

IBM i
7.4

*Programming
IBM Rational Development Studio for i
ILE C/C++ Programmer's Guide*



Note

Before using this information and the product it supports, read the information in [“Notices” on page 443.](#)

This edition applies to version 7, release 2, modification 0 of IBM Rational Development Studio for i (product number 5770-WDS) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

This document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

© **Copyright International Business Machines Corporation 1993, 2018.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

ILE C/C++ Programmer's Guide.....	1
PDF file for ILE C/C++ Programmer's Guide.....	3
About ILE C/C++ Programmer's Guide.....	5
Install Licensed Program Information.....	5
Notes About Examples.....	5
Control Language Commands and the Procedures in This Guide.....	5
Introduction.....	7
Introduction to the ILE C/C++ Compiler.....	7
Multi-Language Program Creation.....	7
Programming Languages Supported by the IBM i Operating System.....	7
ILE Program Creation.....	7
Binding Directories.....	8
Service Programs.....	8
Program and Resource Management.....	8
Program Flow.....	8
Program and Procedure Calls.....	9
Resource Allocation.....	9
Bindable APIs.....	9
Runtime Exceptions.....	9
Program Debugging.....	9
Creating and Compiling Programs.....	11
Creating a Program.....	11
The Program Development Process.....	11
Preparing a Program.....	11
Compiling a Program.....	11
Binding Modules.....	12
Running or Calling Objects.....	12
Debugging a Program.....	12
Entering Source Statements.....	13
Example Of Creating a Source File.....	13
Instructions.....	13
Source Code Sample.....	13
Creating a Program in One Step.....	14
Creating a Program in Two Steps.....	15
Identifying Program and User Entry Procedures.....	16
Understanding the Internal Structure of a Program Object.....	16
Using Static Procedure Calls.....	16
Working with Binding Directories.....	17
Creating a Binding Directory.....	17
Using the Binder to Create a Program	17
Preparing to Create a Program.....	17
Specifying Parameters for the CRTPGM Command.....	18
How Import Requests Are Resolved.....	19
Using a Binder Listing.....	19
Updating a Module or a Program Object	20
Updating a Program.....	21
Activating Groups.....	21

Messaging Support.....	21
Service Programs.....	22
Differences Between Programs and Service Programs.....	22
Public Interface.....	22
Considerations When Creating a Service Program.....	23
Using the Binder to Create a Service Program.....	23
Specifying Parameters for the CRTSRVPGM Command.....	23
Updating or Changing a Service Program.....	24
Using Control Language (CL) Commands with Service Programs.....	24
Creating, Compiling, and Binding a Service Program.....	24
Creating the Source Files.....	24
User Header File.....	25
Source Code Files.....	25
Compiling and Binding the Service Program.....	26
Binding the Service Program to a Program.....	27
Working with Exports from Service Programs.....	28
Determining Exports from Service Programs.....	28
Displaying Exported Defined Symbols with the Display Module Command.....	28
Creating a Binder Language Source File.....	29
Creating Binder Language Using SEU.....	29
Creating Binder Language Using the RTVBNDSRC Command.....	30
Updating a Service Program Export List.....	30
Using the Demangling Functions.....	31
Handling Unresolved Import Requests During Program Creation.....	31
Creating an Export Service Program Using Binder Language.....	32
Creating a Program with Circular References.....	32
Creating the Source Files.....	33
Compiling the Source Files into Modules.....	34
Generating the Binder Language to Create the Service Program.....	34
Binding the Modules into the Program.....	35
Handling Unresolved Import Requests Using the *UNRSLVREF Parameter.....	35
Handling Unresolved Import Requests by Changing Program Creation Order.....	36
Binding a Program to a Non-Existent Service Program.....	37
Instructions.....	37
Code Samples.....	38
Running the Program.....	38
Updating a Service Program Export List.....	38
Program Description.....	38
Creating the Source Files.....	39
Compiling and Binding Programs and Service Programs.....	40
Running the Program.....	41
Running a Program.....	41
The ILE C/C++ Runtime Model.....	42
Activations and Activation Groups.....	42
Runtime Library Functions and Activation Groups.....	43
Calling Programs.....	44
Using the Call (CALL) Command.....	44
Passing Parameters to the Called Program.....	44
Call (CALL) Command Parameter Conversions.....	46
Using the Process Commands (QCAPCMD) API.....	46
Using the Transfer Control (TFRCTL) Command.....	46
Example: Creating and Running a Program that Uses the TFRCTL Command.....	47
Code Samples.....	47
Creating a CL Command to Run a Program.....	48
Program Description.....	48
Instructions.....	48
Code Samples.....	49
Normal and Abnormal End-of-Program.....	50

Managing Activation Groups.....	50
Specifying an Activation Group.....	50
Running a Program in a Named Activation Group.....	50
Running a Program in Activation Group *NEW.....	51
Non-Standard Behavior with Named Activation Groups.....	52
Running a Program in Activation Group (*CALLER).....	52
Presence of a Program on the Call Stack.....	53
Deleting an Activation Group.....	53
Reclaiming System Resources.....	53
Using the Reclaim Resources (RCLRSC) Command.....	54
Managing Runtime Storage.....	54
Managing the Default Heap.....	54
Using Bindable APIs to Manage the Default Heap.....	55
Dynamically Allocating Storage at Runtime.....	55
Overriding Replacement Functions.....	55
Overloading the new or delete Operator.....	56
Improving Runtime Performance.....	56
Choosing Data Types to Improve Performance.....	57
Avoiding Use of the Volatile Qualifier.....	57
Replacing Bit Fields with Other Data Types.....	57
Minimizing the Use of Static and Global Variables.....	57
Using the Register Storage Class.....	57
Creating Classes to Improve Performance.....	58
Enabling Performance Measurement.....	58
Using a Compiler Option to Enable Performance Measurement.....	58
Minimizing Exception Handling.....	58
Turning on Return Codes during Record I/O.....	58
Turning Off C2M Messages during Record Input and Output.....	59
Using a Direct Monitor Handler.....	59
Minimizing Percolation of Exceptions.....	59
Example of Exception Percolation for a Sample ILE C Source Code.....	59
Reducing the Number of Function Calls and Arguments.....	61
Inlining Function Calls.....	61
Using Static Class Member Functions or Global Variables.....	61
Passing Arguments in Registers.....	61
Using Prototypes to Minimize Function Call Processing.....	62
Choosing Input and Output Functions to Improve Performance.....	62
Using Record Input and Output Functions.....	62
ISO C Record I/O.....	62
ILE C Record I/O.....	63
Using Input and Output Feedback Information.....	63
Blocking Records.....	63
Manipulating the System Buffer.....	63
Opening Files Once for Both Input and Output.....	64
Minimizing the Use of Shared Files.....	64
Minimizing the Number of File Opens and Closes.....	65
Defining Tape Files to Improve Performance.....	65
Improving Performance when Using Stream Input and Output Functions.....	65
Using C++ Input and Output Stream Classes.....	65
Using Physical Files Instead of Source Physical Files.....	65
Specifying Library Names.....	66
Using Pointers to Improve Performance.....	66
Avoiding Use of Open Pointers.....	66
Avoiding Pointer Comparisons.....	66
Reducing Indirect Access through Pointers.....	68
Using Shallow Copy instead of Deep Copy.....	69
Minimizing Space Requirements.....	69
Choosing Appropriate Data Types.....	69

Minimizing Dynamic Memory Allocation Calls.....	69
Arranging Variables to Reduce Padding.....	70
Removing Observability.....	71
Compressing Objects.....	72
Optimizing Use of Activation Groups.....	72
Calling Functions in Other Activation Groups.....	72
Reducing Program Startup Time.....	72
Minimizing Use of Virtual Functions.....	72
Choosing Compiler Options to Optimize for Speed or Size.....	73
Setting Runtime Limits.....	74
Example: Creating an ILE C Application.....	74
Process Flow.....	75
ILE Activation Group.....	76
Resource Requirements.....	76
Task Summary.....	76
Instructions to Create the Sample Application.....	78
Source Code Samples.....	81
Source Code for an Audit Log File.....	81
Source Code Pass Terminal Session Input to an ILE Program.....	81
Source Code to Define a CL Command to Collect Session Data.....	81
Source Code for a User Entry Procedure (UEP).....	82
Source Code to Calculate Tax and Format Cost for Output.....	83
Source Code to Write an Audit Trail.....	84
Source Code to Export Tax Rate Data.....	86
Binder Language to Export Tax Rate Data.....	86
Binder Language to Export the write-audit-trail Procedure.....	86
Debugging Programs.....	89
The ILE Source Debugger.....	89
Debug Data Options.....	89
Debug Language Syntax.....	89
Limitations of the Debug Language Syntax.....	90
Debug Commands.....	90
Examples of Using Debug Expressions in ILE C Programs.....	92
Examples of Program Definitions and Corresponding Debug Expressions.....	92
Evaluating Pointers to Find and Correct Errors.....	93
Evaluating Simple Expression to Find and Correct Errors.....	94
Evaluating Bit Fields to Find and Correct Errors.....	95
Evaluating Structures and Unions to Find and Correct Errors.....	95
Evaluating Enumerations to Find and Correct Errors.....	96
Examples of Displaying System and Space Pointers in the ILE Source Debugger.....	97
Preparing a Program for Debugging.....	98
Setting Up a Test Library.....	99
Creating a Listing View for Debugging.....	99
Working with Source Debug Sessions.....	99
Starting a Source Debug Session.....	100
Adding and Removing Programs from a Debug Session.....	102
Setting or Changing Debug Options During a Session.....	103
Example: Adding an OPM Program to an ILE Debug Session.....	103
Example: Setting Debug Options during a Debug Session.....	103
Viewing the Program Source.....	104
Displaying Other Modules in Your Program.....	104
Example: Changing the Module Displayed in a Session.....	104
Displaying a Different View Of a Module.....	105
Using Breakpoints to Aid Debugging.....	105
Types Of Breakpoints.....	106
Job and Thread Breakpoints.....	106
Conditional and Unconditional Breakpoints.....	106

Setting Breakpoints.....	106
Setting Unconditional Breakpoints from the Display Module Source Display.....	106
Setting Unconditional Breakpoints from the Command Line.....	107
Setting Conditional Breakpoints for a Macro.....	107
Setting Conditional Breakpoints for a Statement.....	108
Setting Conditional Thread Breakpoints.....	108
Setting a Conditional Thread Breakpoint from the Work with Module Breakpoints Display	108
Setting a Conditional Thread Breakpoint from the Command Line.....	109
Testing Breakpoints.....	109
Removing All Breakpoints.....	109
Using Watches to Aid Debugging.....	109
Characteristics and Limitations Of Watches.....	109
Setting and Removing Watch Conditions.....	110
Setting watch conditions.....	110
Using the WATCH Debug Command.....	111
Removing Watch Conditions.....	112
Automatic Removal Of Watch Conditions.....	112
Example Of Setting a Watch Condition.....	112
Displaying Active Watches.....	113
Stepping Through Programs.....	114
Stepping Over Programs.....	114
Using F10 to Step Over Programs.....	114
Using the STEP OVER Debug Command.....	114
Stepping into Programs.....	114
Using F22 to Step into Programs.....	114
Using the STEP INTO Debug Command.....	114
Stepping into Called Programs.....	115
Example of Stepping into a Program Using F22.....	115
Stepping into an OPM Program.....	116
Stepping Over Procedures.....	116
Stepping into Procedures.....	117
Debugging Variables.....	118
Displaying the Value Of a Variable.....	118
Using F11 to Display Variables.....	119
Changing the Value of a Variable.....	120
Changing the Value of a Scalar Variable.....	120
Equating a Name with a Variable, Expression, or Debug Command.....	121
Displaying a Structure.....	122
Displaying Variables As Hexadecimal Values.....	122
Displaying Null-Ended Character Arrays.....	123
Displaying Character Arrays.....	124
Sample EVAL Commands for Pointers, Variables, and Bit Fields.....	125
EVAL Commands for System and Space Pointers.....	127
Displaying a Class Template and a Function Template.....	128
Source for Sample EVAL Commands.....	129
Source for Sample EVAL Commands for Displaying System and Space Pointers.....	130
Source for Sample EVAL Commands for Displaying C++ Constructs.....	131
Changing Module Optimization and Observability.....	133
Changing Optimization Levels.....	134
Removing Module Observability.....	135
Performing I/O Operations.....	137
Using ILE C/C++ Stream and Record I/O Functions with IBM i files.....	137
ILE C Record I/O Functions.....	137
Stream Buffering.....	137
Dynamic Stream File Creation.....	138
Open Modes for Dynamically Created Stream Files.....	138
Standard I/O Text Stream Files (<stdio.h>).....	138

Overriding Standard Output to the Terminal.....	139
Allowing a Program to Re-Read an Input File with QINLINE Specified.....	139
IBM i Files.....	139
IBM i File Types.....	139
Stream Files and ILE C I/O Operations.....	140
Avoiding Positioning Problems in the File.....	140
Using the fopen() Function.....	140
Using the open() member Function.....	140
IBM i File Naming Conventions.....	140
File Control Structure of Text Streams and Binary Streams.....	142
I/O Processes for Text Stream Files.....	143
Opening Text Stream Files.....	143
Writing Text Stream Files.....	144
Reading Text Stream Files.....	145
Updating Text Stream Files.....	146
I/O Process for Binary Stream Files.....	146
Opening Binary Stream Files (character at a time).....	146
Writing Binary Stream Files (character at a time).....	148
Reading Binary Stream Files (character at a time).....	148
Updating Binary Stream Files (character at a time).....	149
Opening Binary Stream Files (record at a time).....	151
Writing Binary Stream Files (record at a time).....	152
Reading Binary Stream Files (record at a time).....	153
Open Feedback Area.....	154
I/O Feedback Area.....	154
Using Session Manager.....	155
Obtaining the Session Handle.....	155
Using Session Manager APIs.....	155
Example: Using an ILE Bindable API to Display a DSM Session.....	155
Instructions.....	155
Code Samples.....	156
Using ILE C/C++ Stream Functions with the IBM i Integrated File System.....	157
The Integrated File System (IFS).....	158
root(/) File System.....	159
User Access.....	159
Path Names.....	159
Open Systems (QOpenSys) File System.....	159
User Access.....	160
Path Names.....	160
Library (QSYS.LIB) File System.....	160
File Handling Restrictions.....	160
Path Names.....	160
Document Library Services (QDLS) File System.....	161
User Access.....	161
Path Names.....	161
LAN Server/400 (QLANSrv) File System.....	161
User Access.....	161
Path Names.....	161
Optical Support (QOPT) File System.....	162
Path Names.....	162
File Server (QFileSvr.400) File System.....	162
Path Names.....	162
Enabling Integrated File System Stream I/O.....	163
Using Stream I/O with Large Files.....	163
Stream Files.....	164
Stream Files Versus Database Files.....	164
Text Streams.....	165
Binary Streams.....	166

