EXE
CODE MOVEABLE DISCARDABLE PRELOAD

;DLLs require DATA SINGLE because there is only one instance
DATA SINGLE PRELOAD MOVEABLE
HEAPSIZE 4096
EXPORTS
   IcLibUpdateConfig    @1
   IcLibCloseSession    @2
   IcLibEvent        @3
   IcLibGetString     @4
   IcLibIdentifySession  @5
   IcLibInstall      @6
   IcLibLd        @7
   IcLibRcv        @8
   IcLibOpenSession    @9
   IcLibGetSessionInfo  @10
   IcLibSetResult     @11
   IcLibTerminate     @12
   IcLibXmt        @13
   IcLibVerifyConfig   @14
   IcLibPrintConfig    @15
   IcLibOpenChannel    @16
   IcLibCloseChannel   @17

;  Don't forget to include callback functions in this list

   cbChannelConfigDlg   @50
   cbPathConfigDlg    @51
   cbAboutDlg       @52

   WEP          @53 RESIDENTNAME
```

The IMPLIB utility

You do NOT have to run the IMPLIB utility and create an import library for your DLL. INFOConnect applications are not linked directly with your DLL; they are linked with IcWin.lib, an import library provided with the development kit.

PS2TTY - A Sample Service Library

PS2TTY scans receive buffers and removes any poll/select escape-sequences before passing the buffer on to the application. It is useful in connections to A Series hosts. PS2TTY is meant to be used with the simple TTY-like communications applications in the IDK like IcXvtApp, IcWinApp, or IcDosApp. PS2TTY cleans up the displayed output by removing MT-emulator commands that these sample programs are not built to recognize. PS2TTY requires no path or channel configuration; simply insert it in a path anywhere above the external interface.

What can I learn from PS2TTY?

PS2TTY is an example of a simple, filtering service library. The only code specific to this library is in function IcLibEvent in the IC_RCVDONE processing.

Source file descriptions

PS2TTY.c	C-language source
PS2TTY.h	Header file
PS2TTY.hic	Header file defining library-specific errors and statuses
PS2TTY.rc	Resource file
PS2TTY.dlg	Configuration dialog source statements
PS2TTY.hpj	Help project file for on-line help
PS2TTY.hh	Header file for on-line help
PS2TTY.doc	On-line help text in Microsoft Word format
PS2TTY.rtf	On-line help text in Rich Text Format
PS2TTY.hlp	On-line help text after Help Compiler processing
PS2TTY.def	Module-definition file used to link

All source files needed to build PS2TTY.dll and PS2TTY.hlp are provided with the IDK in the SAMPLE directory. To build the PS2TTY library, do:

```
nmake -f makelib LIBRARY=ps2tty
```

CoupleS - A Sample Service Library

Normally, statuses and errors are only 'seen' by the components in the current path: the application, the service libraries and the external interface. Status and error events are not sent across the connection by INFOConnect (except for accessories executing locally on the same workstation). CoupleS is a service library that extends the scope of statuses and errors by intercepting, encoding, and transmitting them across the connection. A partner CoupleS library on the other end of the connection decodes the status or error, generates the appropriate event, and passes it to the application.

CoupleS is normally used with a LAN connection (for example, XNS). The partner path at the other end of the connection must also use CoupleS. CoupleS intercepts statuses initiated by the application, then encodes and transmits them across the connection. The external interface below CoupleS never sees the status. Encoded status messages coming across the connection are decoded and passed up to the application as normal status messages. Status messages initiated by the external interface below CoupleS are intercepted and discarded.

What can I learn from CoupleS?

CoupleS actually does transmits and receives "behind the application's back." It also has a crude encoding/decoding capability to address the problem of transmitting across *Transparent=FALSE* sessions.

Source file descriptions

CoupleS.c	C-language source
CoupleS.h	Header file
CoupleS.hic	Header file defining library-specific errors and statuses
CoupleS.rc	Resource file
CoupleS.dlg	Configuration dialog source statements
CoupleS.hpj	Help project file for on-line help
CoupleS.hh	Header file for on-line help
CoupleS.doc	On-line help text in Microsoft Word format
CoupleS.rtf	On-line help text in Rich Text Format
CoupleS.hlp	On-line help text after Help Compiler processing
CoupleS.def	Module-definition file used to link

All source files needed to build CoupleS.dll and CoupleS.hlp are provided with the IDK in the SAMPLE directory. To build the CoupleS library, do:

```
nmake -f makelib LIBRARY=couples
```

Service - A Generic Service Library

Service can be used as the starting point for your own service library development. It contains all the required functions that a service library DLL must provide. If executed as is, Service will pass all data unaltered to and from the application.

Source file descriptions

File	Description
Service.c	C-language source
Service.h	Header file
Service.hic	Header file defining library-specific errors and statuses
Service.rc	Resource file
Service.dlg	Configuration dialog source statements
Service.hpj	Help project file for on-line help
Service.hh	Header file for on-line help
Service.doc	On-line help text in Microsoft Word format
Service.rtf	On-line help text in Rich Text Format
Service.hlp	On-line help text after Help Compiler processing
Service.def	Module-definition file used to link

All source files needed to build Service.dll and Service.hlp are provided with the development kit in the sample directory. To build the Service library, do:

```
nmake -f makelib LIBRARY=service
```

Reflect - A Sample External Interface

Reflect is a sample external interface that simply stores any transmitted messages from the application and returns them to the application when a receive is issued. Receive requests are queued if there is no stored message to return. As soon as a transmit is processed, pending receives are processed and passed to the application. Reflect can be built and executed without requiring an actual datacomm cable or specific data communications environment.

Additional Features of Reflect

Reflect has been coded to demonstrate the library upgrade feature. Define REFLECT01 to build Reflect with Release 2.0 configuration tables. That is, the table contains only the prefix and error factor fields as in Release 2.0. Define REFLECT02 to build Reflect with Release 3.0 configuration tables. That is, the table contains additional fields (suffix and a bit field) and the prefix field is obsoleted. Notice that during the upgrade, the data from the prefix field is salvaged and converted to the suffix.

Source file descriptions

Reflect.c	C-language source
Reflect.h	Header file
Reflect.hic	Header file defining library-specific errors and statuses
Reflect.rc	Resource file
Reflect.dlg	Configuration dialog source statements
Reflect.hpj	Help project file for on-line help
Reflect.hh	Header file for on-line help
Reflect.doc	On-line help text in Microsoft Word format
Reflect.rtf	On-line help text in Rich Text Format
Reflect.hlp	On-line help text after Help Compiler processing
Reflect.def	Module-definition file used to link

All source files needed to build Reflect.dll and Reflect.hlp are provided with the Development kit in the SAMPLE directory. To build the Reflect external interface, do:

```
nmake -f makelib LIBRARY=reflect
```

IcStack2- A Sample External Interface

IcStack2 is a sample external interface that can be used to stack one INFOConnect path on top of another. Although IcStack2 maps the INFOConnect API onto itself, it also serves as a model for mapping any other event-driven API on to the INFOConnect API

Source file descriptions

IcStack2.c	C-language source
IcStack2.h	Header file
IcStack2.hic	Header file defining library-specific errors and statuses
IcStack2.rc	Resource file
IcStack2.dlg	Configuration dialog source statements
IcStack2.hpj	Help project file for on-line help
IcStack2.hh	Header file for on-line help
IcStack2.doc	On-line help text in Microsoft Word format
IcStack2.rtf	On-line help text in Rich Text Format
IcStack2.hlp	On-line help text after Help Compiler processing
IcStack2.def	Module-definition file used to link

All source files needed to build IcStack2.dll and IcStack2.hlp are provided with the Development kit in the SAMPLE directory. To build the IcStack2 external interface, do:

```
nmake -f makelib LIBRARY=icstack2
```

Intrface - A Generic External Interface

Intrface can be used as the starting point for your own external interface development. It contains all the required functions that an external interface DLL must provide.

Intrface will compile successfully as is, however, if executed as is, it will never return IC_XMTDONE or IC_RCVDONE events to the application.

Source file descriptions

Intrface.c	C-language source
Intrface.h	Header file
Intrface.hic	Header file defining library-specific errors and statuses
Intrface.rc	Resource file
Intrface.dlg	Configuration dialog source statements
Intrface.hpj	Help project file for on-line help
Intrface.hh	Header file for on-line help
Intrface.doc	On-line help text in Microsoft Word format
Intrface.rtf	On-line help text in Rich Text Format
Intrface.hlp	On-line help text after Help Compiler processing
Intrface.def	Module-definition file used to link

All source files needed to build Intrface.dll and Intrface.hlp are provided with the development kit in the SAMPLE directory. To build the Intrface library, do:

```
nmake -f makelib LIBRARY=intrface
```

Section 8
Debugging

General Debugging Procedures

This section covers debugging techniques that work on all platforms.

Tracing INFOConnect Datacomm Activity

The INFOConnect trace facility traces datacomm traffic as it travels through the INFOConnect architecture and writes it to a trace file named Trace.log. Calls and events are displayed from the application's point of view. If multiple sessions are being traced, they are all captured in one trace log. All INFOConnect API calls are not recorded in Trace.log, only those involved in datacomm traffic. For example, calls to IcAllocBuffer or ic_buf_alloc are not captured in Trace.log.

What does the output from Trace look like?

Each line beginning with **Session:** indicates either a function call made by an application/ library or an event returned to the application. The INFOConnect/Windows names of functions and events are used. If selected, the contents of transmit and receive buffers are also displayed.

Sample Trace File

Below is an actual trace listing of a short session between an INFOConnect terminal accessory and a mainframe host. The trace shows the calls made by the accessory and the events returned by INFOConnect. The traced session consisted of a login attempt and the responses from the host.

```
Session:0000 LoadLibrary    C:\INFOCONN\ICAAPI16.DLL IC_OK
Session:0000 LoadLibrary    C:\INFOCONN\ICIPC16.DLL IC_OK
Session:5003 IcMgrOpenSession PINE-VALLEY IC_OK
Session:0000 LoadLibrary    C:\INFOCONN\TCPLWP.EIL IC_OK
Session:0000 ExitTCP.IcLibOpenSession Result IC_VERIFY_OK
Session:0000 LoadLibrary    C:\INFOCONN\ICTELNET.DLL IC_OK
Session:0000 ExitTELNET.IcLibOpenSession Result IC_VERIFY_OK
Session:2003 Evnt:IC_SESSIONESTABLISHED PINE-VALLEY:123.12.123.123 IC_OK
Session:4003 Evnt:IC_SESSIONESTABLISHED PINE-VALLEY:123.12.123.123 IC_OK
Session:6003 Evnt:IC_SESSIONESTABLISHED PINE-VALLEY:123.12.123.123 IC_OK
Session:2003 Entr:Trace.IcLibRcv     hBuff:11ee length(dec):511
Session:2003 Evnt:IC_STATUS          IC_CONNECT_OPEN
Session:2003 Entr:Trace.IcLibSetResult IC_REACTIVATE_OFF
Session:2003 Entr:Trace.IcLibSetResult IC_REACTIVATE_OFF
Session:2003 Evnt:IC_RCVDONE         hBuff:11ee length(dec):58

 0D 0A 0D 0A 55 4E 49 58 28 72 29 20 53 79 73 74  ....UNIX(r) Syst
 65 6D 20 56 20 52 65 6C 65 61 73 65 20 34 2E 30  em V Release 4.0
 20 28 70 69 6E 65 76 61 6C 79 29 0D 0A 0D 00 0D   (pinevaly).....
 0A 0D 00 6C 6F 67 69 6E 3A 20                    ...login:

Session:2003 Entr:Trace.IcLibRcv     hBuff:11ee length(dec):511
Session:2003 Entr:Trace.IcLibSetResult IC_REACTIVATE_ON
Session:2003 Entr:Trace.IcLibXmt     hBuff:2576 length(dec):7

 75 73 65 72 69 64 0D                 userid.

Session:2003 Evnt:IC_XMTDONE         hBuff:2576 length(dec):7
Session:2003 Evnt:IC_RCVDONE         hBuff:11ee length(dec):9

 50 61 73 73 77 6F 72 64 3A           Password:

Session:2003 Entr:Trace.IcLibRcv     hBuff:11ee length(dec):511
Session:2003 Entr:Trace.IcLibXmt     hBuff:2576 length(dec):13

 75 73 65 72 70 61 73 73 77 6F 72 64 0D     userpassword.

Session:2003 Evnt:IC_XMTDONE         hBuff:2576 length(dec):13
Session:2003 Evnt:IC_RCVDONE         hBuff:11ee length(dec):2

 0D 0A                                ..

Session:2003 Entr:Trace.IcLibRcv     hBuff:11ee length(dec):511
Session:2003 Evnt:IC_RCVDONE         hBuff:11ee length(dec):24

 4C 6F 67 69 6E 20 69 6E 63 6F 72 72 65 63 74 0D  Login incorrect.
 0A 6C 6F 67 69 6E 3A 20                          .login:
```

```
Session:2003 Entr:Trace.IcLibRcv     hBuff:11ee length(dec):511
Session:2003 Entr:Trace.IcLibSetResult IC_REACTIVATE_OFF
Session:2003 Entr:Trace.IcLibSetResult IC_REACTIVATE_ON
Session:2003 Entr:Trace.IcLibSetResult IC_REACTIVATE_OFF
Session:2003 Entr:Trace.IcLibSetResult IC_REACTIVATE_OFF
Session:2003 Entr:Trace.IcLibLd     LCL_CLOSESESSION|LCL_RCVXMT
Session:2003 Evnt:IC_SESSIONCLOSED
Session:4003 Evnt:IC_SESSIONCLOSED
Session:6003 Evnt:IC_SESSIONCLOSED
Session:6003 IcMgrCloseSession PINE-VALLEY IC_OK
Session:2003 Entr:Trace.IcLibSetResult IC_REACTIVATE_ON
Session:4003 IcMgrCloseSession PINE-VALLEY IC_OK
Session:0003          TimerDestroy Session IC_OK
Session:0000 FreeLibrary   AAPI16 IC_OK
Session:0000 FreeLibrary   IPC16 IC_OK
Session:0000 FreeLibrary   TCP IC_OK
Session:0000 FreeLibrary   TELNET IC_OK
```

How do I activate Trace?

Activating the INFOConnect Trace facility is done through the Administer menu. You must be logged on as the Administrator.

The generated file is named Trace.log and is written to your DataDir directory which is normally c:\windows. New trace sessions are appended to Trace.log, so you may need to occasionally erase or prune Trace.log.

The Path and Path Template configuration dialogs also provide a *Trace* check box that affects the Trace facility. See INFOConnect's on-line help for more information about using Trace.

Adding Trace Information to the Trace Log

There are two functions, IcMgrTraceBuffer and IcMgrTraceResult, which allow applications and libraries to write information to the Trace.log file when the trace log is active.

The IcMgrTraceResult function can be called by to append a trace entry containing an IC_RESULT in the trace log file.

IcMgrTraceBuffer can be used to add data or state information to the Trace.log file. For example, an application or library may need to trace transmit and receive buffers when buffers appear to be "lost." Another useful possibility is to log pertinent session information after receiving an error result or an unexpected event.

IC_OK is returned for both procedures if the request is valid, even if the trace log is inactive and the information was not traced.

Note: *The "Show Xmt/Rcv buffer contents" option in the Trace Log Options window must be set in order to display the buffer contents in Trace.log when using IcMgrTraceBuffer.*

Sample code and Trace log entry

The following code sample demonstrating the usage of IcMgrTraceResult and IcMgrTraceBuffer is from IcStack2:

```
icerror = IcLd(pSession->hIcSessionBelow, which);
if IC_CHECK_RESULT_SEVERE(icerror) {
  /* Log the result and the session record in the trace log */
  IcMgrTraceResult(LibContext, pSession->hIcSession,
          IC_NULLEVENT,
          QMODULEID".IcLibLd: Error from IcLd call",
          icerror);
  IcMgrTraceBuffer(LibContext, pSession->hIcSession,
          IC_NULLEVENT,
          QMODULEID".IcLibLd: Session Record",
          pSession, sizeof(SESSIONREC));
```

The following trace log entries occurred when an INFOConnect closed while a session that included IcStack2 as an EIL was active:

```
Session:0003 ICSTACK2.IcLibLd: Severe Error from IcLd call IC_ERROR_BADPARAMETER
Session:0003 ICSTACK2.IcLibLd: Contents of Session Record IC_OK
  00 00 03 00 80 07 44 04 01 00 01 00 53 59 53 54  ......D.....SYST
  45 4D 58 5F 58 58 58 58 00 20 20 20       EMX_XXXX.
```

Using the Diagnostic Library

The diagnostic library can be used to detect and log interoperability problems between applications and libraries in the Trace.log file. When trace logging is enabled, the Trace.log will contain entries that can be analyzed so that corrections in the library can be implemented. The diagnostic library can also be set up to display a dialog box for each interoperability problem that is uncovered. Unit test your application and/or library with the diagnostic library for a period of time in order to detect interoperability problems.

The following interoperability problems are detected:

- Premature transmit and receive requests initiated by the application.
- Validation of the buffer handles in IC_XMTDONE and IC_RCVDONE events.
- Detection of premature events before the IC_SESSIONESTABLISHED event.
- Detection of informational or warning errors in IC_RCVERROR and IC_XMTERROR messages.

See the INFOConnect on-line help for details on using the Diagnostic Library and the interoperability problems it detects (and corrects as needed).

Using the Assert Macro in Applications

The assert macro is used liberally in the IDK sample applications for many error conditions. XVT and Windows both support it. The following code fragment uses assert to verify that IcAllocBuffer returned valid buffer handles:

```
/* Allocate INFOConnect buffers. */
s.uBufsize=min((unsigned)sinfo.max_size,MAXBUFSIZE);
s.hXmtBuf=IcAllocBuffer(s.uBufsize);
s.hRcvBuf=IcAllocBuffer(s.uBufsize);
assert(s.hXmtBuf!=NULL);
assert(s.hRcvBuf!=NULL);
```

When this code is executed, if the conditional expression in parentheses is FALSE, assert prints a dialog box with a message of the form

    Assertion failed: (s.hXmtBuf!=NULL) at line 1234

Assert is usually supported in any standard C runtime library. When it is time to compile the production version of your code, a compile time flag disables code generation of assert statements or you can replace the assertions with more

formalized error messages. If you are not familiar with the assert macro, you should look at it; it's a very useful debugging tool.

Note: *The standard Assert macro cannot be used in Windows DLL libraries. The IDK provides a version of assert that does work with DLLs. It is defined in <IcAssert.h>*

Using the INFOConnect -d Debug Option

INFOConnect can be started in debug mode using the -d option.