Opening Text Stream and Binary Stream Files.....	166
Storing Data as a Text Stream or as a Binary Stream.....	166
Using the Integrated File System (IFS).....	167
Copying Source Files into the IFS.....	167
Editing Stream Files.....	167
The SRCSTMF Parameter.....	168
Header File Search.....	168
Include File Links.....	168
Include Directive Syntax.....	169
Include Search Path Rules.....	170
Considerations for Specifying Source Stream Files.....	173
Restrictions on the Absolute Include Path Name.....	173
Recommendation for Source and Header Files.....	174
Preprocessor Output.....	174
Listing Output.....	174
Code Pages and CCSIDs.....	175
Pitfalls to Avoid.....	175
Examples of Using Integrated File System Source.....	175
Using Stream I/O.....	176
Large Files.....	176
Open Mode.....	177
Line-End Characters.....	177
Working with File Systems and Devices.....	179
Using Externally Described Files in a Program.....	179
Creating Externally Described Database Files.....	179
Creating Type Definitions.....	180
Creating Header Descriptions.....	180
Specifying the Record Format Name.....	181
Specifying Record Field Names.....	181
Including Database Files in the Type Definition.....	182
Defining the Structure Type (KEY Field).....	182
Using Long Names for Files.....	184
Level Checking to Verify Descriptions.....	184
Using the GENCSRC Utility for Level Checking.....	185
Using the #pragma mapinc Directive for Level Checking.....	187
Avoiding Field Alignment Problems in C/C++ Structures.....	188
Including External Field Definitions in a Program.....	189
The INPUT Option.....	189
The OUTPUT Option.....	190
The BOTH Option.....	190
Defining and Using Indicators.....	191
Creation of Indicators in the File Buffer.....	191
Creating a Separate Indicator Area.....	191
Including Physical and Logical Database Files in a Program.....	193
Including Device Files in a Program.....	193
Including Externally Described Multiple Record Formats in a Logical File.....	193
Using Externally Described Packed Decimal Data in a Program.....	196
Using Database Files and Distributed Files in a Program.....	197
Database Files and Distributed Files.....	197
Physical Files and Logical Files.....	197
Describing Records in Database Files.....	198
Defining Externally Described Files.....	198
Data Files and Source Files.....	198
Access Paths.....	198
Arranging Key Fields.....	199
Duplicate Key Values.....	199
Deleted Records.....	199

Locking.....	199
Sharing.....	200
Null-Capable Fields.....	200
Opening Database and DDM Files as Record Files.....	201
Record Functions for Database and DDM Files.....	201
I/O Considerations for DDM Files.....	202
Opening Database and DDM Files as Binary Stream Files.....	202
I/O Considerations for Binary Stream Database and DDM Files.....	203
Binary Stream Functions for Database and DDM Files.....	203
Processing a Database Record File in Arrival Sequence.....	203
Instructions.....	203
Source Code Sample.....	204
Processing a Database Record File in Keyed Sequence.....	204
Processing a Database Record File Using Record Input and Output Functions.....	206
Synchronizing Database File Changes in a Single Job.....	208
Blocking Records.....	211
Using Device Files in a Program.....	211
Using IBM i Feedback Areas for all Device Files.....	212
Using Indicators to Transfer Information.....	212
Types of Indicators.....	212
Separate Indicator Areas.....	212
Major and Minor Return Codes.....	213
Example: Returning Indicators to a Separate Indicator Area.....	213
Instructions.....	213
Source Code Samples.....	214
Example: Returning Indicators to the File Buffer.....	214
Instructions.....	214
Code Samples.....	215
Establishing the Default Program Device.....	216
Changing the Default Program Device.....	218
Obtaining Feedback Information.....	220
Using Display Files and Subfiles.....	222
Display Files and Subfiles.....	222
I/O Considerations for Display Files.....	222
I/O Considerations for Subfiles.....	222
Using Subfiles to Minimize I/O Operations.....	223
Opening Display Files and Subfiles as Binary Stream Files.....	224
I/O Considerations for Binary Stream Subfiles.....	225
Program Devices for Binary Stream Display Files.....	225
Binary Stream Functions for Display Files and Subfiles.....	225
Opening Display Files as Record Files.....	225
I/O Considerations for Record Display Files.....	225
I/O Considerations for Record Subfiles.....	226
Record Functions for Display Files and Subfiles.....	226
Using Intersystem Communication Function Files.....	227
I/O Considerations for Intersystem Communication Function Files.....	227
Opening ICF Files as Binary Stream Files.....	227
I/O Considerations for Binary Stream ICF Files.....	227
Program Devices for Binary Stream ICF Files.....	227
Binary Stream Functions for ICF Files.....	227
Opening ICF Files as Record Files.....	228
I/O Considerations for Record ICF Files.....	228
Program Devices for Record ICF Files.....	228
Record Functions for ICF Files.....	229
Using Printer Files.....	233
I/O Considerations for Printer Files.....	233
Opening Printer Files as Binary Stream Files.....	233
Binary Stream Functions for Printer Files.....	233

Opening Printer Files as Record Files.....	233
Record Functions for Printer Files.....	234
Writing to a Tape File.....	236
I/O Considerations for Tape Files.....	236
Opening Tape Files as Binary Stream Files.....	236
I/O Considerations for Binary Stream Tape Files.....	236
Binary Stream Functions for Tape Files.....	236
Opening Tape Files as Record Files.....	237
I/O Considerations for Record Tape Files.....	237
Record Functions for Tape Files.....	237
Writing to a Diskette File.....	239
I/O Considerations for Diskette Files.....	239
Opening Diskette Files as Binary Stream Files.....	240
I/O Considerations for Binary Stream Diskette Files.....	240
Binary Stream Functions for Diskette Files.....	240
Opening Diskette Files as Record Files.....	240
I/O Considerations for Record Diskette Files.....	241
Record Functions for Diskette Files.....	241
Using Save Files.....	243
I/O Considerations for Save Files.....	243
Opening Save Files as Binary Stream Files.....	243
I/O Considerations for Binary Stream Save Files.....	243
Binary Stream Functions for Save Files.....	243
Opening Save Files as Record Files.....	243
I/O Considerations for Record Save Files.....	244
Record Functions for Save Files.....	244
Working with IBM i Features.....	245
Handling Exceptions in a Program.....	245
ILE Language-Specific Error Handling.....	245
Exception Messages.....	246
How the System Processes Exceptions.....	246
How the Call Message Queue Handles ILE Procedures and Functions.....	247
How Control Boundaries Affect Exception Handling in ILE.....	247
Unmonitored Exceptions and Unhandled Exceptions.....	247
Example of ILE C Source Code with an Unhandled Exception.....	247
Nested Exceptions.....	248
Detecting Stream File and Record File Errors.....	248
Checking the Return Value of a Function.....	249
Checking the Errno Value.....	249
Initializing Errno.....	249
Viewing and Printing the Errno Value.....	249
Example: Checking the errno Value for the fopen() Function.....	250
Checking the Major/Minor Return Code.....	250
Checking the System Exceptions for Record Files.....	250
Checking the Global Variable _EXCP_MSGID.....	251
Using ILE Exception Handlers.....	251
Types of Exception Handlers.....	251
Using ILE Direct Monitor Handlers.....	251
Using the pragma Directives.....	251
Using Communications Area Variables.....	252
Scoping Direct Monitor Handles.....	252
Using Exception Classes.....	253
Specifying Control Actions.....	253
Specifying Message Identifiers.....	254
Example of Source Code that Uses a Direct Monitor Handler.....	255
Example of Source that Illustrates How to Use Direct Monitor Handlers.....	255
Example of a Service Program that Provides Direct Monitor Handle.....	257

Example that Uses Labels instead of Functions as Handlers.....	258
Using ILE Condition Handlers.....	258
When to Use an ILE Condition Handler.....	259
Example of ILE Source that Uses Condition Handlers.....	259
Example of a Service Program that Provides ILE Condition Handlers.....	259
Examples of Handling an Exception.....	260
Example of Handling a Divide-By-Zero Exception.....	260
Example of Promoting an Exception.....	262
Using the C/C++ Signal Handler.....	263
When to Use the Signal Handler.....	264
Raising Signals.....	264
Signal Handling Function Prototypes.....	264
How the ILE C/C++ Runtime Environment Handles Signals.....	265
Resetting the Signal Action.....	266
Stacking Signal Handlers.....	266
Example: Setting Up a Signal Handler.....	266
Instructions.....	266
Source Code Sample that Sets Up Signal Handlers.....	267
Using Both C/C++ Signal and ILE Exception Handlers.....	269
Order of Priority.....	269
Example of Using a Direct Monitor Handler and Signal Handler Together.....	269
Handling Nested Exceptions.....	270
Using Cancel Handlers.....	271
Example: Using a Variety of Ways to Detect Errors and Handle Exceptions.....	273
Instructions.....	273
Source Code Samples.....	273
Using pointers in a Program.....	276
IBM i pointer Types.....	276
Using Open Pointers.....	277
Using Pointers Other than Open Pointers.....	277
Declaring Pointer Variables.....	278
Declaring IBM i Pointer Variables in C and C++.....	278
Declaring a Function Pointer to a Bound Procedure in ILE C.....	278
Declaring a Function Pointer with OS-Linkage in ILE C and ILE C++.....	279
Casting Pointers.....	280
Example: Passing IBM i pointers as Arguments on a Dynamic Program Call to Another ILE C Program.....	281
Instructions.....	281
Source Code Samples.....	282
Using ILE C/C++ Call Conventions.....	283
Program and Procedure Calls.....	283
Using Dynamic Program Calls.....	284
How the ILE Call Stack Is Used to Control Program Flow.....	284
Renaming Programs and Procedures.....	285
Calling Programs that Have Library Qualification.....	286
Calling C++ Programs and Procedures from ILE C.....	287
Specifying the Linkage Convention.....	288
Example: An ILE C Program that Uses C++ Objects.....	288
Program Structure.....	288
Program Flow.....	290
Program Output.....	290
Accessing C++ Classes from ILE C.....	291
Mapping a C++ Class to a C Structure.....	291
Example: An ILE C Program that Uses C++ Objects.....	292
Program Files and Structures.....	292
Program Description.....	293
Program Output.....	294
Porting Programs from Another Platform to ILE.....	295

Limitations to Porting Code to ILE C or C++.....	295
File Inclusions.....	295
Platform-Specific Extensions.....	295
Members of a Union.....	295
Members of a Structure.....	295
Decimal Constants.....	296
Decimal Constants and Case Statements.....	296
Library QSYS.LIB under IFS.....	296
Teraspace Considerations.....	297
Modifying Calls of ILE C++ Objects.....	297
Differences in Header Files.....	297
Differences in Linkage Specification.....	297
Differences in Function Definitions.....	298
Using BCD Macros to Port Coded Decimal Objects to ILE C++.....	298
Examples.....	298
Mapping Class Template Instantiations to ILE C Syntax.....	299
Handling Extra Precision for Multiplication and Division.....	299
Determining the Number of Digits in an Object.....	299
Determining the Number of Digits in an Internal Packed Decimal Data Object.....	300
Formatting the Value of a Formatted C Input or Output Function.....	301
Print Function Flags.....	301
Print Function Field Width.....	301
Print Function Field Precision.....	301
Conversion Specifiers.....	301
Porting Conditional Operators to ILE C or C++.....	302
Example of an Explicit Cast that Resolves Class Differences between Expression.....	302
Example of Use of a Consistent Variable Type.....	302
Porting ILE C Packed Decimal Data Types to the _DecimalT Class Template.....	303
Differences in Using Packed Structures.....	304
Differences in Error Checking.....	304
Invalid Decimal Format.....	305
Mathematical Operators.....	305
Header Files that Work with Both C and C++.....	305
Using Dual Function Prototypes.....	305
Permitting ILE C Programs to Access C++ Linkage Functions.....	306
Including QSYSINC Header Files.....	307
Handling the Stricter C++ Type Checking.....	307
Resolving Integer Data Type Size Issues.....	307
Resolving Incompatible Pointer Types.....	307
Disabling Name Mangling to Avoid Undefined Name Errors.....	308
Resolving Type Mismatches with the C++ Function Prototype.....	308
Example of Function Prototype Mismatch.....	308
Handling the Function Prototype Mismatch.....	309
Declaring unsigned char Pointers as unsigned char Variables.....	309
Initializing Character Arrays.....	309
Specifying Access to String Literals.....	310
Avoiding Uncaught Exceptions by Scoping to a Single Activation Group.....	310
Working with Multi-Language Applications.....	311
Inter-Language Procedure Calls.....	311
ILE Conventions for Calling Any Program (*PGM).....	312
Mixing Recursive and Non-Recursive Calls.....	313
Passing Arguments from an ILE Program to a Non-EPM Program.....	313
Passing Arguments from an ILE Program to an EPM Program.....	314
Using a Linkage Specification in a C++ Dynamic Program Call.....	314
Valid String Literals.....	315
Linkage Specification.....	315
Calling Any ILE Program from ILE C/C++.....	315
Passing Parameters from ILE C++ to a Different High-Level Language.....	315

Using Different Linkage Specifications.....	315
Using Different Linkage Specifications (C++ Only).....	318
Using Default Parameter Passing Styles.....	319
Using Operational Descriptors to Pass Parameters of Unknown Data Type.....	320
Example: Calling a Function with Operational Descriptors.....	321
Type Casting to Override a Function without Overriding Linkage.....	321
Passing Arguments from a CL Program to an ILE C++ Program.....	321
How CL Constants Are Passed to an ILE C++ Program.....	321
How CL Variables Are Passed to an ILE C++ Program.....	322
CL Example: a Multi-Language ILE Application.....	322
Program Description.....	322
Program Structure.....	322
Program Activation.....	323
Application Modules and Files.....	324
Invoking the ILE Program.....	328
Example: a User-Defined CL Program that Calls an ILE C++ Program.....	328
Programming Tasks.....	328
Source Code.....	329
Using the CL Command SQUARE to Return the Calculated Value.....	329
Example: CL Program that Passes Parameters to an ILE C++ Program.....	330
Accessing ILE C Procedures from Any ILE Program.....	331
Static Procedure Calls.....	331
Procedure Pointer Calls.....	332
Called Procedures and Operational Descriptors.....	332
Example: Calling an ILE API from ILE C.....	332
Operational Descriptors and the #pragma descriptor Directive.....	333
Example: Declaring a Function that Requires Use of Operational Descriptors.....	333
Example: Generating Operational Descriptors.....	333
OPM CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C Program.....	333
Basic Program Structure.....	334
Program Modules and Activation Groups.....	334
Programming Tasks.....	335
ILE CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C Program.....	340
Program Modules and Activation Groups.....	341
Programming Tasks.....	342
ILE-OPM CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C++ Program.....	348
Program Description.....	348
Program Structure.....	348
Program Activation.....	349
Program Files.....	349
Invoking the ILE-OPM Program.....	353
Using a Linkage Specification to Call an ILE Procedure.....	353
Using a Linkage Specification in a C++ Dynamic Program Call.....	354
Valid String Literals.....	354
Linkage Specification.....	354
Using Packed Decimal Data in a C Program.....	354
Converting from Packed Decimal Data Types.....	355
Converting from a Packed Decimal Type to a Packed Decimal Type.....	355
Converting from a Packed Decimal Type to an Integer Type.....	357
Converting from a Packed Decimal Type to a Floating Point Type.....	358
Overflow Behavior.....	358
Passing Packed Decimal Data to a Function.....	359
Passing a Pointer to a Packed Decimal Variable to a Function.....	359
Calling Another Program that Contains Packed Decimal Data.....	360
Using Library Functions with a Packed Decimal Data Type.....	361
Understanding Packed Decimal Data Type Errors.....	364
Packed Decimal Warnings and Error Conditions.....	364
Suppressing a Runtime Overflow Exception.....	365

Using Packed Decimal Data in a C++ Program.....	365
The IBM i Binary Coded Decimal (BCD) Header File.....	366
Using the <code>_DecimalT</code> Class Template.....	367
Declaring <code>_DecimalT</code> Class Template Objects.....	367
Using the <code>__D</code> Macro to Simplify Code.....	368
<code>_DecimalT</code> Class Template Input and Output.....	368
Using Operators with the <code>_DecimalT</code> Class Template.....	368
Using Arithmetic Operators with the <code>_DecimalT</code> Class Template.....	369
Using Relational Operators with the <code>_DecimalT</code> Class Template.....	369
Using Conditional Expressions with the <code>_DecimalT</code> Class Template	370
Using Equality Operators with the <code>_DecimalT</code> Class Template.....	370
Using Unary Operators with the <code>_DecimalT</code> Class Template.....	371
C++ Packed Decimal Data Conversions.....	372
Converting Values from One <code>_DecimalT</code> Class Template to Another.....	372
Converting Values from a <code>_DecimalT</code> Class Template to an Integer Data Type.....	372
Converting Values from a <code>_DecimalT</code> Class Template to a Floating Point Data Type.....	373
Determining the Size of a <code>_DecimalT</code> Class Template.....	374
Determining the Number of Digits in a <code>_DecimalT</code> Class Template.....	374
Determining the Precision of a <code>_DecimalT</code> Class Template.....	374
How Overflows Are Handled.....	374
Using C++ Exception Handling with the <code>_DecimalT</code> Template	375
<code>_DecimalT</code> Class Template Runtime Exceptions.....	375
When Runtime Exceptions Occur.....	375
Runtime Exceptions Issued by the Compiler for <code>_DecimalT</code> Class Templates.....	375
Defining a C++ <code>_DecimalT</code> Class Template Exception Handler.....	377
Using Debug Macros for <code>_DecimalT</code> Class Templates.....	377
Enabling and Disabling Error Checking for the <code>_DecimalT</code> Class Template.....	377
Passing a <code>_DecimalT</code> Class Template Object to a Function.....	378
Passing a Pointer to a <code>_DecimalT</code> Class Template Object.....	379
Calling Another Program Containing a <code>_DecimalT</code> Class Template.....	379
File I/O With <code>_DecimalT</code> Class Templates.....	380
Using Templates in C++ Programs.....	383
Managing Template Instantiations.....	384
Template Instantiation Management Options.....	384
How the ILE C++ Compiler Handles Template Instantiations.....	385
Generation of Static Member Definitions.....	385
Internal Linkage.....	385
External Linkage.....	386
Example of a Class Template Instantiation.....	386
Declarations and Definitions.....	386
Linkage.....	386
Using the Default Template Instantiation Management Option.....	387
Manually Structuring Code for Single Instantiations.....	387
Explicit Instantiations.....	388
Using the ILE Template Registry Option.....	388
How the ILE Template Registry Option Works.....	388
Specifying Values for the <code>TMPLREG</code> Parameter.....	389
Using the ILE <code>TEMPINC</code> Option.....	389
How the ILE <code>TEMPINC</code> Option Works.....	389
Structuring a Program for <code>TEMPINC</code> -Managed Instantiations.....	390
The Template-Implementation File.....	391
Tempinc Files.....	392
Using Teraspace in ILE C and C++ Programs.....	392
Supported Teraspace Environments.....	393
C/C++ Pointer Support.....	393
C/C++ Pointer Conversions.....	393
Bindable APIs for Using Teraspace.....	394
The 16-Byte Runtime Binding Libraries.....	394

The 8-Byte Runtime Binding (RTBND) Library Extensions.....	394
Using RTBND to Optimize Performance of a C++ Program.....	395
Requirements.....	395
Error Conditions.....	395
Limitations.....	395
Characteristics of Each Teraspace Storage Model.....	395
Binary Compatibility Considerations When Porting Code in a Teraspace Environment.....	397
Specifying the Teraspace Environment.....	397
Determining the Size of a Specific Pointer.....	397
Maintaining Consistent Argument Declarations.....	397
Source Code Samples.....	398
Example: Effect of Forward Declarations on the Data Model.....	398
Example: Redefining the new or delete Operator.....	398
Example: How a Template Adopts a Data Model.....	399
Examples: Overloading Functions.....	400
Casting with RunTime Type Information.....	400
The RTTI Language Extension.....	400
Using C++ Language-Defined RTTI.....	401
The dynamic_cast Operator.....	401
Dynamic Casts with Pointers.....	401
Dynamic Casts with References.....	402
The typeid Operator.....	402
Results of typeid Operations.....	403
Using the typeid Operator in Expressions.....	403
The type_info Class.....	404
Using RTTI in Constructors and Destructors.....	404
ILE C++ Extensions to RTTI.....	404
The extended_type_info Classes.....	405
Using International Locales and Coded Character Sets.....	407
Internationalizing a Program.....	407
Coded Character Set Identifiers.....	407
Source File Conversions to CCSID.....	407
Creating a Source Physical File with a Coded Character Set Identifier.....	408
Changing the Coded Character Set Identifier (CCSID).....	408
Converting String Literals in a Source File.....	409
Using Unicode Support for Wide-Character Literals.....	410
Representation of Wide-Character Literals.....	410
Enabling Unicode Character Set Support.....	411
Effect of Unicode on #pragma convert() Operations.....	411
GB18030 Code Page Support.....	411
Generating Wide Characters and String Literals in UTF-32.....	411
Considerations.....	412
Targeting a CCSID.....	412
How the ILE C/C++ Compiler Converts a Source File to a Target CCSID.....	412
Literals, Comments, and Identifiers.....	413
Limitations.....	413
International Locale Support.....	413
Elements of a Language Environment.....	413
Locales.....	414
ILE C/C++ Support for Locales.....	414
ILE C/C++ Support for *CLD and *LOCALE Object Types.....	414
C Locale Migration Table.....	415
POSIX Locale Definition and *LOCALE Support.....	418
LOCALETYPE Compiler Option.....	418
Creating Locales.....	418
Creating Modules Using LOCALETYPE(*LOCALE).....	419
Categories Used in a Locale.....	419

Setting an Active Locale for an Application.....	420
Using Environment Variables to Set the Active Locale.....	420
SAA and POSIX *Locale Definitions.....	421
Locale-Sensitive Runtime Functions.....	421
The GENCSRC Utility and the #pragma mapinc Directive.....	423
Interlanguage Data-Type Compatibilities.....	425
Ensuring thread safety (C++).....	437
Related information.....	439
Notices.....	443
Programming interface information.....	444
Trademarks.....	444
Terms and conditions.....	445
Index.....	447

ILE C/C++ Programmer's Guide

This information is for programmers who are familiar with the C and C++ programming languages and who plan to write or maintain ILE C/C++ applications.

Users need experience in using at least one of the following:

- Applicable IBM® i menus and displays
- Control language (CL) commands

PDF file for ILE C/C++ Programmer's Guide

You can view and print a PDF file of this information.


To view or download the PDF version of this document, select [ILE C/C++ Programmer's Guide](#).

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the [Adobe Web site](http://www.adobe.com/products/acrobat/readstep.html) (www.adobe.com/products/acrobat/readstep.html) .

About ILE C/C++ Programmer's Guide

This guide provides instructions on:

- Entering source statements
- Creating a program in two steps
- Creating a program in one step
- Running a program
- Debugging a program
- Managing streams and record files
- Writing programs that:
 - Use externally described files
 - Use database files and distributed files
 - Use device files
 - Handle exceptions
 - Call programs and procedures
 - Use pointers on the system
- Internationalizing a program
- Using templates in C++ programs
- Porting programs to ILE C++
- Casting with RunTime Type Information
- Using Teraspace support
- Customizing programs using locales

Install Licensed Program Information

The QSYSINC library must be installed on systems that use the ILE C/C++ compiler.

Notes About Examples

Examples illustrating the use of the ILE C/C++ compilers are written in a simple style. Note the following:

- The examples do not demonstrate all of the possible uses of C/C++ language constructs.
- Some examples are code fragments and cannot be compiled without additional code.
- All complete, runnable examples begin with T1520. They can be found in the QCPPL library, in source file QACSRC.
- Most of the examples found in this guide are illustrated by entering control language (CL) commands on a CL command line. You can use a CL program to run most of the examples.
- See the member T1520INF in QCPPL/QAINFO for information about running the examples.

Control Language Commands and the Procedures in This Guide

In this guide, the procedures instruct you to enter CL commands. To enter a CL command, type the command on the command line and then either press the Enter key or press F4 to be prompted for options and parameters.

If you need online help information, press F1 (Help) on the CL command prompt display.

See the *ILE C/C++ Compiler Reference* for command syntax for the CL commands including Integrated Language Environment® (ILE) CL commands. CL commands can be used in either batch or interactive mode, or from a CL program.

Note: You need object authority to use CL commands.

Introduction

This topic introduces Integrated Language Environment® (ILE) and IBM i operating system programming features. It includes overviews of the following:

- [Multi-language program creation](#)
- [Program and resource management](#)
- [Program debugging](#)

Introduction to the ILE C/C++ Compiler

Integrated Language Environment (ILE), together with the IBM i operating system, provides a wide range of support for serious program development. C and C++ are two of the programming languages supported by ILE. [Table 1 on page 7](#) lists the complete set of ILE languages.

The ILE C/C++ Compiler supports program development in both C and C++ programming languages. C++ extends the capabilities of the C compiler by providing:

- Additional keywords
- Parameterized types (templates)
- Support of object-oriented programming via classes
- Stricter type checking

ILE C/C++ provides advantages in the following areas of program development:

- [Creation of multi-language programs and applications](#)
- [Program flow and resource management](#)
- [Program debugging](#)

Multi-Language Program Creation

You can build mixed-language programs that are composed of modules written in any ILE programming language.

Programming Languages Supported by the IBM i Operating System

The ILE family of compilers includes: ILE C++, ILE C, ILE RPG, ILE COBOL, and ILE CL. [Table 1 on page 7](#) lists the programming languages supported by the IBM i operating system.

Table 1. Programming Languages Supported by the IBM i family

Integrated Language Environment (ILE)	Original Program Model (OPM)	Extended Program Model (EPM)
C++	BASIC (PRPQ)	C
C	CL	FORTRAN
CL	COBOL	PASCAL (PRPQ)
COBOL	PL/I (PRPQ)	
RPG	RPG	

ILE Program Creation

ILE program creation consists of:

1. Compiling source code into modules
2. Binding (combining) one or more modules into a program object.

You can create and maintain multi-language programs because you can combine modules from any ILE language.

Figure 1 on page 8 shows the process of creating an ILE program through compiler and binder invocation.

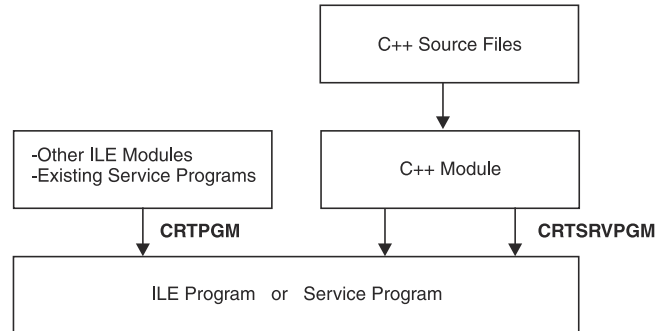


Figure 1. Program Creation in ILE

Note: Once a program is created, you can update it by using the Update Program (UPDPGM) or Update Service Program (UPDSRVPGM) command. These commands are useful because you only need to have the new or changed modules available when you want to update the program.