```
c:\infoconn\infoconn -d
```

This sets a flag that can be tested by applications and libraries using the IcIsDebug function call. In debug mode, INFOConnect displays all error types returned to the default error procedure (IcDefaultErrorProc or ic_default_error_proc) rather than just some of the types.

IcIsDebug is needed by libraries because libraries don't normally have an execution-time user-interface. The only way to activate debug code in a library would be to recompile the library or add a button to the configuration dialog box.

The INFOConnect trace facility uses debug mode to choose between two methods of writing to the trace file. In normal mode, the trace file is kept open until IcLibTerminate. In debug mode, the trace file is flushed after every write. This allows the trace file to be viewed at any time and provides a complete trace file even if the system crashes, but affect performance.

Windows 3.x Debugging Procedures

This section covers debugging techniques unique to the Windows 3.x platform.

Windows 3.1 Issues

Special steps must be taken to develop Windows applications with the Windows 3.1 SDK if the application is to also run with Windows 3.0. See the Microsoft documentation.

Source Level Debugging

All of the major Windows-capable compilers provide a source level debugger including Microsoft's CodeView for Windows and Borland's Turbo Debugger for Windows.

Note: *If you have an old compiler that doesn't include a source level debugger, upgrade now! Early versions of some debuggers required a secondary monochrome monitor and display adapter card, but current versions can run on single monitor systems.*

Using Debug version of Windows

Microsoft strongly recommends that developers test with the debug system binaries. Various types of error checking are done and any errors or warning messages are sent to the debug terminal. If you aren't running with a debugger or debug terminal, you can run the DBWIN application (provided on the Windows SDK) to see messages produced by the debug system. The SDK installation process sets up two batch files, D2N.bat and N2D.bat, that switch between the normal and debug versions.

Win.ini debug options

The Windows 3.1 debug kernel has additional facilities activated through the Win.ini file.

To force termination after invalid parameters are detected

[Kernel]
ErrorOptions=1

To detect overwrites to buffers passed into the Windows APIs

[Windows]
ILoveBear=0

Common Coding Mistakes

Applications cannot use a pending session handle

Don't use the session handle returned by IcOpenSession (or ic_open_session) until the IC_SessionEstablished message (or E_IC_SESSION_EST event) has been returned. See the discussion in "Opening a Session."

Running with wrong or mixed versions of ICS

Multiple versions of INFOConnect can exist on the same machine, but extra precautions must be taken. The installation directory of some of the INFOConnect DLLs changed between INFOConnect releases. Also, a number of INFOConnect DLLs have been added.

If you have installed INFOConnect 3.0, but execute an application from a directory containing older INFOConnect DLLs, you can encounter problems. Problems may also occur if the PATH environment variable specifies an INFOConnect directory that contains older INFOConnect DLLs.

Wrong copy of <string.h>

Both Windows and Microsoft C provide a <string.h> header file. This can cause UNRECOVERABLE APPLICATION ERRORS at execution time unless the \WINDEV\INCLUDE directory has been specified before the \C\INCLUDE directory in the INCLUDE environment variable.

Fix the INCLUDE variable as discussed in the System Verification Checklist in Section 1, "Installation."

Insufficient stack size

If your code behaves differently from one execution to the next, you may need to increase the stack size in the Module Definition file (.DEF). Windows automatically increases the heap as needed, but not the stack. An insufficient stack can lead to memory corruption during execution.

Inadvertently freeing INFOConnect buffers more than once

Be careful to only free INFOConnect buffers once. To avoid problems, it's a good idea to set released buffer handles to NULL after calling IcFreeBuffer (ic_buf_free). This is especially important in functions that could be executed several times during some shutdown sequences. Windows can crash if the same buffer handle is released twice.

The following code fragment demonstrates freeing INFOConnect buffers:

```
void TerminateApplication(HWND hWnd)
{
  ...

  if(s.hXmtBuf!=NULL){
    IcFreeBuffer(s.hXmtBuf);
    s.hXmtBuf=NULL;
  }
  if(s.hRcvBuf!=NULL){
    IcFreeBuffer(s.hRcvBuf);
    s.hRcvBuf=NULL;
  }
}
```

Using trace entries as a debugging tool

Adding trace entries to the trace log using IcMgrTraceResult and IcMgrTraceBuffer is a useful way of debugging code. It is also a convenient way to support the production version of your code by adding log entries in code segments that should never be reached or to log error results. See "Adding Trace Information to the Trace Log" in this section for details on using IcMgrTraceResult and IcMgrTraceBuffer.

Using message boxes as a debugging tool

Interspersing message boxes throughout your code is a simple but useful debugging technique, especially if you aren't able to run a source code debugger like CodeView for Windows. However, be aware that message boxes can alter the timing of code execution by yielding control to Windows at a time that ordinarily would not happen. Service library and External Interface developers must be especially wary of this.

Note: *It may be better to use IcMgrTrace... calls so as not to alter the timing of code execution.*

Version verification errors on resource files

Missing resources can result in version errors from INFOConnect. You can inadvertently "lose" resources because of the way the Resource Compiler processes #include directives. If you use #include directives, be aware that the Resource Compiler only processes preprocessor directives from .H or .C files. This behavior is documented in the Windows SDK literature.

Section 9
Packaging INFOConnect Components

Overview

What is the purpose of the INFOConnect Installation Manager?

The INFOConnect Installation Manager is a general purpose Windows installation program that provides a common look and feel for all INFOConnect-based workstation software. The Installation Manager runs as a Windows application and provides the following basic capabilities:

- Installation (copying files from distribution media)
- Deinstallation (removing files from the workstation)
- Decompression of compressed distribution media
- File version checking
- Windows Program group modification
- Windows Font and bitmap installation
- Registration of INFOConnect accessories and libraries

Developers can also write code that is called during installation to perform further customization like:

- Updating Win.ini, System.ini, Config.sys and AutoExec.bat
- Configuring INFOConnect paths

How do I control the installation process?

There are three ways you control Installation Manager. You must write an IcSetup.inf script file and you can optionally provide Exit Hook and Quick Configuration libraries to further customize the installation process.

Terminology

The following terminology is specific to INFOConnect installation and packages:

Package

Users order and install INFOConnect packages based on their specific application and connectivity needs. For maximum flexibility, INFOConnect solutions are broken into different packages which are installed on the workstation in different combinations and "plugged" into the INFOConnect architecture.

A package is a set of files transferred from distribution disks to the workstation. Information about each installed package is saved in the installation database, InstMgr.cfg. This information is used to deinstall a package later, if requested by the user. A package can span more than one disk.

Installation Manager

The Installation Manager controls the installation and deinstallation of packages. The Installation Manager allows INFOConnect accessories and libraries to be registered in the INFOConnect configuration database, INFOConn.cfg. A series of packages can be installed in any order by the user and the Installation Manager insures that they are processed properly.

Install Shell

The Install Shell (Install.exe) bootstraps the installation process. When a user starts an installation from the Windows Program Manager Run command, an Install Shell is invoked which does a minimal amount of work before transferring control to the Installation Manager.

Standalone Installation (Install)

An environment in which INFOConnect packages are installed for non-shared use by an individual workstation. All of the package files are copied to a workstation destination. The workstation operator is the administrator for all packages on that workstation. (Install)

Publish Installation (Install /A)

An environment in which an administrator installs INFOConnect packages for shared use by multiple workstations. All of the package files are copied to a shared destination which is typically on a file server. The administrator has single point control over package availability and package configuration. The individual workstation operator must subscribe to a published package in order to enable use of the shared files.

Subscribe Installation (Install /N)

An environment in which published INFOConnect packages are enabled for use by an individual workstation. Only a subset of the package files are copied to a workstation destination. The workstation configuration is influenced by the published package installation and configuration.

Complete Deinstallation (Install /U)

To completely remove all INFOConnect packages, including the Connectivity Services packages, from the workstation. This includes package files, package configuration and system modifications.

INFOConnect Packages window

The INFOConnect Packages window can be opened by selecting the INFOConnect Manager Install Menu and then selecting Packages. A list of all installed packages and a brief description is displayed.

The Installation Manager is executed by selecting the Add button to install or replace a package, the Delete button to deinstall a package or the Examine button to display package information.

Quick Configuration is initiated for an individual package by first selecting the package and then selecting the Configure button. Quick Configuration is initiated for all packages by selecting the Config All button.

INF files

INF files are text files that describe your INFOConnect package to the Installation Manager. They look very similar to Windows INI files. The default INF file on a package disk is named IcSetup.inf.

CodeDir

After the INFOConnect Connectivity Services is installed, Win.ini contains an [INFOConnect] section with several statements. The CodeDir statement defines the directory containing the standard INFOConnect accessories and libraries. The default value for CodeDir is c:\infoconn.

DataDir

After the INFOConnect Connectivity Services is installed, Win.ini contains an [INFOConnect] section with several statements. The DataDir statement defines the directory containing the configuration and installation databases. Also, accessories typically save options files in the DataDir directory. The default value for DataDir is c:\windows.

Installation Database

The INFOConnect installation database, InstMgr.cfg, is comprised of INFOConnect package information. The Installation Manager maintains the information in the installation database.

Configuration Database

The INFOConnect configuration database, INFOConn.cfg, is comprised of INFOConnect component and configuration information. The Configuration Manager maintains the information in the configuration database.

Registration

INFOConnect accessories and libraries are required to have special information about themselves in their resource files. During registration, this information is transferred to the Configuration Manager and recorded in the configuration database.

Quick Configuration Manager

The Quick Configuration (Quick Config) Manager controls the sequence of package Quick Configuration. The Quick Config Manager calls the Quick Configuration libraries after package installation and when selected in the INFOConnect Packages window.

Quick Configuration Library

Each INFOConnect package can optionally contain one Quick Configuration (Quick Config) library. A Quick Config library is a Windows DLL (Dynamic Link Library) that is called after package installation and before package deinstallation. The intent of Quick Configuration after package installation is to get the package into a turnkey state with a minimum amount of user input. The intent of Quick Configuration before package deinstallation is to allow the package to participate in package removal.

Sequence Number

Packages can optionally assign sequence numbers. When a series of packages is installed at once, the sequence number determines the order that the quick config libraries are processed by the Quick Configuration Manager.

Exit Hook Library

An Exit Hook library is a Windows DLL (Dynamic Link Library) that is called at different points during the installation and deinstallation of a package. Exit hooks are used to modify or augment the installation process.

Writing a .INF Script File

This section describes how to write an installation script file (IcWinApp.inf) that:

- Installs IcWinApp.exe and registers it as an accessory in the configuration database.
- Displays IcWinApp.txt at the end of the installation as a readme file.
- Builds an icon for the IcWinapp executable file in the Windows INFOConnect group.
- Installs the IcWinApp.c, IcWinApp.h, and IcWinApp.rc source files in a subdirectory called src, for additional demonstration purposes.

Note: *For a "real" product, you may only want to install the* ***.EXE*** *and* ***.TXT*** *files.*

To build IcWinApp with Microsoft C 7.0, do the following from the DOS prompt:

```
>cd\idk\sample
>nmake -f makefile PROGRAM=icwinapp
```

To execute this installation script (IcWinApp.inf), run the INFOConnect Manager and select the Add button in the INFOConnect Package window. Enter the fully qualified script file name (c:\idk\sample\icwinapp.inf) when prompted for a package.

Sample .INF file

Here are the contents of IcWinApp.inf, the script file that controls the installation of IcWinApp:

```
;************************************************************
;* ICWINAPP.INF Installation Script File          *
;*                             *
;*      Sample INFOConnect Windows application  *
;*                             *
;************************************************************

[dialog]
  caption   = "INFOConnect IcWinApp Accessory"

[package]
;'name' cannot contain blank characters
  name     = "IcWinApp"
  description = "Sample Accessory"
  version=3,0,0,0
  lowicver=2,0,0,0
  highicver=3,0,0,0

[data]
  defdir = c:\icwinapp
  codedir = ignore
  format = VER

[disks]
  1 = ., "INFOConnect IcWinApp Disk 1"

[needed.space]
  minspace = 120000

[app.copy.appstuff]
  #acc,  0:,   Accessory
  #readme, 0:,   Readme
  #source, 0:src

[acc]
  1:icwinapp.exe, icwinapp.exe, "IcWinApp Sample Accessory"

[readme]
  1:icwinapp.txt, icwinapp.txt, "Readme file"

[source]
  1:icwinapp.c,  icwinapp.c,  "Source files"
  1:icwinapp.h,  icwinapp.h
  1:icwinapp.rc, icwinapp.rc