Binding Directories

You can create a *binding directory* to contain the names of modules and service programs that your ILE C++ program or service program may need. Each binding directory is created in a specific library with the Create Binding Directory (CRTBNDDIR) command. Use binding directories to reduce program size. Modules or service programs listed in a binding directory are called only if needed.

Service Programs

You can bind modules into service programs (*SRVPGM). *Service programs* are a means of packaging callable routines (functions or procedures) into a separately bound program. The use of service programs provides modularity and improves maintainability. You can use off-the-shelf modules developed by third parties or you can package your own modules for third-party use.

Program and Resource Management

ILE provides a common basis for:

- [Managing program flow](#)
- [Sharing resources](#)
- [Bindable application program interfaces \(APIs\)](#)
- [Handling exceptions during a program's runtime](#)

Program Flow

The process of getting a program or service program ready to run is known as activation. *Activation* allocates resources within a job so that one or more programs can run in that space. When a program is called, ILE automatically initiates the activation group specified for the program. If the specified activation group for a program does not exist when the program is called, it is created within the job to hold the program's activation.

Note: For more information on activation groups, see:

- [“Activations and Activation Groups” on page 42.](#)

- [“Managing Activation Groups” on page 50.](#)

Program and Procedure Calls

In ILE, you can write programs in which ILE C++, OPM, and EPM programs can work together by using *dynamic program calls*. When using such calls, the calling program specifies the name of the called program. This name is resolved to an address at runtime, just before the calling program passes control to the called program.

You can optimize the use of dynamic program calls by using *static procedure calls*. Because the procedure names are resolved at bind time (that is, when you create the program), static procedure calls are faster than dynamic calls.

In addition, static procedure calls allow operational descriptors. *Operational descriptors* are used to call bindable APIs or procedures written in other ILE languages.

Note: A *procedure* is a self-contained set of code that performs a task and then returns to the caller. An ILE C++ module consists of one or more procedures.

See [“Using ILE C/C++ Call Conventions” on page 283](#) for information on calls between programs and procedures.

Resource Allocation

An *activation group* is the key element in governing an ILE program's resources and behavior. You can scope commitment-control operations to the activation group level. You can scope file overrides and shared open data paths to the activation group of the running program. The behavior of a program upon termination is affected by the activation group in which the program runs.

Note: For more information on activation groups, see:

- [“Activations and Activation Groups” on page 42.](#)
- [“Managing Activation Groups” on page 50.](#)

Bindable APIs

ILE offers a number of bindable APIs that supplement ILE C/C++ functions. *Bindable APIs* provide program calling and activation capability, condition and storage management, math functions, and dynamic screen management. The *System API Reference* contains information on bindable APIs.

Runtime Exceptions

Many C and C++ runtime library functions have a return value associated with them for error-checking purposes. For example, the `_Rfeov()` function returns 1 if the file has moved from one volume to the next. The `fopen()` function returns NULL if a file is not opened successfully. *ILE C/C++ Runtime Library Functions* contains information about the ILE C/C++ function return values.

Program Debugging

In ILE, you can perform source-level debugging on any program written in one or more ILE languages, provided that the program was compiled with debug information.

You can:

- Control the flow of a program by using debug commands while the program is running
- Set conditional and unconditional breakpoints prior to running the program
- Step through a specified number of statements and display or change variables after calling the program

When a program stops because of a breakpoint, a step command, or a runtime error, the pertinent module is displayed at the point where the program stopped. At that point, you can enter more debug commands.

See [“Debugging Programs” on page 89](#) for information on debugging ILE C/C++ programs.

Creating and Compiling Programs

This topic describes how to:

- [Use compiler and binder commands to create ILE C/C++ programs](#)
- [Create ILE service programs](#)
- [Work with procedures and data items that can be exported from a service program](#)
- [Run ILE programs](#)
- [Improve program performance](#)
- [Use the compiler and binder programs to create program modules and executables](#)

Creating a Program

This topic describes:

- [The program development process](#)
- [How to enter source statements](#)
- [How to create a program in one step](#)
- [How to create a program in two steps](#)
- [Messaging support](#)

The Program Development Process

During the development process, an ILE program passes through five stages:

- Preparing
- Compiling
- Binding
- Running
- Debugging

These process steps are not necessarily performed in the order listed. You can compile, correct compile time errors, modify, and recompile the program several times before binding it.

Preparing a Program

Preparing a program involves designing, writing, and creating source code. See [“Entering Source Statements”](#) on page 13 for more information about creating source code.

Compiling a Program

Issue a compile command against your source, and fix any compile errors that arise. You can see the errors either as messages in the job log or in the listing (if you chose to create one).

The ILE C/C++ compiler includes the following compile commands:

Table 2. ILE C/C++ Compile Commands

Compile Command	Use	Description
CRTCMOD	Create C Module	The Create Module command creates a module object. If your program will include objects from more than one source file, you must use the Create Module command for each source file, and then run CRTPGM specifying all the required *MODULEs to create the bound program.
CRTCPMOD	Create C++ Module	
CRTBNDC	Create Bound C Program	The Create Bound Program command performs both the module creation and the binding operation in one step, and produces a *PGM object from a single source file.
CRTBNDCPP	Create Bound C++ Program	

Note: The compile command might also originate in a CL program or makefile.

See the *ILE C/C++ Compiler Reference* for more information about the Create Module and Create Bound Program commands and their options.

Note: In the following pages:

1. CRTCMOD and/or CRTCPMOD may be referred to as simply the "Create Module" command.
2. CRTBNDC and/or CRTBNDCPP may be referred to as simply the "Create Bound Program" command.
3. Examples may show the use of either of the C or C++ versions of the Create Module and Create Bound Program commands. Unless specifically stated otherwise, both C and C++ versions of these commands function in the same way and can be used interchangeably, according to the language of the source program being compiled.

Binding Modules

If you created modules during compilation, you need to bind the module objects together using the Create Program (CRTPGM) or Create Service Program (CRTSRVPGM) commands. The result is an executable *PGM or *SRVPGM object.

Binding combines one or more modules into a program (*PGM) or a service program (*SRVPGM). Modules written in ILE C or C++ can be bound to modules written in any other ILE language. C++ programs can use functions from C++ class libraries, C libraries, and any ILE service program. The binder resolves addresses within each module, import requests and export offers between modules that are being bound together.

Once a program is created, you can later update it using the Update Program (UPDPM) or Update Service Program (UPDSRVPGM) commands. These commands are useful because you only need to have the new or changed modules available when you want to update the program.

Running or Calling Objects

*CMD objects are run, while *PGM objects are called. For example, to run HELLO *CMD, type HELLO on a command line and press Enter. To run HELLO *PGM, type CALL HELLO on the QCMD line and press Enter.

Debugging a Program

Debugging allows you to detect, diagnose, and eliminate runtime errors in a program. You can use the ILE source debugger to debug ILE or OPM programs.

For information about ILE debugging considerations, see *ILE Concepts, SC41-5606-09*

Entering Source Statements

Before you can start an edit session and enter your source statements, you must create a library and a source physical file. You can also compile source statements from integrated file system (IFS) files. See [“Using the Integrated File System \(IFS\)” on page 167](#) for details.

You can use the Start Programming Development Manager (STRPDM) command to start an edit session, and enter your source statements.

Besides Programming Development Manager (PDM), there are several other ways to enter your source:

- The Copy File (CPYF) command.
- The Start Source Entry Utility (STRSEU) command.
- The Programmer Menu.

This is by no means an exhaustive list. There are other ways of creating source and placing it on the IBM i platform, including NFS and ftp.

Example Of Creating a Source File

The following example shows you how to create a library, a source physical file, a member, start an edit session, enter source statements, and save the member.

Instructions

1. To create a library called MYLIB, enter:

```
CRTLIB LIB(MYLIB)
```

2. To create a source physical file called QCSRC in library MYLIB, enter

```
CRTSRCPF FILE(MYLIB/QCSRC) TEXT('Source physical file for all ILE C programs')
```

QCSRC is the default source file name for ILE C commands that are used to create modules and programs. For ILE C++ commands, the corresponding default is QCPPSRC. For information about how to copy this file to an integrated file system file, see [“Using the Integrated File System \(IFS\)” on page 167](#).

3. To start an edit session enter:

```
STRPDM
```

4. Choose option 3 (Work with members); specify the source file name QCSRC, and the library MYLIB.
5. Press F6 (Create), enter the member name T1520ALP, and source type C. The SEU Edit display appears ready for you to enter your source statements.
6. Type the source shown in [“Source Code Sample” on page 13](#) into your SEU Edit display. Trigraph sequences can be used in place of square brackets, as demonstrated in [Figure 2 on page 14](#).

Note: For more information about using trigraph sequences, see the *ILE C/C++ Language Reference*.


7. Press F3 (Exit) to go to the Exit display. Type Y (Yes) to save the member T1520ALP.

Source Code Sample

The ILE C compiler recognizes source code written in any single-byte EBCDIC CCSID (Coded Character Set Identifier) except CCSID 290, 905 and 1026. See [“Internationalizing a Program” on page 407](#) for information on CCSIDs.

Some characters from the C and C++ character set are not available in all environments. You can enter these characters into a C or C++ source program using a sequence of three characters called a trigraph.

Note: For more information about using trigraph sequences, see the *ILE C/C++ Language Reference*.

 The C compiler also supports digraphs. (The C++ compiler does not support digraphs.)

```

/* This program reads input from the terminal, displays characters, */
/* and sums and prints the digits. Enter a "+" character to      */
/* indicate EOF (end-of-file).                                  */

#define MAXLEN 60          /* The maximum input line size.    */

#include <stdio.h>
#include <ctype.h>

void main(void)
{
    int c;
    int i = 0, j = 0;
    int sum = 0;
    int count, cnt;
    int num[MAXLEN];      /* An array of digits.          */
    char letter??(MAXLEN??); /* An array of characters. Trigraphs
                           replace the square brackets.      */

    while ( ( c = getchar( ) ) != '+' )
    {
        if ( isalpha ( c ) ) /* A test for an alphabetic    */
        { /* character.          */
            letter[i++] = c;
        }
        else if ( isdigit ( c ) ) /* A test for a decimal digit. */
        { /* Trigraphs replace the square
           brackets.          */
            num??(j+??) = c - '0';
        }
    }
    printf ( "Characters are " );
    for ( count = 0; count < i; ++count )
    {
        printf ( "%c", letter[count] );
    }
    printf( "\nSum of Digits is " );
    for ( cnt = 0; cnt < j; ++cnt )
    {
        sum += num[cnt];
    }
    printf ( "%d\n", sum );
}

```

Figure 2. ILE C Source to Add Integers and Print Characters

Creating a Program in One Step

You can use the CRTBNDC and CRTBNDCPP Create Bound Program commands to create a program (*PGM object) in one step.

The Create Bound Program commands combine the steps of compiling and binding. Using them is the same as first calling the CRTCMOD or CRTCPMOD Create Module command, then calling the Create Program (CRTPGM) command, except that the module created by the Create Module command step is deleted after the CRTPGM step.

To use the Create Bound Program commands, the source member must contain a main() function.

Note: When a CRTPGM parameter does not appear in the Create Bound Program command, the CRTPGM parameter default is used. For example, the parameter ACTGRP(*NEW) is the default for the CRTPGM command, and is used for the Create Bound Program command. You can change the CRTPGM parameter defaults by using the Change Command Defaults (CHGCMDDFT) command.

You can use the CRTSQLCI or CRTSQLCPPI command to start the ILE C compiler and create a program object. The SQL database can be accessed from an ILE C/C++ program if you embed SQL statements in the ILE C/C++ source.

Example:

1. To create the program T1520ALP, using the source found in [Figure 2 on page 14](#), enter:

```

CRTBNDC PGM(MYLIB/T1520ALP) SRCFILE(QCPPLE/QACSRC)
        TEXT('Adds integers and prints characters') OUTPUT(*PRINT)

```



```
OPTION(*SHOWINC *NOLOGMSG) FLAG(30) MSGLMT(10)
CHECKOUT(*PARM) DBGVIEW(*ALL)
```

The options specified are:

- OUTPUT(*PRINT) - specifies that you want a compiler listing.
- OPTION(*SHOWINC *NOLOGMSG) - specifies that you want to expand include files in a compiler listing and not log messages in the job log.
- FLAG(30) - specifies that you want severity level 30 messages to appear in the listing.
- MSGLMT(10) — specifies that you want compilation to stop after 11 messages at severity level 30.
- CHECKOUT(*PARM) — shows a list of function parameters not used.
- DBGVIEW(*ALL) specifies that you want all three views and debug data to debug this program.

2. To see the compiler listing, enter one of the following CL commands:

- DSPJOB and then select option 4 (Display spooled files)
- WRKJOB and then select option 4 (Work with spooled files)
- WRKOUTQ *queue-name*
- WRKSPLF

Select an option to see the compiler listing.

3. To run the program enter:

```
CALL PGM(MYLIB/T1520ALP)
```

4. Type a and press Enter. Type 9 and press Enter. Type b and press Enter. Type 8 and press Enter. Type + and press Enter.

The interactive session is as shown:

```
> a
> 9
> b
> 8
> +
  Characters are ab
  Sum of Digits is 17
  Press ENTER to end terminal session.
```

Creating a Program in Two Steps

To take advantage of the flexibility that ILE C/C++ offers, you can compile and bind source code into an ILE C/C++ program in two steps:

1. In the first step, you create one or more ILE C/C++ *module objects* (*MODULE) from their respective source members using the Create Module command.
2. In the second step, you use the Create Program (CRTPGM) command to bind one or more of these module objects into an executable ILE *program object* (*PGM). Binding is the process of combining one or multiple modules and optional service programs, and resolving external symbols between them. The system code that combines modules and resolves symbols is called the binder.

For example,

```
CRTCMOD HELLO
CRTPGM HELLO
CALL HELLO
```

Using modules has these advantages:

- Modules are easier to maintain. It is easier to maintain a small module representing a single function than to maintain an entire program. For example, if you change only a line or two in a module, you may only need to recompile the module, rather than the entire program.

- Modules are easier to test. Testing of functions can be done in isolation. You do not have to run the entire program. A test harness which includes the module under test can be used instead.
- Modules are easier to code. You can subdivide the work into smaller source members rather than coding an entire program in a single source file.
- Modules can be reused in different application programs.

Identifying Program and User Entry Procedures

When a module object is created, a program entry procedure (PEP) and a user entry procedure (UEP) may also be generated.

Both ILE C and C++ require the `main()` function, but in ILE C, it becomes the UEP of an ILE program. After the PEP runs, it calls the associated UEP, and starts the ILE program running.

As part of the binding process, a procedure must be identified as the startup procedure, or program entry procedure (PEP). When a program is called, the PEP receives the command line parameters and is given initial control for the program. The procedures that get control from the PEP are called user entry procedures (UEP).

An ILE module contains a program entry procedure only if it contains a `main()` function. Therefore, one of the modules being bound into the program must contain a `main()` function.

Understanding the Internal Structure of a Program Object

Figure 3 on page 16 shows the internal structure of a typical program object, MYPROG, created by binding two modules, TRNSRPT and INCALC. In this example, TRNSRPT is the entry module containing the PEP, in addition to a UEP. Module INCALC contains a UEP only.

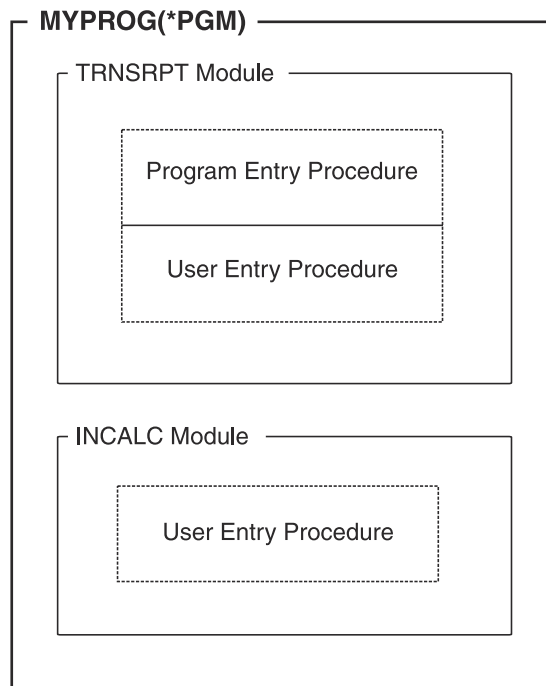


Figure 3. Structure of Program MYPROG

Using Static Procedure Calls

Within a bound object, procedures can be called using static procedure calls. These bound calls are faster than external calls. Therefore, an application consisting of a single bound program with many bound calls should perform faster than a similar application consisting of separate programs with many external inter-program calls.

Working with Binding Directories

A binding directory contains the names of the modules and service programs that you may need when creating an ILE program or service program.

Binding directories offer:

- A convenient method of packaging modules or service programs that you may need when creating an ILE program or service program.
- Reduce program size, because modules or service programs listed in a binding directory are used only if they are needed.

Binding directories are optional. They are objects identified to the system by the *BNDDIR* parameter on the CRTPGM command.

Modules or service programs listed in a binding directory are used only if they provide an export that can satisfy any currently unresolved import requests. Entries in the binding directory may refer to objects that do not yet exist at the time the binding directory is created, but exist later.

Creating a Binding Directory

If you want to create a binding directory, use the Create Binding Directory (CRTBNDDIR) command to contain the names of modules and service programs that your ILE C/C++ program or service programs may need.

For example,

```
CRTBNDDIR BNDDIR(MYBNDDIR) MODULES (MOD1, MOD2)
CRTCMOD PROG(MYPROG) BNDDIR (MYBNDDIR)
```

or

```
CRTBNDDIR BNDDIR(MYBNDDIR) MODULES (MOD1, MOD2)
CRTCPPMOD PROG(MYPROG) BNDDIR (MYBNDDIR)
```

Using the Binder to Create a Program

The binder is invoked through the Create Program (CRTPGM) or the Create Service Program (CRTSRVPGM) commands. The CRTPGM command creates a program object from one or more module object objects and, if required, binds to one or more service programs. The CRTSRVPGM command creates a service program object from one or more module objects and, if required, binds to one or more service programs. See [“Service Programs” on page 22](#) for more information about service programs.

The CRTPGM and CRTSRVPGM commands invoke an IBM i component referred to as the *binder*. The *binder* processes import requests for procedure names and data item names from specified modules. The binder then tries to find matching exports in the specified modules, service programs, and binding directories. An *export* is an external symbol defined in a module or program that is available for use by other modules or programs. An *import* is the use of, or reference to, the name of a procedure or data item that is not defined in the current module object.

You can bind modules created by the compiler with modules created by any of the other ILE Create Module commands, including CRTRPGMOD, CRTCMOD, CRTCLMOD, or CRTCLMOD, or other ILE compilers.

Note: The modules or service programs to be bound must already have been created.

Preparing to Create a Program

Before you create a program object using the CRTPGM command, you should:

1. Establish a program name.
2. Identify the module(s) and, if required, the service programs you want to bind into a program object.

3. Make sure that the program has a program entry procedure that gets control when a dynamic program call is made. (That is, one module must contain the `main()` function of the program.)

You indicate which module contains the program entry procedure through the `ENTMOD` parameter. The default is `ENTMOD(*FIRST)`, which means that the module containing the first program entry procedure found in the list for the `MODULE` parameter is the entry module.

If you are binding more than one ILE module together, you should specify `ENTMOD(*FIRST)` or else specify the module name with the program entry procedure. You can use `ENTMOD(*ONLY)` when you are binding only one module into a program object, or if you are binding several modules but only one contains a program entry procedure. For example, if you bind a module with a `main()` function to a C module without a `main()` function, you can specify `ENTMOD(*ONLY)`.

4. Identify the activation group that the program is to use.

Specify `ACTGRP(*NEW)` if your program has no special requirements or if you are not sure which group to use.

Note that `ACTGRP(*NEW)` is the default activation group for CRTPGM. This means that your program will run in its own activation group, and the activation group will terminate once the program terminates. This default ensures that your program has a refresh of the resources necessary to run, every time you call it.

See [“Activating Groups” on page 21](#) for more information on unnamed and named activation groups.

Specifying Parameters for the CRTPGM Command

Table 3 on page 18 lists CRTPGM command parameters and their default values. Each parameter has default values which are used whenever you do not specify your own values.

Note: For a detailed description of the parameters, see the *CL and APIs* section in the *Programming* category at the IBM i Information Center web site: <http://www.ibm.com/systems/i/infocenter>.

Table 3. Parameters for CRTPGM Command and Their Default Values	
Parameter Group	Parameter (Default Value)
Identification	PGM(<i>library name/program name</i>) MODULE(*PGM) TEXT(*ENTMODTXT)
Program access	ENTMOD(*FIRST)
Binding	BNSRVPGM(*NONE) BNDDIR(*NONE)
Runtime	ACTGRP(*NEW)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWLIBUPD(*NO) USRPRF(*USER) REPLACE(*YES) AUT(*LIBCRTAUT) TGTRLS(*CURRENT) ALWRINZ(*NO) STGMDL(*SINGLVL) IPA(*NO) IPACTLFILE(*NONE)

How Import Requests Are Resolved

Whenever modules from different sources are combined into a single program, the compiler might have to process duplicate symbols.

Whenever you enter a CRTPGM command, the ILE compiler resolves import requests by:

1. Copying listed modules into what will become the program object and links any service programs to the program object.
2. Identifying the module containing the program entry procedure and locates the first import in this module.
3. Checking the modules in the order in which they are listed and matches the first import with a module export.
4. Returning to the first module and locates the next import.
5. Resolving all imports in the first module.
6. Continuing to the next module and resolving all imports in each subsequent module until all imports have been resolved.

After all the imports have been resolved, the ILE compiler completes the binding process and creates the program object. If any imports cannot be resolved with an export, the compiler terminates the binding process without creating a program object.

Note: If you have specified in the binder language that a variable is to be exported (using the EXPORT keyword), it is possible that the variable name will be identical to a variable in another procedure within the bound program object. You can use the **DUPPROC* option on the CRTPGM *OPTION* parameter to allow duplicate procedure names. See the *ILE C/C++ Compiler Reference* for further information on how to handle this situation.

Using a Binder Listing

The binding process can optionally produce a binder listing that describes the resources used, symbols and objects encountered, and problems that were resolved, or not resolved, in the binding process.

The listing is produced as a spooled file for the job you use to enter the CRTPGM command. You can choose a *DETAIL* parameter value to generate the listing at three levels of detail:

- **BASIC*
- **EXTENDED*
- **FULL*

The default is not to generate a listing. If it is generated, the binder listing includes the sections described in [Table 4 on page 19](#), depending on the value specified for *DETAIL*.

Section Name	<i>*BASIC</i>	<i>*EXTENDED</i>	<i>*FULL</i>
Command Option Summary	X	X	X
Brief Summary Table	X	X	X
Extended Summary Table		X	X
Binder Information Listing		X	X
Cross-Reference Listing			X
Binding Statistics			X

The information in this listing can help you diagnose problems if the binding was not successful, or give feedback about what the binder encountered during the binding process.

Figure 4 on page 20 shows the basic binder listing for a program CVTHEXPGM. Note that this listing is taken out of context. It only serves to illustrate the type of information you may find in a binder listing.

XXXXXXXXXXXXXX

XXXXXXXXXXXXXX

```

Create Program
5722SS1 V5R1M0 010525 MYLIB /CVTHEXPGM TORAS597 00/12/07 16:25:32 Page 1
Program . . . . . : CVTHEXPGM
Library . . . . . : MYLIB
Program entry procedure module . . . . . : *FIRST
Library . . . . . :
Activation group . . . . . : *NEW
Creation options . . . . . : *GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF
Listing detail . . . . . : *BASIC
Allow Update . . . . . : *YES
Allow bound *SRVPGM library name update . . . . . : *NO
User profile . . . . . : *USER
Replace existing program . . . . . : *YES
Authority . . . . . : *LIBCRTAUT
Target release . . . . . : *CURRENT
Allow reinitialization . . . . . : *NO
Storage model . . . . . : *SINGLVL
Interprocedural analysis . . . . . : *NO
IPA control file . . . . . : *NONE
IPA replace IL data . . . . . : *NO
Text . . . . . : *ENTMODTXT

Create Program
5722SS1 V5R1M0 010525 MYLIB/CVTHEXPGM TORAS597 Page 2
00/12/07
16:25:32
Module Library Module Library Module Library Module Library
CVTHEXPGM MYLIB
Service Service Service Service
Program Library Program Library Program Library Program Library
*NONE
Binding Binding Binding Binding
Directory Library Directory Library Directory Library Directory Library
*NONE

Create Program
5722SS1 V5R1M0 010525 MYLIB/CVTHEXPGM TORAS597 Page 3
00/12/07
16:25:32
Program entry procedures . . . . . : 1
Symbol Type Library Object Bound Identifier
*MODULE MYLIB CVTHEXPGM *YES _CXX_PEP_Fv
Multiple strong definitions . . . . . : 0
Unresolved references . . . . . : 0
***** END OF BRIEF SUMMARY TABLE *****

Create Program
5722SS1 V5R1M0 010525 MYLIB/CVTHEXPGM TORAS597 Page 4
00/12/07
16:25:32
Binding Statistics
Symbol collection CPU time . . . . . : .001
Symbol resolution CPU time . . . . . : .000
Binding directory resolution CPU time . . . . . : .158
Binder language compilation CPU time . . . . . : .000
Listing creation CPU time . . . . . : .015
Program/service program creation CPU time . . . . . : .030
Total CPU time . . . . . : .562
Total elapsed time . . . . . : 2.618
***** END OF BINDING STATISTICS *****
*CPC5D07 - Program CVTHEXPGM created in library MYLIB.
***** END OF CREATE PROGRAM LISTING *****

```

Figure 4. Example of a Basic Binder Listing

Updating a Module or a Program Object

There are many reasons why you may want to change a module or a program object:

- An object may need to be changed to accommodate enhancements, or for maintenance reasons.

You can isolate what needs to be changed by using debugging information or the binder listing from the CRTPGM command. From this information you can determine what modules, procedures, or fields need to change.

- You may want to change the optimization level or observability of a module or program.

This is often the case when you want to debug a program or module, or when you are ready to put a program into production. Such changes can be performed more quickly and use fewer system resources than the re-creation of the object in question.

- You may want to reduce the program size once you have completed an application.

ILE program objects have additional data added to them, which makes them larger than similar OPM or EPM program objects.

Each of the above approaches requires different data to make the change.

Updating a Program

In general, you can update a program by replacing modules as needed. You do not have to re-create the program object. The ability to replace specific modules is helpful if, for example, you are supplying an application to other sites that are already using the program. You need only send the revised modules, and the receiving site can update the application using the UPDPGM and UPDSRVPGM commands.

The update commands work with both program and module objects. The parameters for these commands are very similar to those for the Create Program (CRTPGM) command. For example, to replace a module in a program, you would enter the module name for the *MODULE* parameter and the library name.

To use the UPDPGM command, the modules to be replaced must be located in the same libraries they were in when the program was created. You can specify that all modules, or only some subsets of modules, are to be replaced.

Activating Groups

Activation is the process used to prepare an ILE program to run. Activation allocates and initializes static storage for an ILE program, and completes the binding of ILE programs to ILE service programs. The ACTGRP parameter on the CRTPGM and CRTSRVPGM commands specifies the activation group in which a program or service program runs.

All ILE programs and service programs are activated within a substructure of a job called an *activation group*. This substructure contains the resources necessary to run the ILE programs. The static and automatic program variables and dynamic storage are assigned separate address spaces for each activation group. Activation and activation groups:

- Help ensure that ILE programs running in the same job run independently without intruding on each other (for example, commitment control, overrides, shared files) by scoping resources to the activation group.
- Scope resources to the ILE program.
- Uniquely allocate the static data needed by the ILE program or service program.
- Change the symbolic links to ILE service programs into physical addresses.