[progman.groups]
  INFOConnect,   infoconn.grp

[INFOConnect]
  IcWinApp,    icwinapp.exe
```

The **dialog** section provides a string to be displayed in the title bar during package installation. The **package** section provides a package name. It must not contain blank characters. The **format = VER** statement is required. It specifies the syntax being used in the .INF script file and that the Installation Manager will use Windows Ver.dll to check version information. The **disks** section provides disk names that are on the distribution disk labels. The **needed.space** section contains a value expressed in bytes. The user is prompted if the workstation has insufficient room to hold the package.

The **app.copy.appstuff** section begins to describe the files that make up the package. All of the files in the package are divided into several groups and this section serves as the index into those groups. Each group is defined by a single statement in the **app.copy.appstuff** section. Each statement specifies where all files in that group are to be copied. Each statement also specifies any special processing unique to that group of files. In the sample, the **Accessory** label on the **#acc** statement specifies that files in the acc section, in this case just IcWinApp.exe, are to be registered in the configuration database as accessories. The **#readme** statement specifies that **IcWinApp.txt** is to be displayed as the package ReadMe file at the end of the installation. The #**source** section has no special processing requirements. All files in this section are simply copied to the workstation in a subdirectory called src (c:\infoconn\src).

The **acc**, **readme** and **source** sections were pointed to by the #acc, #readme and #source statements in the app.copy.appstuff section. Each file to be installed requires a single statement containing the input file name, destination file name, and comment string to be displayed as that file is installed. Remember that common characteristics applying to all files in the section, like the destination directory, are specified in the app.copy.appstuff section.

Notice that only one comment string is specified in the source section. The "source files" comment string will be displayed for all three source files during installation. Another alternative is to specify a different comment string for each individual file to be displayed during installation.

The **progman.groups** section serves as the index into the program groups that are to be created or updated by this package. In the sample, the **INFOConnect** program group is to be updated with an icon labeled "IcWinApp" that executes IcWinApp.exe.

Creating the Package Diskette(s)

Root directory contents

You control and specify the directory structure of your distribution diskette in your INF script file. However, several files must be in the root directory of the first diskette in your package.

IcSetup.inf

A package must have an IcSetup.inf script file in the root directory.

Install.exe

A copy of the Install Shell (Install.exe)must exist in the root directory. The Install Shell bootstraps the installation process. You can get this program from the root directory of the IDK package itself.

File Compression

Files can optionally be compressed on the distribution diskette using Compress.exe, the Microsoft compression utility. Compress is not part of the IDK, but is part of the *Microsoft Windows 3.x Software Development Kit*. The naming convention for compressed files is to replace the last letter of the file extension with an underscore ("_").

A Closer Look at Processing Flow

Installation Flow

The processing flow of a package installation depends in a large measure on what sections are present in the INF file and what options are specified there. This section describes a full featured installation using all sections in the INF file. If some of those sections have not been specified, the corresponding installation step is skipped.

Getting Started

The Install.exe program is run from the Windows Program Manager Run command. For a floppy installation from drive A:, the user would enter a:\install. Install.exe then reads the IcSetup.inf file.

A package installation may also be started by selecting the Add button in the INFOConnect Packages window. When prompted for the package root directory, enter the package root directory (for example, a:\ to install from a:\IcSetup.inf) or enter a fully qualified script file name (for example, c:\idk\sample\icwinapp.inf).

Release Notes

If a **release.notes** statement is present in the **package** section of the INF, a dialog box is presented with the specified file at the start of installation. This provides you with a method for displaying text or instructions before installation gets underway.

Buttons are presented, giving the user the options of continuing with the installation or quitting. Scroll bars are displayed on the right side of the box allowing the user to scroll through the document if it is longer than one screen.

Destination

A dialog box is now presented which asks the user for verification of the destination directory.

This dialog box has a caption specified by the **caption** statement of the **dialog** section of the INF.

The user is presented with a default destination. This default destination is taken from the **defdir** statement of the **data** section in the INF. If there is no **defdir** statement, the current value of **Codedir** in the Win.ini [INFOConnect] section is used, if there is one from a previous INFOConnect product or installation. If there is no **Codedir** statement in Win.ini, the default destination is C:\INFOConn. This is an editable dialog box; the user can modify the destination.

The user's chosen destination is then checked against the value in Win.ini. If they are different when installing INFOConnect Connectivity Services, a dialog box is presented which states:

> "INFOConnect section of WIN.INI mismatched. Overwrite Codedir=<original destination> with CodeDir=<user specified destination>?

The user may select either the Yes or No button. If Yes is selected, the value of **Codedir** is changed in Win.ini.

Bootstrap

When the user starts Install.exe from the Windows Program Manager Run command, what is actually invoked is an Install Shell which does a minimal amount of work before the Installation Manager takes over to do the main job of the install. The bootstrap step transfers control from the Install Shell to the Installation Manager. If the user has executed the Installation Manager from the INFOConnect Manager Install menu, the bootstrap step is not necessary and is not done.

The Install Shell looks in Win.ini for an [INFOConnect] section and gets the **CodeDir** value out of that section. The Install Shell then tries to run the Installation Manager (InstMgr.exe) out of the **CodeDir** directory. If this fails, the user sees a diagnostic message in a dialog box. IcSetup.inf is then passed from the Install Shell to the Install Manager.

Package Check

The Installation Manager checks to see if the package the user is copying already exists. If it is, the user is prompted with a dialog box which states:

"The package **name** is already installed. Do you want to reinstall the package?"

The package **name** is that which is specified in the **package** section of the INF.

The user may select either the Yes or No buttons. If Yes is chosen, the Installation Manager first removes the old package and then the installs the new package. If No is chosen, the installation is discontinued.

App.Copy.Noremove Section

Sections specified in the **app.copy.noremove** section are now processed. Files copied by these sections are not removed during package deletion, except during a complete deinstall (install /U). This section can be used for copy protection files, files shared across packages, or configuration information which you would not want to have removed if the package was deleted.

App.Copy.Appstuff Section

The **app.copy.appstuff** section is processed next. This section specifies other sections which contain the files to be copied to the destination. Sections can be of a special type, indicating that the files require particular actions besides copying.

Copying

The main work of the installation is file copying. This is driven by the **app.copy.appstuff** section which details what files are to be copied and any special action that is taken for those files. This section is processed during a standalone, local, and publish installation.

The user is presented with a progress indicator during file copying. If multiple disks are required, the user is prompted to insert the appropriate disk.

Miscellaneous message boxes are displayed during file copying to report error conditions or solicit user response.

If this is a publish installation (install /A), the **app.copy.publish** and the **app.copy.subscribe** sections are also processed. The **app.copy.publish** details what files are to be copied and any special action that is taken for those files. The **app.copy.subscribe** section actions are interpreted during a publish and will be executed when a user subscribes to the package.

The **app.copy.standalone** section can be used to add special instructions which will not be processed during a publish or subscribe installation.

Program Group

The user is asked for verification of the destination program group and is presented with a default program group. This is an optional and editable dialog box. It is controlled by the **progman.groups** section in the .INF script.

During a publish installation, the **publish.progman.groups** is processed first and then the **progman.groups** section. The **progman.groups** section actions are interpreted during a publish and will be executed when a user subscribes to the package.

In addition, there is an **standalone.progman.groups** section which will not be processed during a publish installation.

Installation Complete

When all files have been copied for the package being installed, the user is presented with a dialog box saying "Package installation complete." At the top of this box are optional control buttons: Help, ReadMe, and Modify. These buttons are present in this dialog box only if the corresponding sections are present in the INF file.

The **help** type section copies the specified files to the destination directory. In addition, the user is presented with a HELP button in the Installation Complete dialog box which allows the user to enter directly into the last help file copied at the specified topic.

The **readme** type section copies the specified files to the destination directory. In addition, the user is presented with a README button in the Installation Complete dialog box. If this button is chosen, NOTEPAD is launched with the readme text in the last file copied for viewing.

The optional modify button is provided in the Installation Complete dialog box if the **driver** section type has been specified in the INF. The modify button launches the Notepad with Config.sys. This enables the user to edit the Config.sys file to include the drivers copied during the installation.

At the bottom of this dialog box is the question

"Do you want to continue with the installation of another package?"

This question is accompanied by Yes and No buttons.

Yes (Installing another package)

Selection of the Yes button starts a subsequent installation of a different package.

The user is presented with a dialog box which prompts the user to insert a new disk. After the user OK's his selection, the Installation Manager restarts the installation process at the Release Notes step described previously. (However, since InstMgr.exe is already installed and running at this point, the Bootstrap step is skipped).

No (No more to install)

Selection of the No button causes the Installation Manager to continue through the steps of quick configuration, if specified.

Quick Config

The Installation Manager now turns control over to the Quick Config Manager. The Quick Config Manager sorts all of the installed packages based on the sequence numbers and calls the quick config libraries. Installation terminates at this point.

Deinstallation Flow

Package deinstallation occurs in one of two ways:

1. An explicit deinstall occurs when a user selects a package in the INFOConnect Packages window and clicks the Delete button.
2. An implied deinstall occurs when a user clicks on the Add button in the INFOConnect Packages window and specifies a package that is already installed. The Installation Manager prompts the user with "Do you want to reinstall the package?'. If the user clicks on the Yes button, the package is deinstalled and then installed.

Deinstalling a package will only undo the installation actions. That is, files which have been copied are removed (unless they are specified in the INF as special files that should not be removed during deinstallation) and the entry is removed from the installation database.

Note: *During a complete deinstall (install /U) all files are removed, even if the files are specified in the INF file as special files that should not be removed.*

The Installation Manager first checks to see if any package files are in use. If a file is in use, the file name is displayed to the user. The user has the option to stop using the specified file and then click the OK button. This allows the package deinstallation to continue. The other option is to just click the Cancel button which stops the package deinstallation.

Deinstallation calls quick config so that it can undo the quick config action. Additional database entries, for example path templates or channels, are not removed during deinstallation.

The next phase of deinstallation is the deregistration of package accessories and libraries in the INFOConnect configuration data base and package fonts with Windows.

All package icons and files are then deleted to complete the package deinstallation.

INF SYNTAX

The format of the .INF script file is similar to that of a Windows initialization (.INI) file:

```
[section]
 entry=value
```

The .INF script file uses additional section syntax, some of which links to other sections which contain the files to be copied to the destination. Sections can be of a special type, indicating to the Installation Manager that the files require particular actions besides copying. The following .INF entry demonstrates the copying and registering of two accessories:

```
[app.copy.appstuff]
 #sample, 0, Accessory

[sample]
 1:sample1.exe, sample1.exe, "Sample1 accessory"
 1:sample2.exe, sample2.exe, "Sample2 accessory"
```

Sections:

Each section is identified by an alphanumeric name enclosed in square brackets (for example, [section]). Some section names are 'hard coded', requiring reserved names, while other section names are user defined. The syntax within a section varies with the section definition.

Comments:

A comment begins with a semicolon and can be included on the same line as syntax as long as it follows the syntax.

Spaces:

Spaces are ignored, except when between double quotes. Blank lines are ignored.

Conventions

Bold typeface denotes 'hard-coded', reserved values.

Only one statement per line.

Statements cannot be continued on multiple lines.

Items appearing in non-bold brackets are optional. Only the information within the brackets should appear in the INF statement, not the brackets themselves.

If bold brackets are present, they are part of they syntax and must be included.

A list may be given within braces, such as a | b | c. The vertical bar, |, means "or." Pick one item from the list to be included in the statement.

Data Section

Optional Section.

Function:

The data section specifies the directory to which files are copied.

Syntax:

```
[data]
defdir=d
codedir=ignore
format=VER
```

where:

d	Fully specified disk directory

Semantics:

The defdir statement defines the default destination directory. Defdir must include a drive letter. If Win.ini contains a codedir statement, it overrides defdir (unless codedir = ignore is specified). The defdir statement is optional. If it is not present, the default is "C:\INFOConn".

The codedir statement is optional. The default is that the codedir statement in Win.ini overrides the defdir statement in the INF. If codedir = ignore is specified, the codedir statement in Win.ini does not override the defdir statement.

The format = VER statement is required for the syntax as described in this section. It also triggers the Installation Manager to use Windows Ver.dll to check version information.

Example:

```
[data]
 defdir=c:\icwinapp
 codedir=ignore
 format=VER
```

Dialog Section

Optional section.

Function:

The dialog section allows you to specify a custom title for the Installation Manager's dialog box. If this section is not present, the default caption is "INFOConnect Installation."

Syntax:

```
[dialog]
caption=s
```

where:

s	String, in double quotes, max length = 128

Semantics:

The caption statement determines the value that appears in the title bar of Installation Manager's dialog box.

Example:

```
[dialog]
 caption="INFOConnect Connectivity Services"
```

Disks Section

Required section.

Function:

The disks section defines the distribution disks. These disk definitions can be for physically different disks as well as for different directories on the same disk.

Syntax:

```
[disks]
n=path,"title"
```

where:

n	The disk identifier, a single character 1-9 or A-Z (or a-z).
path	The source directory on the disk from which the Installation Manager copies the requested files.
title	String, a descriptive name for the disk. The title should match the distribution label exactly.

Semantics:

The Installation Manager prompts the user to enter the correct disk for the requested files. The path can be the root directory denoted by the period (.) or relative to the root directory denoted by period-slash-directory (.\directory). Any number of disk directories can be specified.

Example:

```
[disks]
 1=.,"INFOConnect lcWinApp Disk 1"
```

Package Section

Required section.

Function:

The package section provides information about the package to the Installation Manager. When a user selects the INFOConnect Package window, a list is presented showing the installed packages by the name and description given in the package section.

Syntax:

[package]
name=s32
description=s64
version=v
lowicver=v
highicver=v
release.notes=filename
chainlink=n:scriptname
chaindesc=s64
chaintext=n:textfile
publish.chainlink=n:scriptname
publish.chaindesc=s64
publish.chaintext=n:textfile

where:

s32	String, in double quotes, max length = 32
s64	String, in double quotes, max length = 64 (IC_MAX_DESCRIPTIONSIZE)
v	Major version, minor version, emu level and build revision
filename	The name of the file displayed, including the filename extension. This file is not copied as part of the installation process.
n:	The disk identifier: a single character 1-9, A-Z or a-z.
scriptname	The name of the .INF file used for chaining the installation of another package after the current .INF file installation is completed.
textfile	The name of the file displayed, including the filename extension. This file is not copied as part of the installation process.

Semantics:

The name statement must specify a unique package identifier and is limited to 32 characters; no embedded blanks are allowed. It is used as the primary key into the table of installed packages. The proposed convention is to use a marketing style id for the name.

The description statement specifies a string which is displayed along with the name field as a description of what the package contains.

The version statement is an informational field which assigns a version identifier to the package. The lowicver statement defines the minimum INFOConnect API level that the package requires. The highicver statement defines the highest INFOConnect API level that the package supports. All three parameters set fields of type IC_VER within the package table: pkg_version, low_icver, and high_icver.

If the release.notes statement is present, a dialog box is presented with the specified file at the start of installation. This provides you with a method for displaying text or instructions before the installation commences. This statement is optional.

If the chainlink and chaindesc (or publish.chainlink and publish.chaindesc for a publish installation) statements are present, the Installation Manager will "chain" install the package identified by chainlink. Chaindesc is a description of the "chain" package. After the current package is installed, the user will be prompted: "Do you want to install the <chaindesc> package?". The user has the option of selecting a Yes or No button. If chaintext (or publish.chaintext) is present, a Pkg Info button is also displayed. When the Pkg Info button is selected, Windows Notepad is executed with the specified text file.

Example:

```
[package]
 name="Sample"
 description="Sample Description"
 version=1,0,0,0
 lowicver=2,0,0,0
 highicver=3,0,0,0
 release.notes=readme.txt
 chainlink=1:next.inf
 chaindesc="Next package description"
 chaintext=1:next.txt
```

Progman.Groups Section
Publish.Progman.Groups Section
StandAlone.Progman.Groups Section

Note: *Notice the spelling, these are prog**man.groups** sections, not prog**ram.groups** sections.*

Optional sections.

Function:

The progman.groups section defines program icons that are to be built during a standalone and subscribe installation.

The publish.progman.groups section defines program icons that are to be built during a publish (/A) installation.

The standalone.progman.groups section defines program icons that are to be built during a standalone installation.

All three sections allow you to set the title of the Windows program group or to use the INFOConnect program group.

Syntax:

```
[progman.groups] | [publish.progman.groups] | [standalone.progman.groups]
groupname [,groupfile.grp]

[section-name]
"description",appfile.exe,[iconfile.exe [,icon#]]
```

where:

groupname	Is the title Program Manager displays under the icon that represents the group. The groupname is also the name of the .INF section that defines the contents of the group.
groupfile.grp	Is the filename in which Program Manager saves information about the group. This parameter is optional. If it is used, the **.grp** extension is required.
section-name	The section name is the groupname.
description	Is the text that will appear below the program icon when displayed in Program Manager. The description can be in double quotes (for example, "Shared Mgr").
appfile.exe	Is the command line that starts the application. This can be a quoted string (for example, "abc.exe /a").
iconfile.exe	Is a file that contains the icon that represents the <appfile.exe> application. This parameter is optional and may be of type EXE or ICO. Typically, this is the executable file itself. If omitted the first icon found in <appfile.exe> is used.
icon #	Is the offset of the icon within the <iconfile.exe> file. This parameter is optional, but if used is zero relative. To use the nth icon, specify the number n-1 in this field.

Semantics:

If 'INFOConnect' is used under [progman.groups] it links to the [INFOConnect] section that defines the icon values as well as naming the Windows Program Manager program group. When the INFOConnect Connectivity Services package is installed, the user is given to option to change the name of the INFOConnect program group. The name is recorded in WIN.INI as ProgramGroup. If any other script file references the "INFOConnect" group name, the value of ProgramGroup will be inserted in its place. A script file can choose its own program group by using a different groupname link in the script file.

Note: **INFOConnect** *is a 'hard coded', case sensitive reserved value.*

Examples:

The following code fragment demonstrates the progman.groups section using INFOConnect as the program group. The IcWinApp icon will be added to the INFOConnect program group with a description of IcWinApp. Remember, if the INFOConnect program group name was modified during installation of ICS, that program group name will be substituted for INFOConnect.

```
[progman.groups]
 INFOConnect,    infoconn.grp

[INFOConnect]
 IcWinApp,      icwinapp.exe
```

Examples:

The following code fragment demonstrates the progman.groups section using IcWinApp as the program group. An IcWinApp program group will be created in Program Manager if it doesn't exist. An IcWinApp icon will be built and added to the IcWinApp program group with a description of IcWinApp Sample.

```
[progman.groups]
 IcWinApp,      icwinapp.grp

[IcWinApp]
 "IcWinApp Sample", icwinapp.exe
```

The following code fragment demonstrates the publish.progman.groups section. An IcAdmin program group will be created in Program Manager if it doesn't exist. An INFOConnect icon will be built and added to the IcAdmin program group with a description of Shared Manager.

```
[publish.progman.groups]
 IcAdmin,    icadmin.grp

[IcAdmin]
 "Shared Manager",        "infoconn.exe /A"
```

Needed.Space Section

Required section.

Function:

The needed.space section defines how much disk space, in bytes, the application requires. If the specified amount of space is not available, the user is prompted to specify a different destination or exit the installation.

Syntax:

```
[needed.space]
minspace=i
```

where:

i	Integer number of needed disk space in bytes

Example:

```
[needed.space]
 minspace=100000
```

App.Copy.AppStuff Section
App.Copy.NoRemove Section
App.Copy.Publish Section
App.Copy.Subscribe Section
App.Copy.StandAlone Section

Optional sections.

Function:

You may define one or more copy sections in the .INF script specifying which files to copy from the release disk(s) to the destination directory. These sections can be a simple list of files as in the undefined type section or they can specify a special type of file to copy such as in a library or a font type section. The following is a list of the "hard coded" sections available in the install script (.INF) file:

- The app.copy.appstuff section contains installation instructions that are executed during standalone, publish (install /A) and subscribe (install /N) installations.
- The app.copy.noremove section contains installation instructions that are executed during standalone, publish and subscribe installations. It defines files to be copied which are not removed by the deinstallation process, except during a complete deinstall (install /U).
- The app.copy.publish section contains installation instructions that are executed during a publish installation.
- The app.copy.subscribe section contains installation instructions that are interpreted during a publish installation and are executed during a subscribe installation.
- The app.copy.standalone section contains installation instructions that are executed during a standalone installation.

Syntax:

[app.copy.appstuff] | [app.copy.noremove] | [app.copy.publish] |
[app.copy.subscribe] | [app.copy.standalone]
#section-name, d:dest[, type section]

[section-name]
n:source-file, dest-file, "desc"[, topic-index | , **delete** | , font-name]

where:

section-name	The name of the INF section that lists the files to be copied.
d	A character representing the destination directory where: **0:** Represents the installation directory **$:** Represents the Codedir directory **%:** Represents the Datadir directory **&:** Represents the windows directory **^:** Represents the windows system directory **!:** Represents the installation directory (0:) during a standalone or publish install. During a subscribe install, designates that the files are to be moved to the workstation so that it does not share the published files.
dest	The pathname of the destination directory to which the Installation Manager copies the requested files.
[, type section]	The following are the different section types which can be specified in the INF to copy files. Typed sections are pointed to by one of the copy sections: • The *accessory* type section copies the specified files to the destination directory. In addition, the files are registered as accessories in the configuration database. INFOConnect accessories are required to have special INFOConnect-related resources in their resource files. During registration, this information is transferred to the Configuration Manager and recorded in the configuration database. • The *bitmap* type section copies the specified files to the Windows directory. The user is informed of this action by the final dialog box.

[, type section]
(continued)

- The *driver* type section copies the specified files to the destination directory. In addition, the user is presented with a MODIFY button on the *Installation Complete* dialog box. If chosen, Windows NOTEPAD is launched with C:\Config.sys loaded for user viewing and modification.
- The *font* type section copies the specified .FON files to the Windows system directory. *Font-name* is an optional parameter as defined in WININI2.txt under [FONTS]. The font is then registered with Windows with the AddFontResource function. For Windows 3.1 or higher, a .TTF file is copied to the destination directory. CreateScalableFontResource is executed to create a .FOT file in the destination directory. The font is then registered with Windows with the AddFontResource function.
- The *help* type section copies the specified files to the destination directory. In addition, the user is presented with a Help button in the Installation Complete dialog box which allows the user to enter directly into the last help file copied at the specified topic. Therefore, only one help section should be specified. If more than one help section is specified, all the files listed are copied, but only the last file in the last section is accessed by the Help button. An optional parameter for the help type section is *topic-index*. A non-zero value denotes a help topic. A zero value denotes an index.
- The *library* type section copies the specified files to the destination directory. In addition, the files are registered as libraries in the configuration database. If the library has templates in it, the template resources are also registered. INFOConnect libraries are required to have special INFOConnect-related resources in their resource files. During registration, this information is transferred to the Configuration Manager and recorded in the configuration database.
- The *noremove* type section marks the section as a "no remove" section. The files are not deleted during package deinstallation. This type section can be used for files that are shared across packages.

[, type section] (continued)

- The *purvey* type section copies files to the Windows directory. The files being copied are OEM files. If these files are already in use, the Installation Manager checks to see what version of the files are currently loaded. If the 3.1 version is already present, the Installation Manager does not need to copy the file. If the 3.0 version of the file is present, the Installation Manager gives the user an error message telling them to close the in-use file so that it can be copied.
- The *readme* type section copies the specified files to the destination directory. In addition, the user is presented with a Readme button in the Installation Complete dialog box. If this button is chosen, Windows NOTEPAD is launched with the readme text in the last file copied for viewing. Therefore, only one readme section should be specified. If more than one is specified, all the files listed are copied, but only the last file in the last section is accessed by the Readme button.
- The *system* type section copies the specified files to the Windows system directory (c:\windows\system). An optional parameter for the system type section is *delete*. If a file of the same name exists in the destination directory, it is deleted.
- The *template* type section copies the specified files to the destination directory. In addition, all template resource data contained in the files is extracted and registered in the configuration database.
- The *undefined* type section copies the specified files to the destination directory. No special actions are taken for those files. The *undefined* type sections are sections where the optional section-type parameter has not been specified. You should not put "undefined" as a section type in the syntax.
- The *windows* type section copies the specified files to the Windows directory (c:\windows). An optional parameter for the windows type section is *delete*. If a file of the same name exists in the destination directory, it is deleted.
- The *winpurvey* type section copies files to the windows directory. The files being copied are OEM files. If these files are already in use, the Installation Manager checks to see what version of the files are currently loaded. If the 3.1 version is already present, the Installation Manager does not need to copy the file. If the 3.0 version of the file is present, the Installation Manager gives the user an error message telling them to close the in-use file so that it can be copied.

n:	The disk identifier: a single character 1-9, A-Z or a-z.
source-file	Specifies the name of a file or set of files from which you want to copy, including the filename extension. The source file can be either compressed or uncompressed. The convention from the Microsoft compress utility is to replace the last letter of the file extension with an underscore ("_") if the source is a compressed file. The Microsoft VER.dll library, included with INFOConnect, is used to copy the file and thereby provides decompression and version support as well.
dest-file	Specifies the name of the file or set of files to which you want to copy, including the filename extension.
desc	Descriptive text that is being displayed as the file is being copied. If this field is blank the previous value will continue to be used.
topic-index	Option for the help type section.
delete	Option for the system and windows type sections.
font-name	Option for the font type section.

Semantics:

In some cases, it may be necessary to create a dummy directory level; that is, a directory which holds only directories but no actual files. For example, if the INF writer wishes to place files in a directory \A\B, he must first create the directory \A, then the directory \A\B can be created. The directory \A is referred to here as a dummy directory. To create that dummy directory, the INF writer should specify a section for that directory. The section that is specified must be present and empty. The IcSetup.inf file for the IDK contains examples of creating dummy directories.

Example:

The following example demonstrates the syntax of type sections described above:

```
[app.copy.appstuff]
 #acc,     0:,   Accessory
 #readme,  0:,   Readme
 #source,  0:src
 #bitmap,  0:,   Bitmap
 #drvr,    0:,   Driver
 #lib,     0:,   library
 #help,    0:,   help

[app.copy.noremove]
 #unitslib,  0:,   System
 #unitsutil, $:
 #fonts,     0:,   Font

[app.copy.publish]
 #publish,   0:

[app.copy.subscribe]
 #subscribe, 0:

[acc]
  1:sample.exe, sample.exe,  "Sample Accessory"

[readme]
  1:readme.txt, readme.txt,  "Readme file"

[source]
  1:sample.c,  sample.c,    "Source files"
  1:sample.h,  sample.h
  1:sample.rc, sample.rc

[bitmap]
  1:sample.bmp, sample.bmp,  "Bitmap file"

[drvr]
 1:DEVCOM.SYS,  DEVCOM.SYS,  "Device Driver"

[fonts]
 1:ABC.FON, ABC.FON, "ABC.FON", Helv
 1:XYZ.TTF, XYZ.TTF, "True Type XYZ"

[lib]
  1:reflect.dll, reflect.dll, "Reflect External Interface"
```

```
[help]
  1:reflect.hlp, reflect.hlp, "Reflect On-line Help"

[unitslib]
 1:ICCUA.DLL,  ICCUA.DLL,  "Usage Allowances DLL"

[unitsutil]
 1:ICCUNITS.EXE, ICCUNITS.EXE, "Usage Units Utility"
 1:ICCUNITS.HLP, ICCUNITS.HLP, "Usage Units Util Help"

[publish]
 1:INSTALL.EXE,  INSTALL.EXE, "INSTALL.EXE"

[subscribe]
 1:INSTALL.EXE,  INSTALL.EXE, "INSTALL.EXE"
```

INF Examples

IcWinApp INF

The following is the INF file for IcWinApp from the IDK sample directory. It illustrates the use of accessory, readme and undefined type sections.

```
;***********************************************************
;* ICWINAPP.INF Installation Script File          *
;*                               *
;*        Sample INFOConnect Windows application   *
;*                               *
;***********************************************************

[dialog]
  caption   ="INFOConnect IcWinApp Accessory"

[package]
;'name' cannot contain blank characters
  name     ="IcWinApp"
  description="Sample Accessory"
  version=3,0,0,0
  lowicver=2,0,0,0
  highicver=3,0,0,0

[data]
  defdir =c:\icwinapp
  codedir=ignore
  format =VER

[disks]
  1 =.,"INFOConnect IcWinApp Disk 1"

[needed.space]
  minspace=120000

[app.copy.appstuff]
  #acc,  0:,   Accessory
  #readme,0:,   Readme
  #source,0:src

[acc]
  1:icwinapp.exe, icwinapp.exe,"IcWinApp Sample Accessory"
```

```
[readme]
  1:icwinapp.txt, icwinapp.txt, "Readme file"

[source]
  1:icwinapp.c,  icwinapp.c,  "Source files"
  1:icwinapp.h,  icwinapp.h
  1:icwinapp.rc,  icwinapp.rc

[progman.groups]
  INFOConnect,   infoconn.grp

[INFOConnect]
  IcWinApp,     icwinapp.exe
```

Reflect INF

The following is the .INF file for Reflect from the IDK sample directory. It illustrates the use of library and help type sections.

```
;*************************************************************
;* REFLECT.INF Installation Script File          *
;*                                               *
;*      Sample INFOConnect Windows Library       *
;*                                               *
;*************************************************************

[dialog]
  caption   ="INFOConnect Reflect EIL"

[package]
;'name' cannot contain blank characters
  name      ="Reflect"
  seqnumber =4000
  description="Sample External Interface"
  version=3,0,0,0
  lowicver=3,0,0,0
  highicver=3,0,0,0

[data]
  defdir=c:\infoconn
  format=VER  ;Use Ver Utility, three parameters copy-from sections
         ;DOS copy, two parameters copy-from sections

[disks]
  1=.,"INFOConnect Reflect Disk 1"

[needed.space]
  minspace=70000

[app.copy.appstuff]
  #lib, 0:, library
  #help, 0:, help

[lib]
  1:reflect.dll, reflect.dll, "Reflect External Interface"

[help]
  1:reflect.hlp, reflect.hlp, "Reflect Online Help"
```

Section 10
Converting from Previous Releases

This section contains two subsections: "Converting from Release 2.0 to 3.0" and "Converting from Release 1.0 to 2.0."

Note: *Be careful when keeping multiple versions of the IDK on your machine. Verify that the compiler and/or linker is not finding more than one of them. This is a common situation that causes compilation errors. Check all the directories in your INCLUDE environment variable for IcDef.h, one of the INFOConnect header files. Be sure the compiler only finds one copy of IcDef.h.*

Converting From Release 2.0 to 3.0

This section describes the new features of INFOConnect Release 3.0. It also identifies new features that are available to existing 2.0 INFOConnect applications and libraries after implementing source changes.

3.0 Features

Architecture Diagram

The main difference between the INFOConnect 2.0 and 3.0 architecture is the addition of a new INFOConnect component, the Application Interface Library (AIL). The AIL component of an INFOConnect session allows for the coexistence of multiple communications interfaces since it provides the session related interfaces of the Accessory Application Programming Interface (AAPI).

The INFOConnect architecture diagram has been updated to illustrate the flow of data through the different INFOConnect components. The large vertical arrows represent the application's data passing through the INFOConnect architecture. The thinner black lines indicate the flow of control through the INFOConnect architecture. The INFOConnect Manager controls all interaction between the four components of the INFOConnect architecture: Accessory/Application, Application Interface Library, Service Library, and External Interface Library.

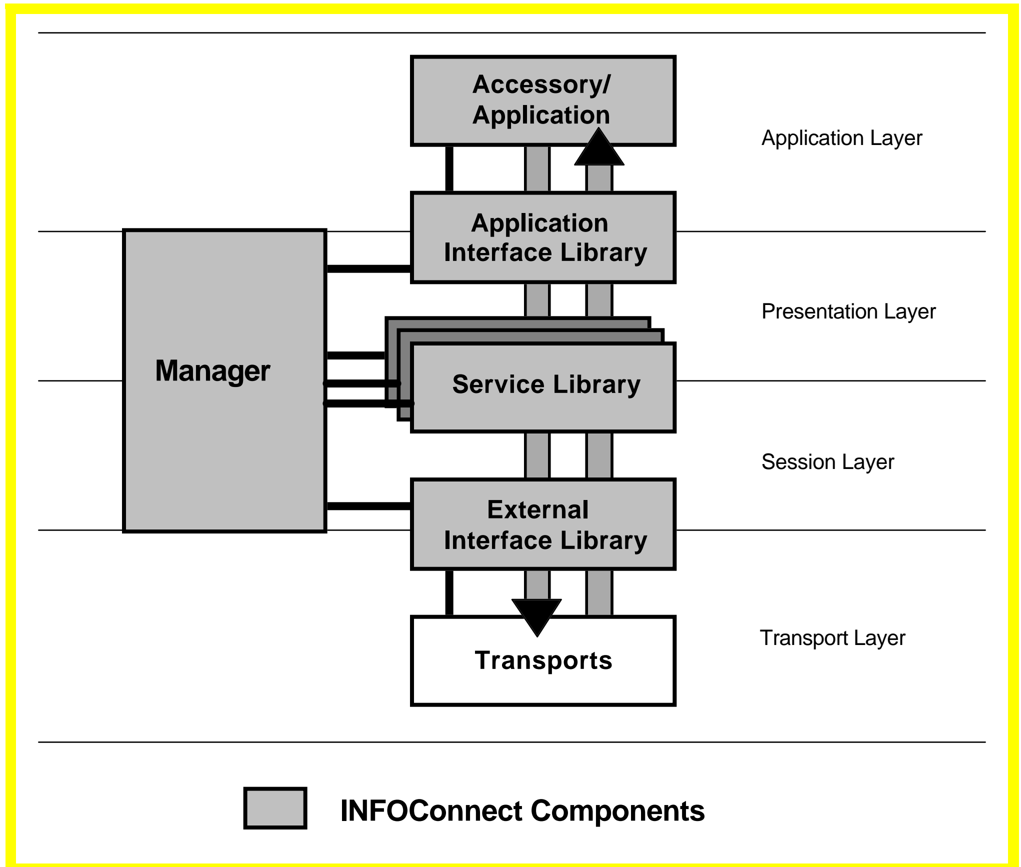


Figure 10–1. INFOConnect 3.0 Architecture

Manager Components

The following is a list of each Manager Component, its dynamic link library (.DLL) or executable (.EXE) name, and the API it provides to Applications, Accessories and Libraries:

Manager Component	DLL/EXE	API
INFOConnect Manager	INFOConn.exe	-
Communication Manager	IcMgr.dll	Library API and Manager API
Configuration Manager	IcMgrCfg.dll	Configuration Accessory API
Installation Manager	InstMgr.exe	-
Quick Configuration Manager	IcQCfg.exe	-
Database Manager	IcDb.dll	-
Utilities	IcUtil.dll and IcAbout.dll	-

Note: *The Application Interface Library (IcAAPI16.dll) provides the session related interfaces of the ICS Accessory Application Programming Interface (AAPI).*

INFOConnect release 3.0 contains the following changes:

- CommMgr.dll is now called IcMgr.dll. CommMgr.dll is now a stub that calls IcMgr.dll or IcAAPI16.dll as appropriate.
- The session related interfaces of the Accessory Application Programming Interface (AAPI) are provided by the Application Interface Library. The Communication Manager component provides the rest of the APIs and the non-session related interfaces of the AAPI to accessories, applications and libraries.

See Section 6 "A Closer Look at the INFOConnect Architecture," for a detailed description and diagram of the Manager Components.

Application Interface Library

An Application Interface Library (AIL) is the component that supports a specific set of INFOConnect interfaces and provides the application interface to INFOConnect communications. In INFOConnect 3.0, the INFOConnect Accessory AIL (IcAAPI16.dll) exports the session related interfaces of the Accessory API for the Windows (Win16) platform.

The AIL component of an INFOConnect session allows for the coexistence of multiple communications interfaces. The AIL requests establishment of an INFOConnect session by calling the Communication Manager to associate the AIL with a path. This allows different applications to use the same path at the same time, or different times, even though they may use different communication APIs.

Interprocess Interface Library

An Interprocess Interface Library (IIL) acts as both an AIL and an EIL. An IIL associates two sessions in **different processes** by internally linking the EIL role of one session to the AIL role of the other session. The IIL is automatically included in sessions when an AIL requests a path that must be opened in a different process. An IIL shields the application from needing to know which underlying environment the INFOConnect subsystem is running in.

Stack Interface Library

A Stack Interface library acts as both an AIL and an EIL. A Stack library associates two sessions in the **same processes** by internally linking the EIL role of one session to the AIL role of the other session. The Stack library associates two sessions by internally linking the EIL role of one session to the AIL role of the other session. Stack libraries can be included in path templates as an EIL.

Switching Library

A type of Stack Interface library that stacks one session (where it is configured as an EIL) on top of another session and filters the data stream for commands to open and close the lower session.

Hook Library

A special purpose library that provides special features to the INFOConnect Connectivity Manager. The INFOConnect Trace Facility is a hook library that manages the trace log file and writes trace information to it. Details on how to add entries to the trace log are provided in Section 8, "Debugging."

Shell

A Shell is an INFOConnect accessory that runs as the INFOConnect Manager. The INFOConnect architecture requires the Manager to be a separately running Windows task. The Manager can optionally provide a user interface to allow session monitoring. It may also include a configurator. Different INFOConnect shells can be developed using the INFOConnect Shell API. Ic16SS.exe is an alternate INFOConnect Shell available with INFOConnect 3.0.

Configuration API

The Configuration API provides interfaces to get and set individual configuration fields within any INFOConnect Component. The Configuration API can be used by an accessory or library which has been initialized as a Configurator.

Configurator

A Configurator is an INFOConnect accessory that provides the user interface for INFOConnect configuration. The INFOConnect architecture allows more than one configurator to be executing simultaneously. Configurators use the Configurator API.

ICS Version Numbers

INFOConnect Connectivity Services version numbers contains four fields: major version, minor version, emu level, and build revision. IC_VERSION_FILE and IC_VERSION_PRODUCT are defined in IcDef.h as follows:

```
#define IC_VERSION_FILE   IC_MAJOR_VERSION,IC_MINOR_VERSION,IC_EMU_LEVEL
          IC_BUILD_REVISION
#define IC_VERSION_PRODUCT IC_MAJOR_VERSION,IC_MINOR_VERSION,IC_EMU_LEVEL,
          IC_BUILD_REVISION
```

The File Version and Product Version can be viewed as follows:

```
/* File/Product Version Information           */
/* image format 3.00 0 (000)              */
/*      *                                  Major version */
/*       **                                     Minor version */
/*         *                               EMU Level    */
/*           ***                      Build revision */
```

Using major release 3.0 as an example: IC_MAJOR_VERSION is 3 and IC_MINOR_VERSION, IC_EMU_LEVEL, and IC_BUILD_REVISION are 0.

Another way to look at ICS version numbers is to group the major version and minor version fields as "Version" information and the EMU level and build revision fields as "Revision" information. The IC_VER_INFO data structure allows programmers to access ICS version information as four BYTE fields or two WORD fields:

```
typedef LONG IC_VER;
typedef union {
  IC_VER IcVer;
  struct {
    WORD Rev;
    WORD Ver;
  } w;
  struct {
    BYTE Revision;
    BYTE EmuLevel;
    BYTE MinorVersion;
    BYTE MajorVersion;
  } b;
} IC_VER_INFO;
```

To continue the 3.0 example, *Ver* consists of major and minor version and is equivalent to IC_VERSION_3_0 (defined as 0x0300). *Rev*, consists of EMU level and revision and is equivalent to IC_REVISION_3_0 (defined as 0x0000) IC_VERSION_... and IC_REVISION_... defines are also found in IcDef.h.

Component Numbers

Component numbers are used by Configurators to configure INFOConnect components. There are two types of component numbers: generic and branded (supplier specific). Generic and branded component number consists of two parts: a component number and a supplier number.

The supplier number and component number of a generic IC_COMPONENT are assigned by the Malvern Development Group. Generic IC_COMPONENT values are defined in Ic.hic.

An INFOConnect component with a non-zero generic component number must perform the same function as other INFOConnect components that designate the same generic number. The configuration information for the generic component is defined in the specified generic component's .HIC file. For example, a library using a generic number of IC_GENERIC_TCP must use the configuration information defined in IcTCP.hic.

The branded IC_COMPONENT uniquely identifies the INFOConnect component. The supplier number of a branded IC_COMPONENT is assigned by the Malvern Development Group. The branded IC_COMPONENT supplier numbers are defined in Ic.hic. The component number of a branded IC_COMPONENT is assigned by the vendor.

Trace Functions

There are two functions, IcMgrTraceBuffer and IcMgrTraceResult, which allow applications and libraries to write information to the trace.log file when the trace log is active.

The IcMgrTraceResult function can be called to append a trace entry containing an IC_RESULT in the trace.log file. IcMgrTraceBuffer can be used by to add data or state information to the trace log. These two functions can be used in conjunction with one another if an unexpected error result is received: first log the result with IcMgrTraceResult and then log pertinent session information with IcMgrTraceBuffer. IcMgrTraceBuffer can also be used to trace transmit and receive buffers which appear to be "lost."

Details on how to add entries to the trace log are provided in Section 8, "Debugging."

IC_STATUS_BUFFER

When an application needs to exchange more information with an ICS library than IC_RESULT_VALUE can store, it can send a buffer of information with the IC_STATUS_BUFFER extended status. Extended statuses can be exchanged in two ways: synchronously and asynchronously. Information on using extended statuses can be found in Chapter 7, "Writing INFOConnect Libraries for Windows 3.x."

Changes Affecting Applications and Libraries

INFOConnect Version Identification

Note: *This section provides information on identifying the INFOConnect release levels that your application or library has been coded to run with. Your application or library can run with INFOConnect 3.0 without making any modifications. Do not change the INFOConnect version information to reflect INFOConnect 3.0 until the modifications specified in "Changes Affecting Applications and Libraries" and "Changes Affecting Applications" or "Changes Affecting Libraries" have been completed.*

The INFOConnect RCDATA section of the resource file (.RC) identifies the minimum and maximum INFOConnect Connectivity Services versions for which your application or library has been coded.

The package section of an installation script file (.INF), which is used to install an INFOConnect package, contains similar INFOConnect Connectivity Services version numbers for the package.

INFOConnect RCDATA version information

The INFOConnect RCDATA section of the application's or library's resource file contains a minimum ICS version/revision level, a maximum ICS version/revision level.

The RCDATA section only needs to be updated if the application or library is using INFOConnect 3.0 functionality. Otherwise, continue to use the 2.0 format.