Messaging Support

The following table describes the level of compiler messages that you could receive during compilation of your source code

Severity	Compiler Response
Informational (00)	Compilation continues. The message reports conditions found during compilation.
Warning (10)	Compilation continues. The message reports valid, but possibly unintended, conditions.
Error (20)	Compilation continues and object code is generated. Error conditions exist that the compiler can correct, but the program might not run correctly.
Severe error (30)	Compilation continues, but object code is not generated. Error conditions exist that the compiler cannot correct.

Table 5. Compiler Messages (continued)	
Severity	Compiler Response
Unrecoverable error (40)	The compiler halts. An internal compiler error has been found. This message should be reported to your IBM service representative.

Service Programs

A service program is an IBM i object of type *SRVPGM. Service programs are typically used for common functions that are frequently called by other procedures within an application and across applications. For example, the ILE compilers use service programs to provide runtime services such as math functions and input/output functions.

Service programs simplify maintenance, and reduce storage requirements, because only a single copy of a service program is maintained and stored.

This topic describes:

- [The difference between programs and service programs](#)
- [Public interface](#)
- [Considerations when creating a service program](#)
- [How to use the binder to create a service program](#)
- [How to create a service program](#)

Differences Between Programs and Service Programs

A service program differs from a program in two ways:

- A service program is bound to existing programs or other service programs. It cannot run independently.
- A service program does not contain a program entry procedure. Therefore, you cannot call a service program using an OS linkage specification. However, you can call a service program with a c linkage specification, because it contains at least one user entry procedure. A service program may have data exports rather than a user entry procedure.
- Service programs are bound *by reference*. This means that the content of the service program is not copied into the program to which it is bound. Instead, *linkage information* about the service program is bound into the program.

This process is different from the static binding process used to bind modules into programs. However, you can still call the service program's exported procedures as if they were statically bound. The initial activation is longer, but subsequent calls to any of the service program's exported procedures are faster than program calls.

Public Interface

The *public interface* of a service program consists of the names of the exported procedures and data items that can be referenced by other ILE objects. In order to be exported from an ILE service program, a data item must be exported from one of the module objects making up the ILE service program.

The exports list is used to specify the public interface for a service program. A *signature* is generated from the procedure and data item names listed in the binder language. This signature can then be used to validate the interface to the service program.

As long as the public interface is unchanged, the clients of a service program do not have to be recompiled after a change to the service program.

Considerations When Creating a Service Program

When creating a service program, you should consider:

- Whether or not you intend to update the program at a later date
- Whether or not any updates involve changes to its interface

If the interface to a service program changes, you may have to rebind all programs bound to the original service program. However, depending on the changes and how you implement them, you may be able to reduce the amount of rebinding if you create the service program using *binder language*. In this case, after updating the binder language source to identify new exports, you need to rebind only those programs that require the new exports.

See “[Displaying Exported Defined Symbols with the Display Module Command](#)” on page 28 for additional information related to service-program exports.

Using the Binder to Create a Service Program

Creating a service program involves compiling source code into module objects, and then binding one or more module objects into a service program object with the Create Service Program (CRTSRVPGM) command. You can also use modules created with other ILE language compilers, such as ILE C/C++, ILE RPG, or ILE COBOL.

Specifying Parameters for the CRTSRVPGM Command

Table 6 on page 23 lists CRTSRVPGM command parameters and their default values.. For a detailed description of the parameters, refer to the CL Reference CHKxxx through CVTxxx Commands: SC41–5724. Each parameter has default values which are used whenever you do not specify your own values.

Parameter Group	Parameter (Default Value)
Identification	SRVPGM(<i>library name/service program name</i>) MODULE(*SRVPGM)
Program access	EXPORT(*SRCFILE) SRCFILE(*LIBL/QSRVSRC) SRCMBR(*SRVPGM)
Binding	BNDSRVPGM(*NONE) BNDDIR(*NONE)
Runtime	ACTGRP(*CALLER)
Miscellaneous	OPTION(*GEN *NODUPPROC *NODUPVAR *WARN *RSLVREF) DETAIL(*NONE) ALWUPD(*YES) ALWRINZ(*NO) REPLACE(*YES) AUT(*LIBCRTAUT) TEXT(*ENTMODTXT) TGTRLS(*CURRENT) USRPRF(*USER) STGMDL(*SINGLVL) IPA(*NO) IPACTLFILE(*NONE)

Updating or Changing a Service Program

You can update or change a service program in the same way you modify a program object. In other words, you can:

- Update the service program, using the Update Service Program (UPDSRVPGM) command
- Change the optimization level, using the Change Service Program (CHGSRVPGM) command
- Remove observability (using CHGSRVPGM)
- Reduce the size, using the Compress Object (CPROBJ) command

See [“Updating a Module or a Program Object”](#) on page 20 for more information on any of the above points.

If you use binder language, a service program can be updated without requiring programs calling it to be recompiled. For example, to add a new procedure to an existing service program:

1. Create a module object for the new procedure.
2. Modify the binder-language source file to handle the interface associated with the new procedure. Add any new export statements following the existing ones. See [“Updating a Service Program Export List”](#) on page 30 for details on modifying binder-language source files.
3. Recreate the original service program and include the new module.

Now existing programs can access the new functions. Because the old exports are in the same order, they can still be used by the existing programs. Until it is necessary to also update the existing programs, they do not have to be recompiled.

Using Control Language (CL) Commands with Service Programs

The following CL commands can be used with service programs:

- Create Service Program (CRTSRVPGM)
- Change Service Program (CHGSRVPGM)
- Display Service Program (DPSRVPGM)
- Delete Service Program (DLTSRVPGM)
- Update Service Program (UPDSRVPGM)
- Work with Service Program (WRKSRVPGM).

Creating, Compiling, and Binding a Service Program

The example in this section is used to show how to create a service program SEARCH that can be called by other programs to locate a character string in any given string of characters.

This section describes how to:

- [Create the source files](#)
- [Compile and bind the service program](#)
- [Bind the service program to a program](#)

Creating the Source Files

The SEARCH program is implemented as a class object Search. The class Search contains:

- Three private data members: `skippat`, `needle_p`, and `needle_size`
- Three constructors, each taking different arguments
- A destructor
- An overloaded function `where()`, which takes four different sets of arguments

The service program is composed of the following files:

- A user-defined header file `search.h`
- A source code file `search.cpp`
- A source code file `where.cpp`

User Header File

The class and function declarations are placed into a separate header file, `search.h`, as shown in the following figure:

```
// header file search.h
// contains declarations for class Search, and inlined function
// definitions

#include <iostream.h>

class Search {
private:
    char skippat[256];
    char * needle_p;
    int needle_size;
public:

// Constructors
    Search( unsigned char * needle, int size);
    Search ( unsigned char * needle);
    Search ( char * needle);

//Destructor
    ~Search () { delete needle_p;}

//Overloaded member functions
    unsigned int where ( char * haystack) {
        return where (haystack, strlen(haystack));
    }
    unsigned int where ( unsigned char * haystack) {
        return where (haystack, strlen((const char *)haystack));
    }
    unsigned int where ( char * haystack, int size) {
        return where ( (unsigned char *) haystack, size);
    }
    unsigned int where ( unsigned char * haystack, int size);
};
```

Figure 5. Example of Header File (`search.h`)

Source Code Files

If the definitions for the member functions of class `Search` are not inlined in the class declaration, they are contained in two separate files:

- The source file `search.cpp`, which contains constructor definitions for class `Search`
- The file `where.cpp`, which contains the member function definition.

These files are shown in the following figures:

```

// source file search.cpp
// contains the definitions for the constructors for class Search

#include "search.h"

Search::Search( unsigned char * needle, int size)
: needle_size(size) , needle_p ( new char [size])
{
    memset (skippat, needle_size, 256);
    for (unsigned int i=0; i<size; ++i) {
        skippat [needle [i]] = size -i-1;
    }
    memcpy (needle_p, needle, needle_size);
}
Search::Search ( unsigned char * needle) {
    needle_size = strlen( (const char *)needle) ;
    needle_p = new char [needle_size];
    memset (skippat, needle_size, 256);
    for (unsigned int i=0; i<needle_size; ++i) {
        skippat [needle [i]] = needle_size -i-1;
    }
    memcpy(needle_p, needle, needle_size);
}
Search::Search ( char * needle) {
    needle_size = strlen( needle) ;
    needle_p = new char [needle_size];
    memset (skippat,needle_size, 256);
    for (unsigned int i=0; i<needle_size; ++i) {
        skippat [needle [i]] = needle_size -i-1;
    }
    memcpy(needle_p, needle, needle_size);
}
}

```

Figure 6. Source File that Contains Constructor Definitions for the Search Class

Figure 7. File that Contains the Member Function Definition for the Search Class

```

// where.cpp
// contains definition of overloaded member function for class Search

#include "search.h"

unsigned int Search:: where ( unsigned char * haystack, int size)
{
    unsigned int i, t;
    int j;
    for ( i= needle_size-1, j = needle_size-1; j >= 0; --i, --j ){
        while ( haystack[i] != needle_p[j]) {
            t = skippat [ haystack [i]] ;
            i += (needle_size - j > t) ? needle_size - j : t ;
            if (i >= size)
                return size;
            j = needle_size - 1;
        }
    }
    return ++i;
}
}

```

The modules that result from the compilation of these source files, SEARCH and WHERE, are bound into a service program, SERVICE1.

Compiling and Binding the Service Program

To create the service program SERVICE1, issue the following commands:

```

CRTCPPMOD MODULE(MYLIB/SEARCH) SRCSTMF(search.cpp)
CRTCPPMOD MODULE(MYLIB/WHERE) SRCSTMF(where.cpp)
CRTSRVPGM SRVPGM(MYLIB/SERVICE1) MODULE(MYLIB/SEARCH MYLIB/WHERE) EXPORT(*ALL)

```

By default, the binder creates the service program in your current library.

The parameter *EXPORT(*ALL)* specifies that all data and procedures exported from the modules are also exported from the service program.

Binding the Service Program to a Program

In the following example, a very short application consisting of a program MYPROGA is bound to the service program. The source code for MYPROGA, MYPROGA.cpp, is shown in the following figure.

Note: This sample application has been reduced to minimal functionality. Its main purpose is to demonstrate how to create a service program.

```
// myproga.cpp
// Finds a character string in another character string.

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include "search.h"
#define HS "Find the needle in this haystack"

void main () {
    int i;
    Search token("needle");
    i = token.where (HS, sizeof(HS));
    cout << "The string was found in position " << i << endl;
}
```

Figure 8. Source Code for myproga.cpp

The program creates an object of class Search. It invokes the constructor with a value that represents the string of characters ("needle") to be searched for. It calls the member function where () with the string to be searched ("Find the needle in this haystack"). The string "needle" is located, and its position in the target string "Find a needle in this haystack" is returned and printed.

To create the program MYPROGA in library MYLIB, and bind it to the service program SERVICE1, enter the following:

```
CRTPGM PGM(MYLIB/MYPROGA) SRCSTMF(myprogA.cpp) BNDSRVPGM(MYLIB/SERVICE1)
```

Figure 9 on page 27 shows the internal and external function calls between program MYPROGA and service program SERVICE1.

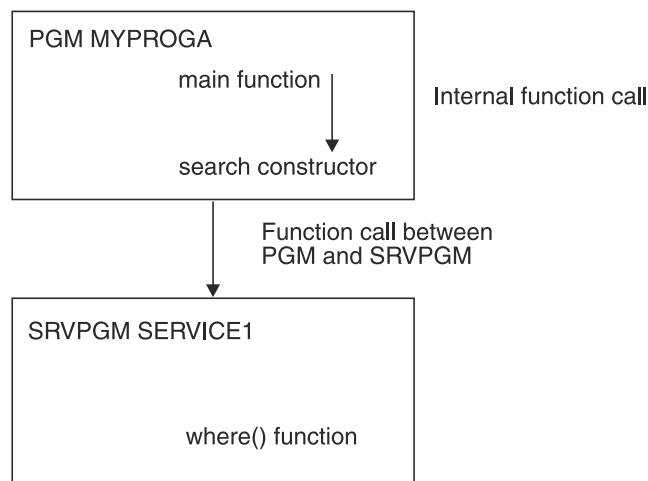


Figure 9. Calls between Program and Service Program

When MYPROGA is created, it includes information regarding the interface it uses to interact with the service program.

To run the program, enter:

```
CALL MYLIB/MYPROGA
```

During the process of making MYPROGA ready to run, the system verifies that:

- The service program SERVICE1 in library MYLIB can be found.
- The public interface used by MYPROGA when it was created is still valid at runtime.

If either of the above is not true, an error message is issued.

The output of MYPROGA is:

```
The string was found in position 9
```

Working with Exports from Service Programs

This section describes how to work with procedures and data items that can be exported from a service program.

Determining Exports from Service Programs

A service program exports procedures and data items that can be imported by other programs. These exports represent the *interface* to the service program. In the C/C++ programming language, procedures and data items correspond to functions and variables.

Information about exports, which can be derived from the modules that form a particular service program, may be used to create a binder language source file which then defines the interface to this service program. A *binder language source file* specifies the exports the service program makes available to all programs that call it. This file can be specified on the EXPORT parameter of the CRTSRVPGM command.

Binder language gives you better control over the exports of a service program. This control can be very useful if you want to:

- Determine export and import mismatches in an application.
- Add functionality to service programs.
- Reduce the impact of changes to a service program on the users of an application.
- Mask certain service program exports from service program users. That is, by not listing certain functions or variables in the binder language source file, you can prevent any calling programs from having access to these exports.

Displaying Exported Defined Symbols with the Display Module Command

To find out which exports are available from a module, enter:

```
DSPMOD MODULE(library-name/module-name)
```

Specify the module name and the library where the module is stored.

This command opens the Display Module Information display. At the bottom of this display, you find the name and type of each symbol that can be exported from the module.

Note: When the compiler compiles a source file, it encodes function names and certain variables to include type and scoping information. This encoding process is called *name mangling*. The symbol names in the sample display below are shown in mangled form. The source code for module SEARCH is shown in [“Source Code Files”](#) on page 25.

```

Display Module Information          Display 3 of 3
Module . . . . . : SEARCH
Library . . . . . : MYLIB
Detail . . . . . : *EXPORT
Module attribute . . . . . :
Exported defined symbols:
Symbol Name                               Symbol Type
__ct__6SearchFPc                          PROCEDURE
__ct__6SearchFPUC                          PROCEDURE
__ct__6SearchFPUci                        PROCEDURE

```

Figure 10. Display Module Information Screen for a Sample Module SEARCH

Creating a Binder Language Source File

Binder language is based on the exports available from modules that are bound into service programs. A binder language source file must contain the following entries:

- The Start Program Export (STRPGMEXP) command identifies the beginning of the list of exports from the service program.
- Export Symbol (EXPORT) commands identify each a symbol name available to be exported from the service program.
- The End Program Export (ENDPGMEXP) command identifies the end of the list of exports from the service program.

The following figure shows the structure of a binder language source file:

```

STRPGEXP PGMLEVEL(*CURRENT)
EXPORT SYMBOL("mangled_procedure_name_a")
EXPORT SYMBOL("mangled_procedure_name_b")
...
EXPORT SYMBOL("mangled_procedure_name_x")
ENDPGMEXP

```

Figure 11. Example of a Binder Language Source File

Note: You must specify the mangled name of each symbol on the EXPORT command, because the binder looks for the mangled names of exports when it tries to resolve import requests from other modules.

After all the modules to be bound into a service program have been created, you can create the binder language source file by using either of the following methods:

- You can write this file yourself, [using the Source Entry Utility \(SEU\)](#).
- You can generate a skeleton binding source by using the [Retrieve Binder Source \(RTVBNSRC\)](#) command.

Creating Binder Language Using SEU

To use the Source Entry Utility (SEU) to create a binder language source file, follow these steps:

1. Create a source physical file QSRVSR in library MYLIB.
2. Create a member MEMBER1 that will contain the binder language.
3. Use the Display Module (DSPMOD) command to display the symbols that can be exported from each module.
4. Decide which exports you want to make available to calling programs.
5. Use the Source Entry Utility (SEU) to enter the syntax of the binder language.

You need one export statement for each procedure whose exports you want to make available to the caller of the service program. Do not list symbols that you do not want to make available to calling programs.

For example, based on the information shown in [Figure 10 on page 29](#), the binder language source file for module SEARCH could list the following export symbols:

```
STRPGEXP PGMLEVEL(*CURRENT)
EXPORT SYMBOL("__ct__6SearchFPc")
EXPORT SYMBOL("__ct__6SearchFPUc")
EXPORT SYMBOL("__ct__6SearchFPUci")
ENDPGMEXP
```

Creating Binder Language Using the RTVBNSRC Command

The Retrieve Binder Source (RTVBNSRC) command can automatically create a binder language source file. It retrieves the exports from a module, or a set of modules. It generates the binder language for these exports, and places exports and binder language in a specified file member. This file member can later be used as input to the *EXPORT* parameter of the Create Service Program (CRTSRVPGM) command.

Note: After the binder language has been retrieved into a source file member, you can edit the binder language and modify it as needed (for example, if you make changes to a module or if you want to make certain exports unavailable to calling programs).

The syntax for the RTVBNSRC command is:

```
RTVBNSRC MODULE(MYLIB/SEARCH) SRCFILE(MYLIB/QSRVSR) SRCMBR(*DFT) MBROPT(*REPLACE)
```

For detailed information on the RTVBNSRC command and its parameters enter RTVBNSRC on a command line and press F1 for Help.

The following example shows how to create a binder language source file for module SEARCH, located in library MYLIB, using the RTVBNSRC command. The source code for module SEARCH is shown in ["Source Code Files" on page 25](#).

```
RTVBNSRC MODULE(MYLIB/SEARCH) SRCFILE(MYLIB/QSRVSR) SRCMBR(ONE)
```

This command automatically:

1. Creates a source physical file QSRVSR in library MYLIB.
2. Adds a member ONE to QSRVSR.
3. Generates binder language from module SEARCH in library MYLIB and places it in member ONE.

Member ONE in file MYLIB/QSRVSR now contains the following binder language:

```
Columns . . . : 1 71          Browse          MYLIB/QSRVSR
SEU=>          ONE
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
0000.01 STRPGMEXP PGMVL(*CURRENT)
0000.02 /*****
0000.03 /*      *MODULE      SEARCH      MYLIB      95/06/10  17:34:41      */
0000.04 /*****
0000.05 EXPORT SYMBOL("__ct__6SearchFPc")
0000.06 EXPORT SYMBOL("__ct__6SearchFPUc")
0000.07 EXPORT SYMBOL("__ct__6SearchFPUci")
0000.08 ENDPGMEXP
***** End of data *****
```

Figure 12. Binder Language Source File Generated for Module SEARCH

Updating a Service Program Export List

You can use binder language to reflect changes in the list of exports a service program makes available. When you create binder language, a signature is generated from the order in which the modules that form a service program are processed, and from the order in which symbols are exported from these modules. The *EXPORT* keyword in the binder language identifies the procedure and data item names that make up the signature for the service program.

When you make changes to the exports of a service program this does not necessarily mean that all programs that call this service program must be re-created. You can implement changes in the binder language such that they are backward-compatible. Backward-compatible means that programs which depend on exports that remain unchanged do not need to be re-created.

To ensure backward compatibility, add new procedure or data item names to the end of the export list, and re-create the service program with the same signature. This lets existing programs still use the service program, because the order of the unchanged exports remains the same.

Note: When changes to a service program result in a loss of exports, or in a change of existing exports, it becomes difficult to update the export list without affecting existing programs that require its services. Changes in the order, number, or name of exports result in a new signature that requires the re-creation of all programs and service programs that use the changed service program.

Using the Demangling Functions

You can retrieve the mangled names of exported symbols with the `RTVBNDSRC` command. To help you find the corresponding demangled names, the runtime library contains a small class hierarchy of functions that you can use to demangle names and examine the resulting parts of the name. The interface is documented in the `<demangle.h>` header file.

Using the demangling functions, you can write programs to convert a mangled name to a demangled name and to determine characteristics of that name, such as its type qualifiers or scope. For example, given the mangled name of a function, the program returns the demangled name of the function and the names of its qualifiers. If the mangled name refers to a class member, you can determine if it is `static`, `const`, or `volatile`. You can also get the whole text of the mangled name.

To demangle a name, which is represented as a character array, create a dynamic instance of the `Name` class and provide the character string to the class's constructor. For example, to demangle the name `f__1XFi`, create:

```
char *rest;
Name *name = Demangle("f__1XFi", rest);
```

The demangling functions classify names into five categories: function names, member function names, special names, class names, and member variable names. After you construct an instance of class `Name`, you can use the `Kind` member function of `Name` to determine what kind of `Name` the instance is. Based on the kind of name returned, you can ask for the text of the different parts of the name or of the entire name.

For the mangled name `f__1XFi`, you can determine:

```
name->Kind() == MemberFunction
((MemberFunctionName *) name)->Scope()->Text() is "X"
((MemberFunctionName *) name)->RootName() is "f"
((MemberFunctionName *) name)->Text() is "X:f(int)"
```

If the character string passed to the `Name` constructor is not a mangled name, the `Demangle` function returns `NULL`.

For further details about the demangling functions, refer to the information contained in the `demangle.h` header file. If you installed ILE C/C++ using default settings, this header file should be in IFS in the `/QIBM/include` directory and in DM in `QSYSINC/H`.

Handling Unresolved Import Requests During Program Creation

An *unresolved import* is an import whose type and name do not yet match the type and name of an export. Unresolved import requests do not necessarily prevent you from creating a program or a service program. You can proceed in two ways:

- Specify the `*UNRSLVREF` option on the `CRTPGM` or `CRTSRVPGM` commands to tell the binder to go ahead and create a program or service program, even if there are imports in the modules, and no matching exports can be found.

- Change the order of program creation to avoid unresolved references.

Both approaches are demonstrated in [“Creating a Program with Circular References”](#) on page 32.

Use the **UNRSLVREF* option to convert, create, or build pieces of code when all the pieces of code are not yet available. After the development or conversion phase has finished and all import requests can be resolved, make sure you re-create the program or service program that has the unresolved imports.

If you use the **UNRSLVREF* option, specify *DETAIL(*EXTENDED)* or *DETAIL(*FULL)*, or keep the job log when the object is created, to identify the procedure or data item names that are not found.

Note: If you have specified **UNRSLVREF* and a program is created with unresolved import requests, you receive an error message (MCH3203) when you try to run the program.

Creating an Export Service Program Using Binder Language

The Create C++ Module (CRTCPPMOD) command creates only one module at a time. You must use the CRTCPPMOD for each source stream file or source file member. The following example consists of two modules: SEARCH asnd WHERE.

Example:

To use binder language to create the service program described in [“Creating, Compiling, and Binding a Service Program”](#) on page 24, follow these steps:

1. To create modules from all source files enter the following commands:

```
CRTCPPMOD MODULE(MYLIB/SEARCH) SRCSTMF(search.cpp)
CRTCPPMOD MODULE(MYLIB/WHERE) SRCSTMF(where.cpp)
```

Note: The CRTCPPMOD command stops the compilation process after the creation of the *MODULE object. The binder is not invoked.

2. To create the corresponding binder language source file, enter the following command:

```
RTVBNDSRC MODULE(MYLIB/SEARCH MYLIB/WHERE)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(TWO)
```

This command creates the binder language source file shown in [Figure 13](#) on page 32.

3. To create service program SERVICE2, enter the following command:

```
CRTSRVPGM SRVPGM(MYLIB/SERVICE2) MODULE(MYLIB/SEARCH MYLIB/WHERE)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(TWO)
```

```
Columns . . . . : 1 71          Browse          MYLIB/QSRVSRC
SEU=>> TWO
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****
0000.03 /* *MODULE SEARCH MYLIB 95/06/11 15:30:51*/
0000.04 /*****
0000.05 EXPORT SYMBOL("__ct__6SearchFPc")
0000.06 EXPORT SYMBOL("__ct__6SearchFPuc")
0000.07 EXPORT SYMBOL("__ct__6SearchFPuci")
0000.08 /*****
0000.09 /* *MODULE WHERE MYLIB 95/06/11 15:30:51*/
0000.10 /*****
0000.11 EXPORT SYMBOL("where__6SearchFPuci")
0000.12 ENDPGMEXP
***** End of data *****
```

Figure 13. Binder Language Source File Generated by the RTVBNDSRC Command

Creating a Program with Circular References



A *circular reference* is a special case of unresolved import requests. It occurs, for example, when a service program SP1 depends on imports from a service program SP2, which in turn depends on an import from service program SP1.

Figure 14 on page 33 illustrates the unresolved import requests between program A and two service programs, SP1 and SP2.

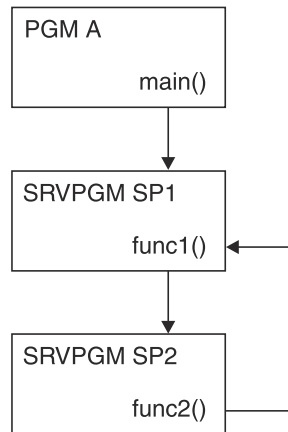


Figure 14. Unresolved Import Requests in a Program with Circular References

The following import requests occur between program A and the two service programs, SP1 and SP2, that are called by program A:

1. Program A uses function `func1()`, which it imports from service program SP1.
2. Service program SP1 needs to import function `func2()` provided by service program SP2, in order to provide `func1()` to program A.
3. Service program SP2, in turn, first needs to import `func1` from service program SP1 before being able to provide `func2`.

To create a program with unresolved circular references, perform the following tasks:

1. Create the source files.
2. Compile the source files into modules.
3. Create the binder language.
4. Bind the modules into the program.

Creating the Source Files

The application consists of three source files, `m1.cpp`, `m2.cpp`, and `m3.cpp`, shown in the following figures:

```
// m1.cpp
#include <iostream.h>
int main(void)
{
    void func1(int);
    int n = 0;
    func1(n);           // Function func1() is called.
}
```

Figure 15. `m1.cpp` – First Source File for Application with Circular References

```

// m2.cpp
#include <iostream.h>
void func2 (int);
void func1(int x)
{
    if (x<5)
    {
        x += 1;
        cout << "This is from func1(), n=" << x << endl;
        func2(x);          // Function func2() is called.
    }
}
}

```

Figure 16. m2.cpp – Second Source Files for Application with Circular References

```

// m3.cpp
#include <iostream.h>
void func1(int);
void func2(int y)
{
    if (y<5)
    {
        y += 1;
        cout << "This is from func2(), n=" << y << endl;
        func1(y);          // Function func1() is called.
    }
}
}

```

Figure 17. m3.cpp – Third Source File for Application with Circular References

Compiling the Source Files into Modules

Compile the source file m1.cpp into a module object from which you later create program A. This allows you to display their exports with the DSPMOD command, or to generate binder language source with the RTVBNSRSC command.

To create module objects from the source files described above, invoke the commands:

```

CRTCPPMOD MODULE(MYLIB/M1) SRCSTMF(m1.cpp)
CRTCPPMOD MODULE(MYLIB/M2) SRCSTMF(m2.cpp)
CRTCPPMOD MODULE(MYLIB/M3) SRCSTMF(m3.cpp)

```

The CRTCPPMOD compiler option indicates to the compiler that you do not want to create a program object from the source file. The target library is specified by the MODULE option.