```
INFOConnect RCDATA
BEGIN
 IC_VERSION_2_0,
 IC_REVISION_2_0,
...
 IC_HEADER_3_0
...

 IC_VERSION_3_0,
 IC_REVISION_3_0,
...
END
```

In 2.0, the version and revision fields (IC_VERSION_2_0 and IC_REVISION_2_0) referred to the version of the IDK with which the component was built. In 3.0, it refers to the minimum (or oldest) level of Connectivity Services that the library requires for proper operation. Older levels of Connectivity Services will refuse to load the library.

The header_size field (IC_HEADER_3_0) specifies the size of the header in the INFOConnect RCDATA section of the resource file. New fields have been appended to the 2.0 definition of the IC_RC_NODE data structure. IC_HEADER_3_0 designates the increased 3.0 size. IC_HEADER_SIZE should still be used if the application or library is using the 2.0 definition of the IC_RC_DATA data structure. The IC_RC_NODE data structure is defined in the *IDK Programming Reference Manual*.

IC_VERSION_3_0 and IC_REVISION_3_0 specify the maximum (or highest) level of Connectivity Services that the library was developed with, in order to take advantage of that level of ICS features.

Note: *IC_VERSION and IC_REVISION in prior releases of the IDK were updated to reflect the current release level of the IDK. For IDK release 3.0 onward, IC_VERSION and IC_REVISION will reflect the 2.0 IDK version and revision values. New IC_VERSION_... and IC_REVISION_... values will be added for each release of the IDK.*

RCDATA section modifications are not required if the application or library wants to run with INFOConnect 3.0 but does not take advantage of the new 3.0 functionality. To make it easier to upgrade in the future, update the RCDATA section as follows:

- Set version and revision fields to designate IC_VERSION_2_0 and IC_REVISION_2_0.
- Set maximum version and revision fields to designate IC_VERSION_2_0 and IC_REVISION_2_0.
- Fill in all of the fields of the IC_RC_NODE data structure and set the header size to IC_HEADER_3_0.

If the application or library takes advantage of new 3.0 functionality and does not need to run with INFOConnect 2.0, update the RCDATA section as follows:

- Set the version and revision fields to designate IC_VERSION_3_0 and IC_REVISION_3_0.
- Set maximum version and revision fields to designate IC_VERSION_3_0 and IC_REVISION_3_0.
- Set the header size to IC_HEADER_3_0.

If the application or library takes advantage of new 3.0 functionality and also wants to run with INFOConnect 2.0, update the RCDATA section as follows:

- Set the version and revision fields to designate IC_VERSION_2_0 and IC_REVISION_2_0.
- Set the maximum version and revision fields to designate IC_VERSION_3_0 and IC_REVISION_3_0.
- Set the header size to IC_HEADER_3_0.

Note: *Additional code is required when running with both ICS 2.0 and ICS 3.0. The library or application must be coded so that it does not use any 3.0 functionality when running with INFOConnect 2.0.*

Installation script file version information

There are three parameters containing version information in the [package] section of the installation script file (*.INF): version, lowicver and highicver:

- The **version** parameter is an informational field which assigns a version identifier to the package.
- The **lowicver** parameter defines the minimum INFOConnect API level that the package requires.
- The **highicver** parameter defines the highest INFOConnect API level that the package supports

Each parameter contains four version number fields: major version, minor version, emu level, and build revision:

```
[package]
;'name' cannot contain blank characters
 name     ="Reflect"
 description="Sample External Interface"
 version=  2, 0,0, 0
 lowicver= 2, 0,0, 0
 highicver= 3,00,0,000
/*      *                                   Major version */
/*        **                                      Minor version */
/*          *                               EMU Level    */
/*            ***                     Build revision */
```

Installation scripts are discussed in detail in Section 9, "Packaging an INFOConnect Application."

Notes:

- *When highicver is set to "3, 0, 0, 0", which is equivalent to IC_VERSION_3_0 (0x0300) and IC_REVISION_3_0 (0x0000) or higher, the 3.0 Quick Configuration model is activated during installation.*
- *The Installation Manager will also check the version, revision, max_version and max_revision values specified in the library's resource file (.RC) to verify that the library can handle the 3.0 Quick Configuration model.*

Component Numbers

The generic and branded component numbers are assigned in the RCDATA section of the INFOConnect component's resource file.

Defining Component Numbers

The 3.0 INFOConnect RCDATA section of the INFOConnect component's resource file contains the generic and branded component numbers: GenericNum and SupplierNum.

The GenericNum field is assigned a 0 if the component is not performing the same function as a currently defined generic component; otherwise, assign the generic number defined in Ic.hic.

The SupplerNum field is assigned a unique branded component number defined by the vendor.

Note: *Component numbers do have to be assigned. If the branded component is defined as 0,0 (or if still using the 2.0 IC_RC_NODE data structure which does not contain the component numbers), a unique value will be assigned when the INFOConnect component is added to the INFOConnect configuration database.*

The following code fragment is from Service.rc:

```
INFOConnect RCDATA
BEGIN
...
 IC_HEADER_3_0
...
 0,0,        /* no GenericNum               */
 SAMPLE_SERVICE,  /* SupplierNum for the Service Sample */
 UIS_SAMPLE      /* Source uses UIS_SAMPLE_SERVICE    */
            /* See Ic.hic and IcSample.hic      */
END
```

Managing Branded Component Numbers

Each vendor is responsible for managing the component numbers for all of their INFOConnect components, ensuring uniqueness. The branded component numbers for the IDK samples are defined in a file called IcSample.hic, which is in the IDK include directory. Vendors may want to adapt this format for defining their branded component numbers.

The following are code fragments from Ic.hic and IcSample.hic that show the branded component definitions for the sample service libraries in the IDK:

```
/* Include File : Ic.hic   */

#define UIS_SAMPLE   1
```

```
/* Include File : IcSample.hic   */

/* IC_SERVICE samples         1 - 99 */
#define SAMPLE_PS2TTY      1
#define SAMPLE_COUPLES     2
#define SAMPLE_SERVICE     3
#define UIS_SAMPLE_PS2TTY    MAKELONG(SAMPLE_PS2TTY ,UIS_SAMPLE)
#define UIS_SAMPLE_COUPLES   MAKELONG(SAMPLE_COUPLES ,UIS_SAMPLE)
#define UIS_SAMPLE_SERVICE   MAKELONG(SAMPLE_SERVICE ,UIS_SAMPLE)
```

Changes Affecting Applications

Existing applications will run without change. However, some recommended changes are explained. Other new features of particular interest to application developers are also explained.

CommMgr.lib Removed

Note: *This change only affects Windows-specific applications. XVT users can ignore it.*

CommMgr.lib has been replaced by IcWin.lib in the 3.0 release. INFOConnect now consists of several DLLs and IcWin.lib resolves references to the appropriate DLL.

If your library or application runs with ICS release 2.0 and 3.0, link with IcWin20.lib.

The 2.0 IDK included both CommMgr.lib and IcWin.lib. Update your makefiles to use IcWin.lib instead of CommMgr.lib when linking.

New XVT Link Libraries (.LIB)

Link your XVT application with one of the following link libraries depending on which XVT release (2.0 or 3.0x), memory model (medium or large), and ICS release (ICS release 3.0 only or ICS releases 2.0 and 3.0) the application will be using:

IcXvtL.lib	Resolves all references to the 3.0 Release of ICS functions for XVT 3.0x (Large memory model)
IcXvtM.lib	Resolves all references to the 3.0 Release of ICS functions for XVT 3.0x (Medium memory model)
IcXvt2L.lib	Resolves all references to the 3.0 Release of ICS functions for XVT 2.0 (Large memory model)
IcXvt2M.lib	Resolves all references to the 3.0 Release of ICS functions for XVT 2.0 (Medium memory model)
Ic2XvtL.lib	Resolves all references to the 2.0 and 3.0 Releases of ICS functions for XVT3.0x (Large memory model)
Ic2XvtM.lib	Resolves all references to the 2.0 and 3.0 Releases of ICS functions for XVT 3.0x (Medium memory model)
Ic2Xvt2L.lib	Resolves all references to the 2.0 and 3.0 Releases of ICS functions for XVT 2.0 (Large memory model)
Ic2Xvt2M.lib	Resolves all references to the 2.0 and 3.0 Releases of ICS functions for XVT 2.0 (Medium memory model)

ICXVTWIN Tag

This tag must be defined by XVT applications that also include the Windows.h include file. Add the following line before #include <xvt.h>:

```
#define ICXVTWIN
```

New IC_STATUS_COMMMGR Status

As the INFOConnect Manager initializes or shuts down, it sends IC_STATUS_COMMMGR status messages to indicate the different states of the initialization or termination process. Existing applications are not required to recognize these messages, but may find them useful. A new termination status has been added in INFOConnect 3.0, IC_COMMMGR_QUERYSHUTDOWN. This status is sent to all ICS communications sessions when Windows is exiting. If the application does not wish to close the session, it should cancel the exit by calling IcExitOk(FALSE). The IC_STATUS_COMMMGR statuses (IC_COMMMGR_...) are documented in the *IDK Programming Reference Manual*.

New IC_STATUS_TRANS Statuses

In order for INFOConnect to keep an accurate count of transactions, your application must notify INFOConnect of the beginning and the end of the transactions. Use IC_TRANSACTION_ON and IC_TRANSACTION_OFF to indicate whether or not transactions will be flanked by IC_TRANSACTION_BEGIN and IC_TRANSACTION_END status messages.

IcRegisterMsgSession

Note: *In order to use this new API interface the application must use IC_VERSION_3_0 and IC_REVISION_3_0 for max_version, max_revision and its call to IcInitIcs.*

This function registers the ICS messages with Windows on a per-session basis. This function allows developers to add INFOConnect messages to the message switch statement in MainWinProc. IcWinApp.c provides sample code using IcRegisterMsgSession.

Running with Old Versions of INFOConnect

Updating your application to a new version of the IDK will normally prevent your application from running with older versions of INFOConnect. See "Running with Old Versions of INFOConnect" in Sections 3, 4, or 5 for instructions on how to build with a new IDK and continue to run with older versions of INFOConnect.

Changes Affecting Libraries

Note: *Service and external interface libraries compiled with the 2.0 release of the IDK will run with INFOConnect release 3.0 without recompiling. This section provides information for recompiling libraries with the 3.0 IDK, whether it is just to update the library or to take advantage of new INFOConnect 3.0 functionality. Each section will contain information detailing if the change is optional, required, or for new 3.0 functionality.*

Library API Changes

The following routines provided in the 2.0 IDK for session communications have been renamed.

Note: *The 2.0 API names can still be used. Stubs, that call the new API routines in IcMgr.dll, have been provided in CommMgr.dll. If your library still supports INFOConnect release 2.0, continue to use the 2.0 API names. (Additional Note: The 2.0 API routines may not be supported in the 4.0 IDK).*

2.0 API Name	3.0 API Name
IcLibraryLcl	IcMgrLcl
IcLibraryRcv	IcMgrRcv
IcLibraryXmt	IcMgrXmt
IcSetResult	IcMgrSetResult
IcSendEvent	IcMgrSendEvent ***Note:*** *The first two parameters have been reversed.*
IcPostEILEvent	IcMgrEilEvent ***Note:*** *Parameters are different from 2.0 API.*

Changes for IcLib Functions

The following IcLib... functions have new or modified parameters in the 3.0 IDK. These parameters must be updated in order to compile without errors and warnings. Changes required for upgrading to INFOConnect 3.0 are also designated, but not required.

IcLibOpenChannel

Old Parameter	New Parameter
unsigned len	UINT len
BOOL bVerify	IC_OPEN_OPTIONS Options
LPHIC_SESSION lpchannel	LPHIC_CHANNEL lphLibChannel

Service and Interface libraries should use the IC_OPEN_VERIFY value for Options.

Old code fragment:

```
if(bVerify){
 return IC_VERIFY_OK;
}
```

Suggested new code:

```
if(Options & IC_OPEN_VERIFY){
 return IC_VERIFY_OK;
}
```

IcLibOpenSession

Old Parameter	New Parameter
unsigned len	UINT len
BOOL bVerify	IC_OPEN_OPTIONS Options

Service and Interface libraries should use the IC_OPEN_VERIFY value for Options.

Old code fragment:

```
if (bVerify) {
  return IC_VERIFY_OK;
}
```

Suggested new code:

```
if (Options & IC_OPEN_VERIFY) {
  return IC_VERIFY_OK;
}
```

IcLibEvent

Old Parameter	New Parameter
unsigned uType	UINT uType
unsigned uSize	UINT uSize

Libraries that support ICS 3.0 must be prepared for events with hLibSession set to NULL_HIC_SESSION. These are global events of possible interest to the library. The library should not pass these events up the stack by calling IcMgrSendEvent. In 3.0 there are two instances of an IC_STATUS event with hLibSession set to NULL_HIC_SESSION: IC_COMMMGR_INITIALIZED and IC_COMMMGR_TERMINATED.

In 3.0 there are two new event types: IC_SENDSTATUS and IC_SESSIONCLOSED. External Interface Libraries that do not support ICS 3.0 or higher only receive an IC_SESSIONESTABLISHED event.

For all events that do not have hLibSession set to NULL_HIC_SESSION, be sure to use IcMgrSendEvent (or IcSendEvent if running with 2.0 and 3.0) to pass the event to the next higher layer in the library stack, including unknown events.

Note: *If the IC_SESSIONCLOSED is not passed up the stack the session will not close. If a library with a maximum version less than IC_VERSION_3_0 in the INFOConnect RCDATA resource does not call IcMgrSendEvent, a session closed event will be passed up the stack when the library returns.*

IcLibLcl

Old Parameter	New Parameter
WORD type	IC_LCL_FLAGS which

Be sure to use IcMgrLcl (or IcLibraryLcl if running with 2.0 and 3.0) in order to pass the message down to the next library in the library stack.

Note: *If the IC_LCL_CLOSESESSION is not passed down the stack the session will not close. If a library with a maximum version less than IC_VERSION_3_0 in the INFOConnect RCDATA resource does not call IcMgrLcl, a close session will be passed down the stack when the library returns.*

IcLibRcv

Old Parameter	New Parameter
unsigned length	UINT length

Be sure to use IcMgrRcv (or IcLibraryRcv if running with 2.0 and 3.0) to post the receive request down to the next library in the library stack.

IcLibXmt

Old Parameter	New Parameter
unsigned length	UINT length

Be sure to use IcMgrXmt (or IcLibraryXmt if running with 2.0 and 3.0) to post the transmit request down to the next library in the library stack.

IcLibSetResult

Old Parameter	New Parameter
unsigned uType	UINT uType

Be sure to use IcMgrSetResult (or IcSetResult if running with 2.0 and 3.0) to pass status or error information down to the next library in the library stack.

IcLibGetString

Old Parameter	New Parameter
unsigned length	UINT length

Be sure to check that hLibSession is not NULL_HIC_SESSION. No additional changes are needed.

IcLibUpdateConfig

Old Parameter	New Parameter
HWND hWnd	HIC_CONFIG hConfig ***Watch:*** *The first parameter is a HIC_CONFIG, not HWND.*
int TableNumber	UINT TableNumber
unsigned len	UINT len
enum IC_COMMAND Command	IC_COMMAND Command

Note: *The following updates are required for 3.0. Be sure the updates have been completed prior to setting the maximum version in the library's resource file to reflect INFOConnect 3.0.*

Two new IC_COMMAND types have been added in INFOConnect 3.0: IC_CMD_SAVE and IC_CMD_DISCARD. IC_CMD_SAVE is received immediately before the data is saved to the data base. IC_CMD_DISCARD is received when data from a previous call to IcLibUpdateConfig or IcLibVerifyConfig is being discarded. For example, IC_CMD_DISCARD is returned to IcLibUpdateConfig when a user cancels during a dialog associated with IcLibUpadateConfig.

IC_ERROR_UNKNOWN_COMMAND should be returned if the library encounters an unknown IC_COMMAND type.

Use IcDialogConfig instead of the Windows API for dialog box manipulation since it uses HIC_CONFIG as a parameter.

IcLibVerifyConfig

Old Parameter	New Parameter
HWND hWnd	HIC_CONFIG hConfig ***Watch:*** *The first parameter is now HIC_CONFIG, not HWND.*
int TableNumber	UINT TableNumber
unsigned len	UINT len
enum IC_VERIFY Command	IC_VERIFY Command

Note: *The following updates are required for 3.0. Be sure the updates have been completed prior to setting the maximum version in the library's resource file to reflect INFOConnect 3.0.*

Two new IC_VERIFY types have been added in INFOConnect 3.0: IC_VER_UPGRADE and IC_VER_DELETE. The IC_VER_UPGRADE command tells the library to perform special upgrade processing and data conversions on the given buffer of data. The IC_VER_DELETE command tells the library that the given configuration data is about to be deleted. If the library returns a severe error, the data will not be deleted.

The IC_VER_NODISPLAY has been renamed as IC_VER_SAVE. This command tells the library that the configuration data is about to be saved. If the library returns a sever error, the data will not be saved.

IC_ERROR_UNKNOWN_COMMAND should be returned if the library encounters an unknown IC_VERIFY type.

IcLibPrintConfig

Old Parameter	New Parameter
int TableNumber	UINT TableNumber
int detail	IC_PRINT_DETAIL detail
unsigned len	UINT len
unsigned prlen	UINT prlen

IC_ERROR_UNKNOWN COMMAND should be returned if the library encounters an unknown IC_PRINT_DETAIL type.

Changes for Service Libraries

A service library that supports ICS 3.0 and defines a channel table must provide an interface (a control on the path dialog) which links a path to a channel. Also consider whether the table should be a custom table rather than a channel table. INFOConnect 3.0 will support a user interface to associate a path with a channel only for libraries that have a maximum version less than INFOConnect 3.0. This user interface will be removed in the next release of ICS.

Note: *Be sure the updates have been completed prior to setting the maximum version in the library's resource file to reflect INFOConnect 3.0.*

Running with Multiple Versions of INFOConnect

Libraries can be written to run with multiple versions of INFOConnect. Updating your library to a new version of the IDK will normally prevent your application from running with older versions of INFOConnect. See "Running with Multiple of INFOConnect" in Section 7, for instructions on how to build with a new IDK and continue to run with older versions of INFOConnect.

Converting From Release 1.0 to 2.0

This section describes the new features of INFOConnect Release 2.0 and identifies features that require source changes to existing INFOConnect applications and libraries developed with INFOConnect Release 1.0. Release 1.0 was delivered with Designer Workbench and is sometimes referred to as "Stage 0."

2.0 Features

Architecture Diagram

The INFOConnect architecture diagram has been updated to better illustrate the flow of data through the different layers. The Manager is more clearly shown as the mediator between the different layers. Also, notice that multiple Service Libraries (formerly Protocol-Modifiers) can now be stacked in a single path. This feature promotes the development of small, reusable components. The different APIs are no longer explicitly labeled in the diagram. The Accessories API has a few additions in release 2.0, but is basically unchanged. The Protocol-Modifier and Protocol-Interface APIs have been collapsed into a single Library API. In essence though, the architecture has not changed.

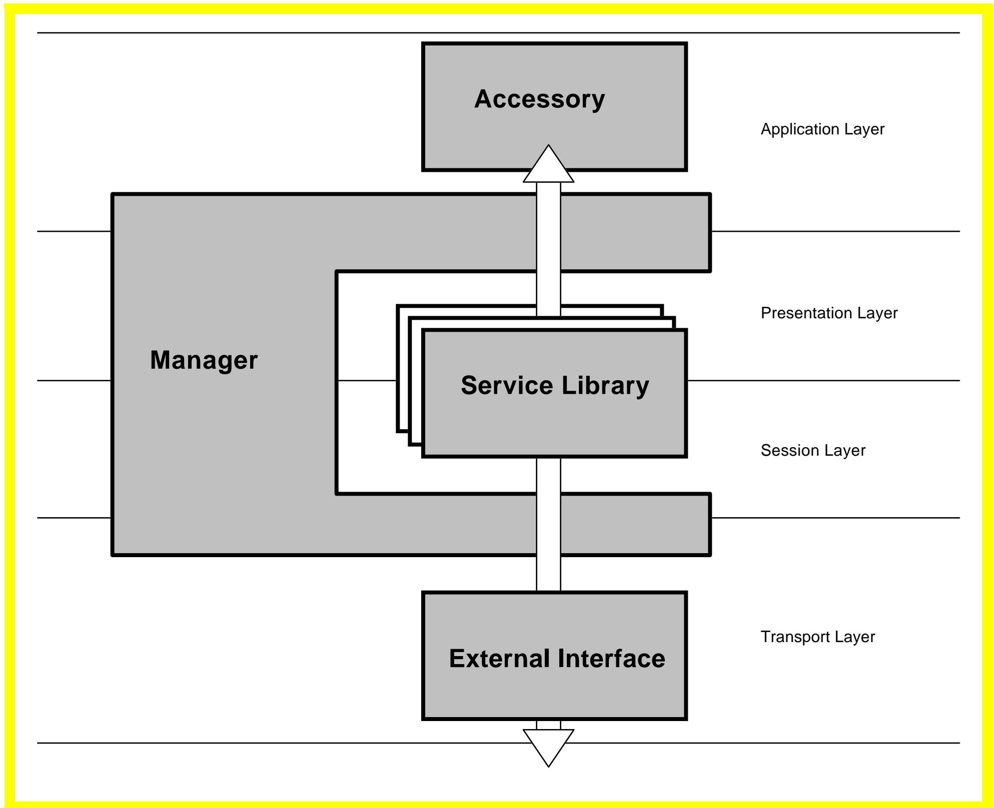


Figure 10–2. INFOConnect 2.0 Architecture

INFOConnect is now made up of several DLLs that are installed into the Windows SYSTEM directory:

CommMgr.dll	INFOConnect manager
IcUtil.dll	Miscellaneous support routines
IcDb.dll	Configuration database manager
IcMgrCfg.dll	Configuration/installation support routines

INFOConnect directories

Directory	Contents	Installation default
CodeDir	Standard INFOConnect libraries and accessories	c:\infoconn
DataDir	Configuration database, macro files, option files, and so on	c:\windows
Windows System Directory	INFOConnect Manager DLLs	c:\windows\system
-	3rd party accessories and libraries	defined in IcSetup.inf

Notice that you can now install your application or library in a directory separate from the standard INFOConnect files.

Terminology

Several new terms are introduced in INFOConnect 2.0 and a few have been renamed.

Service Library

Previously called protocol-modifiers, service libraries act like filters between the application and lower communication transports. Zero or more service libraries can exist in an INFOConnect path. Because service libraries are now optional, the PassThru protocol-modifier no longer exists.

External Interface Library (EIL)

External Interface Libraries, previously called protocol-interfaces, are adapters that provide a mapping between the INFOConnect architecture and externally provided communications components. Each INFOConnect path must contain one (and only one) EIL.

Multiplexing Library

Multiplexing libraries are EILs that provide multiple logical connections across a single connection.

Path

Paths are basically the same. A *path* consists of one external interface and zero or more service libraries. Previously, *paths* consisted of exactly one protocol-modifier and one protocol-interface.

Path Template

Users configure paths by choosing from the available path templates. Path templates define a group of libraries that function together. Templates simplify INFOConnect path configuration by alleviating users from having to know which libraries work together. Furthermore, they allow the development of many, small service libraries without overwhelming the user at configuration time with a long list of libraries. Library developers can define path templates to be installed by the INFOConnect installation utility.

Library API

The Protocol-Modifier API and Protocol-Interface API were collapsed into a single API called the Library API. This primarily requires simple name changes in existing libraries. There are a few new functions for building and processing configuration information.

Accessory IDs and Library IDs

Accessory and Library IDs allow different implementations of an accessory or library to be easily interchanged on the workstation. IDs permit a user to pick and choose from different INFOConnect components provided by different vendors. Accessories are invoked by other INFOConnect applications using their Accessory ID instead of a filename.

Channel Index

Channel indexes no longer exist. They were used to implement multiplexing capabilities (multiple logical connections across a single physical connection), but required careful coordination between the protocol-modifier and protocol-interface. Multiplexing is now supported such that service libraries and EILs that aren't multiplexers themselves are not required to do anything special to support a multiplexing library running in the same library stack.

INFOConnect Installation Manager

Instructions are now provided with the IDK to package your INFOConnect application or library for use with the INFOConnect Installation Manager. The Installation Manager installs your INFOConnect component on the workstation and provides a common look and feel across the INFOConnect program. Installation script files allow you to customize your application's installation.

Accessory IDs

An Accessory ID is a name provided during accessory installation. An INFOConnect application invokes an accessory with its Accessory ID rather than its directory and filename. Different implementations of the same accessory can be developed as long they use the same Accessory ID, but at any one time, only one implementation of an accessory is installed in the INFOConnect environment.

The *IDK Programming Reference Manual* has a complete list, but here is a sampling of the standard Accessory IDs:

INFOConn	The INFOConnect shell
MT	A Series MT emulator
ANSI	VT-220 style emulator
UTS60	UTS60 emulator
UTS60G	UTS60 graphics engine
PPT	Printer pass through

Accessory Window States

IcOpenAccessory and IcRunAccessory now support a ***-Wxy*** command line option to control the state of the accessory and its window when executed by the Manager. The accessory can be made an active or a background task, and its window can be brought up as normal, maximized, iconized, or hidden.

DosLink

DosLink is an INFOConnect solution that allows DOS applications to make connections to other computers using INFOConnect Connectivity Services. The DOS application must be running in a DOS window in enhanced mode Windows, therefore, this capability is only available on 386 class machines and higher.

Library Configuration

The majority of changes and enhancements that went into Release 2.0 involve the configuration of service libraries and external interface libraries.

Library configuration is based on tables. Configuration information for a library is organized into tables of rows and columns (records and fields). Tables can reference each other through link fields. The goal is to eliminate data redundancy. Rather than duplicating information in two places, information is stored in one place and links are established between tables.

Library API Control Flow Diagrams

As mentioned earlier, the Protocol-Modifier and Protocol-Interface APIs have been combined into a single Library API. This was a natural development due to the inherent similarities of the two APIs and the consequences of library stacking.

Refer to the updated control flow diagrams in Section 6, "A Closer Look at the INFOConnect Architecture."

Session and Channel Aliasing

Aliasing is a new feature provided for the convenience and efficiency of INFOConnect libraries. Without aliases, a library must constantly scan its own list of session and channel records for the INFOConnect session handle. Aliasing allows a library to eliminate this excessive searching. Aliasing can be ignored until an existing library has been converted and is up and running.

Tracing INFOConnect Activity

Activating the INFOConnect Trace Facility is now done through the Administration menu options rather than manually configuring paths with the TRACE library.

The generated file is now named Trace.log (instead of Trace.dbg) and is written to your DataDir directory which is normally c:\windows. New trace sessions are appended to Trace.log rather than rewriting over the previous trace, so you may need to occasionally erase or prune Trace.log.

The Path and Path Template configuration dialogs also provide a Trace check box that affects the Trace facility. See INFOConnect's on-line help for more information about using Trace.

INFOConn.ini

INFOConnect path configuration information was previously stored in INFOConn.ini. This information is now stored in a new file named INFOConn.cfg using a new format. One of the new DLLs, IcDb.dll, manages access to INFOConn.cfg. There is currently no utility for moving old paths stored in INFOConn.ini into INFOConn.cfg.

Running Old INFOConnect Modifiers and Interfaces

Old protocol-modifiers and protocol-interfaces will still work with INFOConnect 2.0. They must be reconfigured, though, because there is currently no utility for upgrading old paths stored in INFOConn.ini into INFOConn.cfg.

Changes Affecting Applications

Existing applications will run without change. However, several recommended changes are explained. Other new features of particular interest to application developers are also explained.

Running with Old Versions of INFOConnect

Updating your application to a new version of the IDK will normally prevent your application from running with older versions of INFOConnect. See "Running with Old Version of INFOConnect" in Sections 3, 4 and 5 for instructions on how to build with a new IDK and continue to run with older versions of INFOConnect.

CommMgr.lib Replaced by IcWin.lib

Note: *This change only affects Windows-specific applications. XVT users can ignore it.*

Update your makefiles to use IcWin.lib instead of CommMgr.lib when linking. INFOConnect now consists of several DLLs and IcWin.lib resolves references to the appropriate DLL.

HANDLE Versus HIC_SESSION

Note: *This change only affects Windows-specific applications. XVT users can ignore it.*

INFOConnect session handles are now properly defined using a new typedef, HIC_SESSION, instead of the generic HANDLE. For example, all function prototype parameter lists that pass session handles now use HIC_SESSION. Your application will run without this change, but you should make it now for future compatibility.

```
IC_RESULT FAR PASCAL IcXmt(HIC_SESSION session,
          HANDLE buffer,
          unsigned length);

IC_RESULT FAR PASCAL IcRcv(HIC_SESSION session,
          HANDLE buffer,
          unsigned length);

IC_RESULT FAR PASCAL IcLd(HIC_SESSION session,
          WORD which);
```

Also, instead of using NULL to test or set session handles, use NULL_HIC_SESSION.

New INFOConnect Events

The following INFOConnect events are new in Release 2.0. Existing applications are not required to recognize them, but may find them useful. They are documented in the *IDK Programming Reference Manual*.

Windows API

IC_LclResult	Generated when a call to IcLcl finally completes
IC_StatusResult	Generated when a call to IcSetStatus finally completes

XVT API

E_IC_LCL_RESULT	Generated when a call to ic_lcl finally completes
E_IC_STATUS_RESULT	Generated when a call to ic_set_status finally completes

Windows applications that choose to use these events must register the new message numbers along with the other INFOConnect messages like IC_SessionEstablished, IC_RcvDone, etc. See the sample program, IcWinApp.c, for all references to IC_LclResult and IC_StatusResult.

Freeing Buffers Immediately after Session Closure

Some applications waited to free datacomm buffers until the session closure event was returned to the application. This was to prevent errors during a small window of vulnerability caused by a protocol-modifier or protocol-interface inadvertently writing to a datacomm buffer associated with a recently closed session. Improvements in the INFOConnect Library API have closed this window of vulnerability and applications can safely free buffers immediately after closing a session.

New IC_STATUS_COMMMGR Statuses

As the INFOConnect Manager shuts down, it sends new status messages to indicate the different states during the termination process. Existing applications are not required to recognize these messages, but may find them useful. They are documented in the *IDK Programming Reference Manual*.

IC_COMMMGR_INITIALIZED
IC_COMMMGR_TERMINATED
IC_COMMMGR_QUERYEXIT
IC_COMMMGR_CANCELEXIT
IC_COMMMGR_EXIT

IcGetSessionName and IcGetPathName

Two functions have changed names and parameter lists. Also, the second parameter on these functions has changed from a handle to a long pointer. This avoids having to allocate an INFOConnect buffer to use these functions.

The old names are still supported. Existing applications don't have to change.

Windows API

Old name	New name
IcGetSessionName	IcGetSessionID
IcGetPathName	IcGetPathID

XVT API

Old name	New name
ic_get_session_name	ic_get_session_id
ic_get_path_name	ic_get_path_id

Example

Here are *before* and *after* code fragments from IcWinApp.c (a sample application in the IDK) showing the conversion of IcGetSessionName to IcGetSessionID.

Old code fragment using IcGetSessionName

```
void UpdateWindowName(HWND hWnd, LPSTR lpWindowName)
{
 /*
   Append the INFOConnect session name to the Window name
   and display it.
 */

 HANDLE hSessionName;
 LPSTR lpSessionName;

 hSessionName = IcAllocBuffer(IC_MAXSESSIONIDLEN);
 icerror = IcGetSessionName(s.hSession,
              hSessionName,
              IC_MAXSESSIONIDLEN);
 lpSessionName = IcLockBuffer(hSessionName);
 lstrcat(lpWindowName,(LPSTR)" - ");
 lstrcat(lpWindowName,lpSessionName);
 SetWindowText(hWnd,lpWindowName);
 IcUnlockBuffer(hSessionName);
 IcFreeBuffer(hSessionName);
}
```

Converted code using IcGetSessionID

```
void UpdateWindowName(HWND hWnd, LPSTR lpWindowName)
{
 /*
   Append the INFOConnect session name to the Window name
   and display it.
 */

 char SessionName[IC_MAXSESSIONIDLEN+1];

 icerror = IcGetSessionID(s.hSession,
             SessionName,
             sizeof(SessionName));
 lstrcat(lpWindowName,(LPSTR)" - ");
 lstrcat(lpWindowName,SessionName);
 SetWindowText(hWnd,lpWindowName);
}
```

IcOpenAccessory and ic_open_accessory

If your application uses IcOpenAccessory or ic_open_accessory to invoke any of the standard INFOConnect accessories be sure to use the appropriate Accessory ID and not a hard coded filename. This prevents dependencies in your application on knowing where INFOConnect accessories are installed.

For example, to invoke the A Series MT Emulator:

```
icerror=IcOpenAccessory(hWnd,
          "MT",  //accessory ID
          NULL,
          "mypath",
          &sinfo,
          &hSession);
```

Don't do this:

```
icerror=IcOpenAccessory(hWnd,
          "c:\infoconn\mt.exe",//WRONG!
          NULL,
          "mypath",
          &sinfo,
          &hSession);
```

Initial window state of the accessory

You can now control the initial state of the accessory's window using the -W option. Available states are: normal, maximized, iconized, hidden, active, background. See the *IDK Programming Reference Manual* for details.

To invoke a UTS Emulator with a maximized window:

```
icerror=IcOpenAccessory(hWnd,
          "UTS", //accessory ID
          "-W ma",//maximized, active window
          "mypath",
          &sinfo,
          &hSession);
```

New Requirements for Accessories

Release 2.0 adds several requirements for INFOConnect accessories: scanning the command line for the -K option and adding an INFOConnect resource to the resource (.RCA) file. See Section 6 of the *IDK Programming Reference Manual*, "ICS Accessory Definition", for a complete list of requirements. You can ignore this section if your application is not invoked by other INFOConnect applications as an accessory. You can test your accessory using the IcOpenAc sample program provided in the IDK.

Initial window state of an invoked accessory

IcOpenAccessory and IcRunAccessory accept a -W option to allow the calling application to control the initial state of the invoked accessory's window: normal, maximized, iconized, hidden, active, background, etc. The Manager relies on accessories to honor the standard *nCmdShow* parameter passed to them by Windows at WinMain. After creating their window, accessories should call ShowWindow and pass along the *nCmdShow* parameter. The following code fragment is taken from the IDK sample program, IcWinApp.

```
int PASCAL WinMain(HANDLE hInstance,
        HANDLE hPrevInstance,
        LPSTR lpCmdLine,
        int nCmdShow)
{
 MSG msg;

 if (!hPrevInstance)
  if (!InitApplication(hInstance))
   return (FALSE);

 if (!InitInstance(hInstance, nCmdShow, lpCmdLine))
  return (FALSE);

 while (GetMessage(&msg, NULL, NULL, NULL)) {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
 }
 return (msg.wParam);
}

BOOL InitInstance(HANDLE hInstance,
        int nCmdShow,
|       LPSTR lpCmdLine)
{
 hInst = hInstance;

 hWnd = CreateWindow(......);

 if (!hWnd)
  return (FALSE);

 ShowWindow(hWnd, nCmdShow);
 UpdateWindow(hWnd);   /* Sends WM_PAINT message */
```

-K command line option

Accessories should parse the command line for the Accessory ID parameter and incorporate it in some fashion in the window's title bar. This gives the user a better indication about the relationship between the invoking application and the invoked accessory. See the code fragment from IcWinApp below.

AccessoryID registration

Release 2.0 accessories must contain an INFOConnect RCDATA section in the resource file (.RC) and some accompanying text strings in the STRINGTABLE.

To illustrate accessory registration, here are some code fragments from IcWinApp, a sample INFOConnect accessory in the IDK.

IcWinApp.h

```
#define APPNAME      IcWinApp
#define QAPPNAME     "IcWinApp"
#define QMARKETINGNAME "INFOConnect Sample Accessory"
#define QVENDOR      "Unisys"

#define IC_ACCESSORYID   101
#define IC_ACCESSORYDESC  102
#define IC_VENDOR      103
```

IcWinApp.rc

```
#include <windows.h>
#include <icdef.h>
#include <icdict.h>
#include "icwinapp.h"

INFOConnect RCDATA
BEGIN
  IC_VERSION,
  IC_REVISION,
  IC_ACCESSORY,
  IC_HEADER_SIZE,
  0,
  IC_ACCESSORYID, /* Calling apps use this name to invoke your accessory */
  IC_ACCESSORYDESC,
  IC_VENDOR,
  0,0,0,0
END

STRINGTABLE DISCARDABLE
BEGIN
  IC_ACCESSORYID,   QAPPNAME
  IC_ACCESSORYDESC, QMARKETINGNAME
  IC_VENDOR,      QVENDOR
END
```

IcWinApp.c

```
char sAccessoryID[IC_MAXACCESSORYIDSIZE] = {QAPPNAME};

icerror = GetCmdlineOption(lpCmdLine, 'K', "",
         sAccessoryID, sizeof(sAccessoryID));
icerror = IcRegisterAccessory(sAccessoryID, 0, &iccontext);
if (icerror != IC_OK) {
  HandleIcError(hWnd, NULL, NULL, icerror);
  return(FALSE);
}
```

Changes Affecting Libraries

Existing protocol-modifiers and protocol-interfaces will run without change. However, it is strongly recommended that you update them to release 2.0.

COMMMGR.LIB Replaced by ICWIN.LIB

Update your makefiles to use IcWin.lib instead of CommMgr.lib when linking. INFOConnect now consists of several DLLs and IcWin.lib resolves references to the appropriate DLL.

New Header (.H) Files

The INFOConnect header files have been reorganized.

Include the following files in the main source file (.C)

```
#include <windows.h>
#include <iclib.h>
#include <icproto.h>
```

Include the following files in the resource file (.RC)

```
#include <windows.h>
#include <icdef.h>
#include <icdict.h>
```

IcAssert.h

The assert macro is a useful debugging tool that many of the 1.0 sample applications used, but none of the sample protocol-modifiers and interfaces used it because the standard assert macro did not work with DLLs. INFOConnect now provides a version of assert that can be used from a DLL and most of the sample libraries use it. The INFOConnect assert macro is defined in :

```
#include <icassert.h>
```

Function Name Changes

Most of the following name changes can be made using an editor with a global search and replace command.

IcP- functions changed to IcLib

All of the required callback function names in your library have been renamed with the *IcLib* prefix instead of *IcPm* or *IcPi*. A few functions differ by more than just the prefix. Although it is actually the ordinal number of your callback function (rather than the name) that is critical, we suggest changing your function names to avoid confusion.

Old name	New name
IcP-ConfigPath	IcLibUpdateConfig
IcP-DeselectChannel	IcLibCloseSession
IcP-Event	IcLibEvent
IcP-GetString	IcLibGetString
IcP-IdentifySession	IcLibIdentifySession
IcP-Install	IcLibInstall
IcP-Lcl	IcLibLcl
IcP-Rcv	IcLibRcv
IcP-SelectChannel	IcLibOpenSession
IcP-SetInfo	IcLibGetSessionInfo
IcP-SetResult	IcLibSetResult
IcP-Terminate	IcLibTerminate
IcP-Xmt	IcLibXmt

The following functions are new and have no counterpart in release 1.0, but are shown here for completeness. They are described in more detail elsewhere.

New callback functions

IcLibVerifyConfig
IcLibPrintConfig
IcLibOpenChannel
IcLibCloseChannel

Updated .DEF file

These new names show up in the module definition (.DEF) file which must be updated for your library. Here is the updated .DEF file from the SERVICE sample library in the IDK.

```
;************************************************************/
;* SERVICE.DEF                    */
;************************************************************/

LIBRARY service
DESCRIPTION 'Sample INFOConnect Service Library'
EXETYPE WINDOWS
STUB 'WINSTUB.EXE'
CODE MOVEABLE DISCARDABLE PRELOAD

;DLLs require DATA SINGLE because there is only one instance
DATA SINGLE PRELOAD MOVEABLE
HEAPSIZE 4096
EXPORTS
  IcLibUpdateConfig  @1
  IcLibCloseSession  @2
  IcLibEvent      @3
  IcLibGetString    @4
  IcLibIdentifySession @5
  IcLibInstall     @6
  IcLibLd        @7
  IcLibRcv       @8
  IcLibOpenSession   @9
  IcLibGetSessionInfo @10
  IcLibSetResult    @11
  IcLibTerminate    @12
  IcLibXmt       @13
  IcLibVerifyConfig  @14
```

```
    IcLibPrintConfig   @15
    IcLibOpenChannel   @16
    IcLibCloseChannel  @17

; Don't forget to include callback functions in this list

    cbChannelConfigDlg  @19
    cbPathConfigDlg     @20
    cbAboutDlg          @21

    WEP           @22 RESIDENTNAME
```

Name changes affecting only protocol-modifiers

The following functions that are called by protocol-modifiers have been renamed. The function parameters are unchanged. Update your library to call the new names instead of the old.

Old name	New name
IcEvent	IcSendEvent
IcPiSetResult	IcSetResult
IcPiXmt	IcLibraryXmt
IcPiRcv	IcLibraryRcv
IcPiLcl	IcLibraryLcl

For example, here are two code fragments, one from an old release 1.0 protocol-modifier and the other from an updated release 2.0 service library. The updated library calls IcLibraryRcv instead of IcPiRcv.

Release 1.0 fragment

```
IC_RESULT FAR PASCAL IcPmRcv(HANDLE hSession, /* OLD */
            HANDLE buffer,
            unsigned length )
{
 ...
 return IcPiRcv(hSession, buffer, length);
}
```

Release 2.0 fragment

```
IC_RESULT FAR PASCAL IcLibRcv (HIC_SESSION hSession, /* NEW */
          HANDLE   buffer,
          unsigned  length )
{
 ...
 return IcLibraryRcv(hSession, buffer, length);
}
```

Name changes affecting only protocol-interfaces

The following functions that are called by protocol-interfaces have been renamed. Update your library to call the new names instead of the old.

Old name	New name
IcPmEvent	IcSendEvent

HANDLE Versus HIC_SESSION

INFOConnect session handles are now more properly defined using a new typedef, HIC_SESSION, instead of the generic HANDLE typedef. All function prototype parameter lists that pass session handles now use HIC_SESSION. Existing code that uses HANDLE will still compile, but new code should use the HIC_SESSION typedef.

Old protocol-modifier function using HANDLE

```
IC_RESULT FAR PASCAL IcPmRcv (HANDLE  hSession,
          HANDLE  buffer,
          unsigned length )
```

New service library function using HIC_SESSION

```
IC_RESULT FAR PASCAL IcLibRcv (HIC_SESSION hSession,
          HANDLE   buffer,
          unsigned  length )
```

IcLibInstall

IcLibInstall (formerly IcP-Install) and IcLibTerminate were previously not called during path configuration, but now they are. If this affects your library (it will not affect a lot of libraries), it will just mean moving some code from IcLibInstall to either IcLibOpenChannel or IcLibOpenSession.

Library Configuration

The work required to implement library configuration is broken into the following steps:

Table design:	Design the tables
Table description:	Update the library's resource file (.RC) with your table descriptions
Table processing:	Update the library source file (.C) with the configuration-related callback functions

See the "Configuration Management" discussion in Section 7, Writing INFOConnect Libraries for Windows 3.x.

Session Establishment

IcP-SelectChannel is replaced by IcLibOpenChannel and IcLibOpenSession. The code that is currently in IcP-SelectChannel must be distributed between IcLibOpenChannel and IcLibOpenSession.

Libraries without channels

For those libraries that don't use channels (the most common case) you can just rename your IcP-SelectChannel function as IcLibOpenSession. Add a skeleton function for IcLibOpenChannel that just returns IC_OK. This is necessary because IcLibOpenChannel is still called one time with a NULL channel and NULL buffer for libraries that don't define channels.

```
IC_RESULT FAR PASCAL IcLibOpenChannel
        (HIC_CHANNEL  hChannel,
         void FAR *  lpConfigBuf,
         unsigned    len,
         BOOL        bVerify,
         LPHIC_CHANNEL lphChannel)
{
  return IC_OK;
}
```

Libraries with channels

Libraries that use channels must split the code in IcP-SelectChannel between the new IcLibOpenChannel and IcLibOpenSession. IcLibOpenSession is called whenever a session is opened. IcLibOpenChannel is called before IcLibOpenSession, but only for those sessions that are using a new, unused channel.

IcP-ConfigPath Replacements

The old IcP-ConfigPath routine has been replaced by three new callback functions: IcLibUpdateConfig, IcLibVerifyConfig and IcLibPrintConfig. These functions are discussed in Section 7, "Writing INFOConnect Libraries for Windows 3.x."

Session and Channel Aliases

Aliases are provided as a convenience and performance boost to library execution. They are not required , therefore, you should probably ignore aliases and get your library running, then come back and enable aliases. To disable session and channel aliasing, simply ignore the **lph~Alias** pointers on the IcLibOpenChannel and IcLibOpenSession functions. The fields that these pointers refer to have been properly initialized.

```
IC_RESULT FAR PASCAL IcLibOpenChannel
        (HIC_CHANNEL hChannel,
        void FAR * lpConfigBuf,
        unsigned  len,
        BOOL    bVerify,
        LPHIC_CHANNEL lphChannelAlias)

IC_RESULT FAR PASCAL IcLibOpenSession
        (HIC_SESSION hSession,
        HIC_CHANNEL hChannel,
        void FAR * lpConfigBuf,
        unsigned  len,
        BOOL    bVerify,
        LPHIC_SESSION lphSessionAlias)
```

Aliasing is covered in detail in Section 7, "Writing INFOConnect Libraries for Windows 3.x."

Index

A

accessory
  definition, 2-7
accessory API
  definition, 2-7
  events, 2-19
  functions, 2-16
accessory IDs
  definition, 10-27
accessory management
  functions, 2-18
accessory window states
  definition, 10-27
advanced error handling
  Windows applications, 3-36
  XVT applications, 4-26
advanced status handling
  Windows applications, 3-36
  XVT applications, 4-26
AIL
  definition, 2-7, 10-4
aliasing
  library, 7-53
allocating buffers
  Windows applications, 3-5
  XVT applications, 4-3
alphabetical list
  IDK package, 1-3
  IDK samples package, 1-9
app.copy.appstuff
  installation, 9-12
app.copy.appstuff section
  syntax, 9-27
app.copy.noremove section
  installation, 9-12
  syntax, 9-27
app.copy.publish section
  syntax, 9-27
app.copy.standalone section
  syntax, 9-27
app.copy.subscribe section
  syntax, 9-27
application
  definition, 2-7
Application Interface Library, (*See* AIL)
architecture
  diagram, 2-5, 10-1
  relationship to session, 2-13
architecture diagram
  INFOConnect 2.0, 10-23
  INFOConnect 3.0, 10-1
assert macro
  debugging applications, 8-5

B

basic error handling
  Windows applications, 3-17
  XVT applications, 4-12
basic procedures
  DosLink applications, 5-3
  Windows applications, 3-2
  XVT applications, 4-2
basic session management
  functions, 2-16
basic status handling
  Windows applications, 3-22
  XVT applications, 4-16
bootstrap
  installation, 9-11
Borland C
  compatible makefiles, 1-18
  make command line, 1-21
buffers
  inter-application, 2-15
  intra-application, 2-15

C

calling INFOConnect accessories
Windows applications, 3-41
XVT applications, 4-34
canceling pending requests
DosLink applications, 5-19
Windows applications, 3-32
XVT applications, 4-24
checklist
system verification (for windows), 1-19
choosing between
library, 7-2
closer look
manager components, 6-1
manager components diagram, 6-2
closing a session
DosLink applications, 5-17
Windows applications, 3-26
XVT applications, 4-19
codedir
definition, 9-3
coding mistakes
debugging, 8-7
communicating with applications
library, 7-63
compiling
library, 7-73
Windows applications, 3-44
XVT applications, 4-37
compiling and linking
DosLink applications, 5-26
complete deinstallation
definition, 9-3
component numbers
definition, 10-7
configuration API
definition, 10-5
configuration database
definition, 9-4
configuration management
library, 7-4
configurator
definition, 2-10, 10-5
context
definition, 2-15
converting from release 1.0 to 2.0
2.0 architecture diagram, 10-23
2.0 terminology, 10-25
aliases, 10-28, 10-44
CommMgr.lib replaced by IcWin.lib, 10-30, 10-38
freeing buffers immediately after session closure, 10-31
function name changes, 10-39
HANDLE versus HIC_SESSION, 10-30, 10-42
IcGetSessionName and IcGetPathName, 10-32
IcLibInstall, 10-43
IcOpenAccessory and Ic_open_accessory, 10-34
IcP -ConfigPath replacements, 10-44
INFOConn.ini, 10-29
library API control flow diagrams, 10-28
library configuration, 10-43
library session establishment, 10-43
new IC_STATUS_COMMMGR Statuses, 10-32
new INFOConnect events, 10-31
new library header files, 10-38
new requirements for accessories, 10-35
running old INFOConnect modifiers and interfaces, 10-29
tracing INFOConnect activity, 10-28

converting from release 2.0 to 3.0
 3.0 architecture diagram, 10-1
 3.0 manager components, 10-3
 changes affecting applications
  CommMgr.lib removed, 10-14
  component numbers, 10-12
  defining component numbers, 10-12
  IcRegisterMsgSession, 10-15
  ICXVTWIN tag, 10-15
  INF file version information, 10-11
  INFOConnect RCDATA version information, 10-9
  INFOConnect version identification, 10-8
  managing branded component numbers, 10-13
  new IC_STATUS_COMMMGR status, 10-15
  new IC_STATUS_TRANS statuses, 10-15
  new XVT link libraries, 10-14
  running with old versions of INFOConnect, 10-16
 changes affecting libraries
  changes for IcLib functions, 10-17
  component numbers, 10-12
  defining component numbers, 10-12
  INF file version information, 10-11
  INFOConnect RCDATA version information, 10-9
  INFOConnect version identification, 10-8
  library API changes, 10-16
  managing branded component numbers, 10-13
  running with multiple versions of ICS, 10-22
  service library specific, 10-22
cooperative application
 definition, 2-6

D

data compression and error detection
 Windows applications, 3-39
 XVT applications, 4-32
data section
 syntax, 9-18
datadir
 definition, 9-4
debugging
 adding trace information to the trace log, 8-3
 common coding mistakes, 8-7
 diagnostic library, 8-5
 INFOConnect -d debug option, 8-6
 source level debugging, 8-7
 tracing INFOConnect datacomm activity, 8-1
 using assert macro in applications, 8-5
 using debug version of Windows, 8-7
 Windows 3.1 issues, 8-6
definition
 accessory, 2-7
 accessory API, 2-7
 accessory IDs, 10-27
 accessory window states, 10-27
 AIL, 2-7, 10-4
 application, 2-7
 Application Interface Library, 2-7, 10-4
 codedir, 9-3
 complete deinstallation, 9-3
 component numbers, 10-7
 configuration API, 10-5
 configuration database, 9-4
 configurator, 2-10, 10-5
 context, 2-15
 cooperative application, 2-6
 datadir, 9-4
 DosLink, 10-28
 DosLink API, 2-7
 EIL, 2-8
 exit hook library, 2-9

extended status buffer, 10-8
External Interface Library, 2-8
hook library, 2-9, 10-4
IC_RESULT, 2-15
ICS version numbers, 10-5
IIL, 2-8, 10-4
INF file, 9-3
INFOConnect library, 2-10
INFOConnect packages window, 9-3
install shell, 9-2
installation database, 9-4, 9-5
installation manager, 9-2, 10-27
inter-application buffers, 2-15
Interprocess Interface Library, 2-8, 10-4
intra-application buffers, 2-15
library channel, 2-14
library configuration, 10-28
manager components, 2-6
multiplexing library, 2-9
package, 9-2
path, 2-2, 2-11
path template, 2-14
publish installation, 9-3
quick config library, 2-9
quick configuration library, 2-9, 9-4
quick configuration manager, 9-4
registration, 9-4
sequence number, 9-5
service library, 2-8
session, 2-2, 2-12
session information block, 2-15
shell, 2-10, 10-5
stack interface library, 2-9, 10-4
standalone installation, 9-2
subscribe installation, 9-3
switching library, 2-9, 10-4
XVT, 2-7

design issues
library, 7-2

destination
installation, 9-11

diagnostic library
debugging, 8-5

dialog section
syntax, 9-19

disks section
syntax, 9-20

distribution media
creating, 9-9
file compression, 9-9
icsetup.inf, 9-9
install.exe, 9-9
root directory contents, 9-9

DosLink
definition, 10-28

DosLink API
definition, 2-7

DosLink applications
basic procedures, 5-3
canceling pending requests, 5-19
closing a session, 5-17
compiling and linking, 5-26
DosLink solution, closer look, 5-23
error handling, 5-15
handling data communications errors, 5-20
initializing ICS, 5-3
introduction, 5-1
opening a session, 5-4
receiving a buffer, 5-11
running with old versions of ICS, 5-21
samples
IcBDrive, 5-30
IcDosApp, 5-29
transmitting a buffer, 5-8
using datacomm buffers, 5-13

DosLink solution, closer look
DosLink applications, 5-23

E

EIL, (*See also* library)
definition, 2-8

encoding and decoding
Windows applications, 3-38
XVT applications, 4-31

environment variables
INCLUDE, 1-19
installation, 1-17

error events
ICS control flow, 6-11

error handling
DosLink applications, 5-15
functions, 2-17

error management
library, 7-15
events
accessory API, 2-19
exit hook library
definition, 2-9
extended status buffer
description, 10-8
external interface library, (*See* EIL), (*See also* library)

F

file compression
package diskettes, 9-9
filtering service
libraries, 7-24
functions
accessory API, 2-16
accessory management, 2-18
basic session management, 2-16
error handling, 2-17
memory management, 2-18
path management, 2-17

G

generating errors
library, 7-57

H

handling data communications errors
DosLink applications, 5-20
Windows applications, 3-33
XVT applications, 4-25
hook library
definition, 2-9, 10-4

I

IC_RESULT
definition, 2-15
IC_STATUS_BUFFER
description, 10-8
IC_STATUS_BUFFER extended status
library, 7-17
IcLibCloseChannel
library, 7-35
IcLibCloseSession
library, 7-37
IcLibEvent
converting from release 2.0 to 3.0, 10-18
library, 7-42
IcLibGetSessionInfo
library, 7-46
IcLibGetString
converting from release 2.0 to 3.0, 10-20
library, 7-48
IcLibIdentifySession
library, 7-51
IcLibInstall
library, 7-26
IcLibLcl
converting from release 2.0 to 3.0, 10-19
library, 7-40
IcLibOpenChannel
converting from release 2.0 to 3.0, 10-17
library, 7-34
IcLibOpenSession
converting from release 2.0 to 3.0, 10-18
library, 7-36
IcLibPrintConfig
converting from release 2.0 to 3.0, 10-22
library, 7-33
IcLibRcv
converting from release 2.0 to 3.0, 10-19
library, 7-38
IcLibSetResult
converting from release 2.0 to 3.0, 10-20
library, 7-41
IcLibTerminate
library, 7-28

IcLibUpdateConfig
- converting from release 2.0 to 3.0, 10-20
- library, 7-29

IcLibVerifyConfig
- converting from release 2.0 to 3.0, 10-21
- library, 7-31

IcLibXmt
- converting from release 2.0 to 3.0, 10-19
- library, 7-38

ICS
- application's perspective of, 2-2
- architecture, 2-5
- purpose, 2-1

ICS control flow
- error events, 6-11
- open session request, 6-4
- receive request, 6-8
- status events, 6-11
- transmit request, 6-8

ICS version numbers
- definition, 10-5
- library, 7-19

icxvtmod
- installation, 1-22

IDK package
- alphabetical list, 1-3

IDK samples package
- alphabetical list, 1-9

IIL
- definition, 2-8, 10-4

INCLUDE
- environment variable, 1-19

INF file
- accessory type section, 9-28
- app.copy.appstuff section, 9-27
- app.copy.noremove section, 9-27
- app.copy.publish section, 9-27
- app.copy.standalone section, 9-27
- app.copy.subscribe section, 9-27
- bitmap type section, 9-28
- data section, 9-18
- definition, 9-3
- dialog section, 9-19
- disks section, 9-20
- driver type section, 9-29
- font type section, 9-29
- help type section, 9-29
- library type section, 9-29
- needed.space section, 9-26
- noremove type section, 9-29
- package section, 9-21
- progman.groups section, 9-23
- publish.progman.groups section, 9-23
- purvey type section, 9-30
- readme type section, 9-30
- sample, 9-7, 9-34, 9-36
- standalone.progman.groups section, 9-23
- system type section, 9-30
- template type section, 9-30
- undefined type section, 9-30
- windows type section, 9-30
- winpurvey type section, 9-30
- writing, 9-6

INF script file version information
- library, 7-21

INF syntax
- introduction, 9-16

INFOConnect -d debug option
- debugging, 8-6

INFOConnect 2.0
- architecture diagram, 10-23

INFOConnect 3.0
- architecture diagram, 10-1
- manager components, 10-3

INFOConnect Connectivity Services, (*See* ICS)

INFOConnect header files
- library, 7-84

INFOConnect library
- definition, 2-10

INFOConnect packages window
- definition, 9-3

INFOConnect RCDATA version information
- library, 7-23
- Windows applications, 3-47

initializing ICS
- DosLink applications, 5-3
- Windows applications, 3-2
- XVT applications, 4-2

install shell
- definition, 9-2

installation
- app.copy.appstuff, 9-12
- app.copy.noremove section, 9-12
- bootstrap, 9-11
- copying, 9-12
- deinstallation flow, 9-15
- destination, 9-11
- environment variables, 1-17
- icxvtmod, 1-22
- installation flow, 9-10
- installing the IDK, 1-17
- package check, 9-12
- package contents, 1-3
  - IDK package, 1-3
  - IDK sample package, 1-9
- program group, 9-13
- quick config, 9-14
- release notes, 9-10
- starting, 9-10
- system requirements, 1-1
- Windows 3.0 SDK, 1-19
- windows platform, 1-1
- XVT, 1-22

installation database
- definition, 9-4, 9-5

installation manager
- definition, 9-2, 10-27

installation script file, (*See* INF file)

installing the IDK
- installation, 1-17

inter-application buffers
- definition, 2-15

Interprocess Interface Library, (*See* IIL)

intra-application buffers
- definition, 2-15

introduction
- DosLink applications, 5-1

L

library
- aliasing, 7-53
- choosing between, 7-2
- communicating with applications, 7-63
- compiling, 7-73
- configuration management, 7-4
- design issues, 7-2
- error management, 7-15
- filtering service libraries, 7-24
- generating errors, 7-57
- IC_STATUS_BUFFER extended status, 7-17
- IcLibCloseChannel, 7-35
- IcLibCloseSession, 7-37
- IcLibEvent, 7-42
- IcLibGetSessionInfo, 7-46
- IcLibGetString, 7-48
- IcLibIdentifySession, 7-51
- IcLibInstall, 7-26
- IcLibLcl, 7-40
- IcLibOpenChannel, 7-34
- IcLibOpenSession, 7-36
- IcLibPrintConfig, 7-33
- IcLibRcv, 7-38
- IcLibSetResult, 7-41
- IcLibTerminate, 7-28
- IcLibUpdateConfig, 7-29
- IcLibVerifyConfig, 7-31
- IcLibXmt, 7-38
- ICS version numbers, 7-19
- INF script file version information, 7-21
- INFOConnect header files, 7-84
- INFOConnect RCDATA version information, 7-23
- library checklist, 7-72
- library table serial numbers, 7-23
- linking, 7-86
- list of required functions, 6-3
- modifying global variables, 7-63
- on-line help, 7-61
- requesting session termination, 7-60
- required IcLib functions, 7-26
- resource files, 7-75
- running with multiple versions of ICS, 7-71
- samples
  - CoupleS, 7-90
  - IcStack2, 7-94
  - Intrface, 7-95
  - PS2TTY, 7-89
  - Reflect, 7-93
  - Service, 7-92
- session and channel aliasing, 7-53
- session and channel runtime record layout, 7-13
- session attributes, 7-3
- status management, 7-16

status messages, 7-63
structure, 6-3
system timers, 7-68
table
description, 7-8
design, 7-6
processing, 7-12
tracing, 7-70
version control, 7-19
Windows 3.1 version information resource, 7-22
Windows 3.x issues, 7-25
Windows DLL requirements, 7-52
library channel
definition, 2-14
library checklist
library, 7-72
library configuration
definition, 10-28
library table serial numbers
library, 7-23
linker
Microsoft C segmented, 1-19
linking
library, 7-86
Windows applications, 3-51
XVT applications, 4-40

M

makefiles
Borland C compatible, 1-18
Microsoft C compatible, 1-18
making your application an INFOConnect accessory
Windows applications, 3-42
XVT applications, 4-34
manager components
closer look, 6-1
definition, 2-6
INFOConnect 3.0, 10-3
manager components diagram
closer look, 6-2
memory management
functions, 2-18
Microsoft C
compatible makefiles, 1-18
make command line, 1-21
OLDNAMES, 1-20
unresolved external references, 1-20
modifying global variables
library, 7-63
multiplexing library
definition, 2-9

N

needed.space section
syntax, 9-26

O

OLDNAMES
Microsoft C, 1-20
on-line help
library, 7-61
open session request
ICS control flow, 6-4
opening a session
DosLink applications, 5-4
Windows applications, 3-5
XVT applications, 4-3

P

package
definition, 9-2
package check
installation, 9-12
package diskettes
creating, 9-9
file compression, 9-9
icsetup.inf, 9-9
install.exe, 9-9
root directory contents, 9-9
package section
syntax, 9-21

packaging
deinstallation flow, 9-15
installation flow, 9-10
overview, 9-1
sample INF file, 9-7, 9-34, 9-36
writing a script file, 9-6
path
definition, 2-2, 2-11
relationship to session, 2-12
path management
functions, 2-17
path template
definition, 2-14
procedures for INFOConnect accessories
Windows applications, 3-41
XVT applications, 4-33
progman.groups section
syntax, 9-23
program group
installation, 9-13
publish installation
definition, 9-3
publish.progman.groups section
syntax, 9-23

Q

quick config
installation, 9-14
quick configuration, (*See* Quick Config)
quick configuration library
definition, 2-9, 9-4
quick configuration manager
definition, 9-4

R

receive request
ICS control flow, 6-8
receiving a buffer
DosLink applications, 5-11
Windows applications, 3-12
XVT applications, 4-8
registration
definition, 9-4
release notes
installation, 9-10
requesting session termination
library, 7-60
required IcLib functions
library, 7-26
resource files
library, 7-75
Windows applications, 3-46
XVT applications, 4-39
running with multiple versions of ICS
library, 7-71
running with old versions of ICS
DosLink applications, 5-21
Windows applications, 3-39
XVT applications, 4-32

S

Samples
library
CoupleS, 7-90
IcStack2, 7-94
PS2TTY, 7-89
Reflect, 7-93
Service, 7-92
samples
DosLink applications
IcBDrive, 5-30
IcDosApp, 5-29
library
Intrface, 7-95
Windows applications
CoupleW, 3-76
IcWinApp, 3-53
XVT applications
Couple, 4-46
IcOpenAc, 4-48
IcXvtApp, 4-44
script file, (*See* INF file)
sample, 9-7, 9-34, 9-36
writing, 9-6

section
app.copy.appstuff, 9-12, 9-27
app.copy.noremove, 9-12, 9-27
app.copy.publish, 9-27
app.copy.standalone, 9-27
app.copy.subscribe, 9-27
data, 9-18
dialog, 9-19
disks, 9-20
needed.space, 9-26
package, 9-21
progman.groups, 9-23
publish.progman.groups, 9-23
standalone.progman.groups, 9-23
sequence number
definition, 9-5
service library, (*See also* library)
definition, 2-8
session
definition, 2-2, 2-12
relationship to architecture, 2-13
relationship to path, 2-12
session and channel aliasing
library, 7-53
session and channel runtime record layout
library, 7-13
session attributes
library, 7-3
session information block
definition, 2-15
shell
definition, 2-10, 10-5
SL, (*See* service library), (*See also* library)
stack interface library
definition, 2-9, 10-4
standalone installation
definition, 9-2
standalone.progman.groups section
syntax, 9-23
status events
ICS control flow, 6-11
status management
library, 7-16
status messages
library, 7-63
subscribe installation
definition, 9-3
switching library
definition, 2-9, 10-4
system requirements
installation, 1-1
system timers
library, 7-68
system verification checklist (for windows), 1-19

T

table, library
description, 7-8
design, 7-6
processing, 7-12
terminating your application
Windows applications, 3-28
XVT applications, 4-20
terminology
accessory, 2-7
accessory API, 2-7
AIL, 2-7, 10-4
application, 2-7
Application Interface Library, 2-7, 10-4
codedir, 9-3
complete deinstallation, 9-3
component numbers, 10-7
configuration API, 10-5
configuration database, 9-4
configurator, 2-10, 10-5
context, 2-15
cooperative application, 2-6
datadir, 9-4
DosLink API, 2-7
EIL, 2-8
exit hook library, 2-9
External Interface Library, 2-8
hook library, 2-9, 10-4
IC_RESULT, 2-15
ICS version numbers, 10-5
IIL, 2-8, 10-4
INF file, 9-3
INFOConnect library, 2-10
INFOConnect packages window, 9-3
install shell, 9-2
installation database, 9-4, 9-5
installation manager, 9-2
inter-application buffers, 2-15

Interprocess Interface Library, 2-8, 10-4
intra-application buffers, 2-15
library channel, 2-14
manager components, 2-6
multiplexing library, 2-9
package, 9-2
path, 2-11
path template, 2-14
publish installation, 9-3
quick config library, 2-9
quick configuration library, 2-9, 9-4
quick configuration manager, 9-4
registration, 9-4
sequence number, 9-5
service library, 2-8
session, 2-12
session information block, 2-15
shell, 2-10, 10-5
stack interface library, 2-9, 10-4
standalone installation, 9-2
subscribe installation, 9-3
switching library, 2-9, 10-4
XVT, 2-7

trace
adding trace information to the trace log, 8-3
function descriptions, 10-7
IcMgrTraceBuffer, 8-3
IcMgrTraceResult, 8-3

tracing
library, 7-70

tracing INFOConnect datacomm activity
debugging, 8-1

transmit request
ICS control flow, 6-8

transmitting a buffer
DosLink applications, 5-8
Windows applications, 3-8
XVT applications, 4-6

type section
accessory, 9-28
bitmap, 9-28
driver, 9-29
font, 9-29
help, 9-29
library, 9-29
noremove, 9-29
purvey, 9-30
readme, 9-30
system, 9-30
template, 9-30
undefined, 9-30
windows, 9-30
winpurvey, 9-30

U

unresolved external references
Microsoft C, 1-20

using datacomm buffers
DosLink applications, 5-13
Windows applications, 3-15
XVT applications, 4-10

using event hooks with XVT 3.0
XVT applications, 4-29

using keyboard and event hooks with XVT 2.0
XVT applications, 4-30

V

version control
library, 7-19

W

Windows 3.1 issues
debugging, 8-6
Windows 3.1 version information resource
library, 7-22
Windows 3.x issues
library, 7-25
Windows applications
advanced error handling, 3-36
advanced status handling, 3-36
allocating buffers, 3-5
basic error handling, 3-17
basic procedures, 3-2
basic status handling, 3-22
calling INFOConnect accessories, 3-41
canceling pending requests, 3-32
closing a session, 3-26
compiling, 3-44
data compression and error detection, 3-39
encoding and decoding, 3-38
handling data communications errors, 3-33
INFOConnect version RCDATA version information, 3-47
initializing ICS, 3-2
linking, 3-51
making your application an INFOConnect accessory, 3-42
opening a session, 3-5
procedures for INFOConnect accessories, 3-41
receiving a buffer, 3-12
resource files, 3-46
running with old versions of ICS, 3-39
samples
CoupleW, 3-76
IcWinApp, 3-53
terminating your application, 3-28
transmitting a buffer, 3-8
using datacomm buffers, 3-15
Windows debug version
debugging, 8-7
Windows DLL requirements
library, 7-52
windows platform
installation, 1-1

X

XVT
definition, 2-7
installation, 1-22
XVT applications
advanced error handling, 4-26
advanced status handling, 4-26
allocating buffers, 4-3
basic error handling, 4-12
basic procedures, 4-2
basic status handling, 4-16
calling INFOConnect accessories, 4-34
canceling pending requests, 4-24
closing a session, 4-19
compiling, 4-37
data compression and error detection, 4-32
encoding and decoding, 4-31
handling data communications errors, 4-25
initializing ICS, 4-2
linking, 4-40
making your application an INFOConnect accessory, 4-34
opening a session, 4-3
procedures for INFOConnect accessories, 4-33
receiving a buffer, 4-8
resource files, 4-39
running with old versions of ICS, 4-32
samples
Couple, 4-46
IcOpenAc, 4-48
IcXvtApp, 4-44
terminating your application, 4-20
transmitting a buffer, 4-6
using datacomm buffers, 4-10
using event hooks with XVT 3.0, 4-29
using keyboard and event hooks with XVT 2.0, 4-30