Generating the Binder Language to Create the Service Program

To generate binder language for module M2, from which you want to create service program SP1, enter the following command:

```

RTVBNSRSC MODULE(MYLIB/M2) SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)

```

This command results in the following binder language being created for module M2, in library MYLIB, source file QSRVSRC, file member BNDLANG1:

```

Columns . . . : 1 71          Browse          MYLIB/QSRVSRC
SEU==>
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****
0000.03 /*  *MODULE      M2          MYLIB      95/06/11 18:07:04*/
0000.04 /*****
0000.05 EXPORT SYMBOL("func1__Fi")
0000.06 ENDPGMEXP
***** End of data *****

```

Figure 18. Binder Language for Service Program SP1

To generate binder language for module M3, from which you want to create service program SP2, issue the following command:

```
RTVBNSRC MODULE(MYLIB/M3) SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2)
```

This command results in the following binder language being created for module M3, in library MYLIB, source file QSRVSRC, file member BNDLANG2:

```
Columns . . . : 1 71          Browse          MYLIB/QSRVSRC
SEU=>
FMT **  ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
***** Beginning of data *****
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****
0000.03 /* *MODULE      M3          MYLIB      95/06/11  18:08:14  */
0000.04 /*****
0000.05     EXPORT SYMBOL("func2__Fi")
0000.06 ENDPGMEXP
***** End of data *****
```

Figure 19. Binder Language for Service Program SP2

Binding the Modules into the Program

Program A will be created from M1. Service program SP1 will be created from M2. Service program SP2 is created from M3.

If you try and create service program SP1 from module M2, using the binder language shown in [Figure 18](#) on [page 34](#) and the compiler invocation:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)
```

you find that the binder tries to resolve the import for function `func2()`, but fails, because it is not able to find a matching export. Therefore, service program SP1 is not created.

If SP1 is not created, this leads to problems if you try and create service program SP2 from module M3 using the binder language shown in [Figure 19](#) on [page 35](#) and the compiler invocation:

```
CRTSRVPGM SRVPGM(MYLIB/SP2) MODULE(MYLIB/M3)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2)
```

Service program SP2 is not created, because the binder fails in searching for the import for `func1()` in service program SP1, which has not been created in the previous step.

If you try and create program A with the compiler invocation:

```
CRTPGM PGM(A) MODULE(MYLIB/M1) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)
```

the binder fails, because service programs SP1 and SP2 do not exist.

Handling Unresolved Import Requests Using the *UNRSLVREF Parameter

The following example shows:

- How to create service program SP1 from `m2.cpp`, shown in
- How to use the parameter `*UNRSLVREF` to handle the unresolved import requests which would otherwise prevent you from creating program A.

The example

Example:

1. To create service program SP1 from m2 . cpp, enter:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)
OPTION(*UNRSLVREF)
```

Because the **UNRSLVREF* option is specified, service program SP1 is created even though the import request for `func2()` is not resolved.

2. To create service program SP2 from module M3 and service program SP1, enter:

```
CRTSRVPGM SRVPGM(MYLIB/SP2) MODULE(MYLIB/M3)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2)
BNDSRVPGM(MYLIB/SP1)
```

Because service program SP1 now exists, the binder resolves all the import requests required, and service program SP2 is created successfully.

3. To re-create the service program SP1, enter:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)
BNDSRVPGM(MYLIB/SP2)
```

Although service program SP1 does exist, the import request for `func2()` is not resolved. Therefore, the re-creation of service program SP1 is required. Because service program SP2 now exists, the binder resolves all import requests required and, service program SP1 is created successfully.

4. To create program A, enter:

```
CRTPGM PGM(MYLIB/A) MODULE(MYLIB/M1) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)
```

Because service programs SP1 and SP2 do exist, the binder creates the program A.

Handling Unresolved Import Requests by Changing Program Creation Order

You can also change the order of program creation to avoid unresolved references, by first creating a service program with all modules, and then re-creating this same service program later.

1. To generate binder language for modules M2 and M3, from which you want to create service program SP1, issue the following command:

```
RTVBNDSRC MODULE(MYLIB/M2 MYLIB/M3) SRCFILE(MYLIB/QSRVSRC)
SRCMBR(BNDLANG3)
```

This command results in the binder language shown in [Figure 20 on page 37](#) being created in library MYLIB, source file QSRVSRC, file member BNDLANG3.

2. To create service program SP1 from module M2 and module M3 enter:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2 MYLIB/M3)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG3)
```

Because modules M2 and M3 are specified, all import requests are resolved and service program SP1 is created successfully.

3. To create service program SP2, enter:

```
CRTSRVPGM SRVPGM(MYLIB/SP2) MODULE(MYLIB/M3)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG2)
BNDSRVPGM(MYLIB/SP1)
```

Because service program SP1 exists, the binder resolves all the import requests required and service program SP2 is created successfully.

4. To re-create service program SP1, enter:

```
CRTSRVPGM SRVPGM(MYLIB/SP1) MODULE(MYLIB/M2)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BNDLANG1)
BNDSRVPGM(MYLIB/SP2)
```

Although service program SP1 does exist, the import request for `func2()` is not resolved to the one in service program SP2. Therefore, a re-creation of service program SP1 is necessary to make the circular reference work.

Because service program SP2 now exists, the binder can resolve the import request for `func2()` from service program SP2, and service program SP1 is successfully created.

5. To create program A, enter:

```
CRTPGM PGM(MYLIB/A) MODULE(MYLIB/M1) BNDSRVPGM(MYLIB/SP1 MYLIB/SP2)
```

Because service programs SP1 and SP2 do exist, the binder creates program A.

```
Columns . . . :   1  71          Browse          MYLIB/QSRVSRC
SEU==>          BNDLANG3
FMT **   ...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7
          ***** Beginning of data *****
0000.01 STRPGMEXP PGMLVL(*CURRENT)
0000.02 /*****
0000.03 /*  *MODULE      M2          MYLIB      95/06/11  18:50:23  */
0000.04 /*****
0000.05 EXPORT SYMBOL("func1__Fi")
0000.06 /*****
0000.07 /*  *MODULE      M3          MYLIB      95/06/11  18:50:23  */
0000.08 /*****
0000.09 EXPORT SYMBOL("func2__Fi")
0000.10 ENDPGMEXP
          ***** End of data *****
```

Figure 20. Binder Language for Service Program SP1

Binding a Program to a Non-Existent Service Program

To successfully create a program or a service program, all required modules must exist prior to invoking the binder.

However, if you want to bind a program to a non-existent service program, you can create a "placeholder" service program first. Consider the following example:

A program `MYPROG` requires a function `print()` to be exported by a service program `PRINT`. The code for the program is available in `myprog.cpp`. However, the source for the service program does not yet exist.

To work around this problem, follow the instructions in this sections, using the source code shown in the sample code figures.

Instructions

1. Create a source file `dummy.cpp`, using the source code shown in [Figure 21 on page 38](#).
2. Compile and bind `dummy.cpp` into a service program `PRINT`:

```
CRTCPPMOD MODULE(MYLIB/DUMMY) SRCSTMF(dummy.cpp)
CRTSRVPGM SRVPGM(MYLIB/PRINT) MODULE(MYLIB/DUMMY) EXPORT(*ALL)
```

3. Create the source file for program `MYPROG`, using the source code shown in [Figure 21 on page 38](#).
4. Create the program `MYPROG` from `myprog.cpp` and bind it to the service program `PRINT`. Enter the following commands:

```
CRTCPPMOD MODULE(MYLIB/MYPROG) SRCSTMF(myprog.cpp)
CRTPGM PGM(MYLIB/MYPROG) MODULE(MYLIB/MYPROG)
BNDSRVPGM(MYLIB/PRINT) OPTION(*UNRSLVREF)
```

The option `*UNRSLVREF` ensures that the program binds to the service program, although there is no matching export for MYPROG's import `void print(char *)`.

Code Samples

```
//dummy.cpp
#include <iostream.h>
void function(void) {
    cout << "I am a placeholder only" << endl;
    return;
}
```

Figure 21. Example of Source Code to Create a Dummy C++ Program

```
// myprog.cpp
#include <iostream.h>
#define size 80
void print(char *);
int main(void) {
    char text[size];
    cout << "Enter text" << endl;
    cin >> text;
    print(text);
    return 1;
}
```

Figure 22. Source Code for Example `myprog.cpp`

Running the Program

Before you can run program MYPROG successfully, you must

- Re-create service program PRINT from the real source code instead of from the placeholder code in `dummy.cpp`.
- Re-create program MYPROG, binding it to the new version of service program PRINT to resolve the reference to `print()`.

Note: MYPROG runs successfully only if PRINT actually exports a function that matches MYPROG's import request.

Updating a Service Program Export List

To make backward-compatible changes to an ILE C/C++ service program, you use the binder language. This language allows you to define a list of procedure names and data item names that can be exported. The Export Symbol (EXPORT) command in the binder language identifies the procedure and data item names that make up the signature for the service program module.

New procedure or data item names should be added to the end of the export list to ensure changes are compatible. A signature is generated by the order in which the modules are processed and the order in which the symbols are exported from the copied modules. A service program becomes difficult to update once the exports are used by other ILE C/C++ programs. If the service program is changed, the order or number of exports could change. If the signature changes all ILE C/C++ programs and service programs that use the changed service program have to be re-created.

The following example shows how to add a new procedure called `cost2()` to service program COST without having to re-create the existing program COSTDPT1 that requires an export from COST.

Program Description

The figure below shows the exports in the existing version of service program COST, and in the updated version.

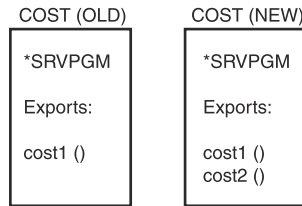


Figure 23. Exports from Service Program COST

The figure below shows the import requests in the existing program COSTDPT1, and in the new program COSTDPT2.

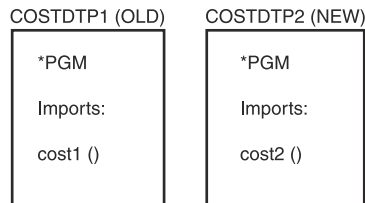


Figure 24. Import Requests in Programs COSTDPT1 and COSTDPT2

The binder language for the old version of service program COST is located in member BND of source file QSRVSRC, in library MYLIB:

```
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
ENDPGMEXP
```

The export signature is 94898385315FD06BB65E44D38A852904.

The updated binder language includes the new export procedure cost2(). It is located in member BNDUPD of source file QSRVSRC, in library MYLIB:

```
STRPGMEXP PGMLVL(*CURRENT)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
  EXPORT SYMBOL("cost2__Fi9_DecimalTXSP10SP2_9_DecimalTXSP3SP1_")
ENDPGMEXP
```

The new export signature is 61E595C21D3EC9DFD29749FB36B42D0.

In the binder language source that defines the old service program, the PGMLVL value is changed from *CURRENT to *PRV:

```
STRPGMEXP PGMLVL(*PRV)
  EXPORT SYMBOL("cost1__Fi9_DecimalTXSP10SP2_")
ENDPGMEXP
```

Its export signature is unchanged.

Note: If you want to ensure that existing programs can call the new version of the service program without being re-created, ensure that you:

1. Add the new exports to the end of the symbol list in the binder language
2. Explicitly specify a signature for the new version of the service program that is identical to the signature of the old version.

Creating the Source Files

The source code for service program COST, module COST2, and programs COSTDPT1 and COSTDPT2 is shown in the following figure:

```

    // cost1.cpp
    // contains the export function cost1() for the old service program
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost1 (
    int q,                // The quantity.
    _DecimalT<10,2> p )   // The price.
{
    _DecimalT<10,2> c;    // The cost.
    c = q*p;
    return c;
}
// cost2.cpp
// contains the export function cost2() for the new service program
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost2 (int quantity, _DecimalT<10,2> price,
    _DecimalT<3,1> discount )
{
    _DecimalT<10,2> c = __D(quantity*price*discount/100);
    return c;
}
// costdpt1.cpp
// This program prompts users (from dept1) to enter the
// quantity, and price for a product. It uses function
// cost1() to calculate the cost, and prints the result out.
#include <iostream.h>
#include <bcd.h>
_DecimalT<10,2> cost1(int, _DecimalT<10,2>);
int main(void)
{
    int            quantity;
    _DecimalT<10,2> cost;
    _DecimalT<10,2> price;
    cout << "Enter the quantity, please." << endl;
    cin >> quantity;
    cout << "Enter the price, please." << endl;
    cin >> price;
    cost = cost1(quantity, price);
    cout << "The cost is $" << cost << endl;
}

```

```

// costdpt2.cpp
// This program prompts users (from dept2) to enter the
// quantity, price, and discount rate for a product.
// It uses function cost2() to calculate the cost, and prints
// the result out.
#include <iostream.h>
#include <decimal.h>
_DecimalT<10,2> cost2(int, _DecimalT<10,2>, _DecimalT<3,1>);
int main(void)
{
    int            quantity;
    _DecimalT<10,2> price;
    _DecimalT<10,2> cost;
    _DecimalT<3,1> discount;
    cout << "Enter the quantity, please." << endl;
    cin >> quantity;
    cout << "Enter the price, please." << endl;
    cin >> price;
    cout << "Enter the discount, please.( %)" << endl;
    cin >> discount;
    cost = cost2(quantity, price, discount);
    cout << "The cost is be $" << cost << endl;
}

```

Figure 25. Source Code for Service Program COST

Compiling and Binding Programs and Service Programs

1. Create service program COST from source file cost1.cpp, using the binder source member BND, located in source file QSRVSRC, in library MYLIB:

```

CRTCPPMOD MODULE(MYLIB/COST1) SRCSTMF(cost1.cpp)
CRTSRVPGM SRVPGM(MYLIB/COST) MODULE(MYLIB/COST1) SRCFILE(MYLIB/QSRVSRC)
SRCMBR(BND) DETAIL(*EXTENDED)

```

2. Create program COSTDPT1 from source file `costdpt1.cpp` and service program COST, located in library MYLIB:

```
CRTCPPMOD MODULE(MYLIB/COSTDPT1) SRCSTMF(costdpt1.cpp)
CRTPGM PGM(MYLIB/COSTDPT1) MODULE(MYLIB/COSTDPT1) BNDSRVPGM(MYLIB/COST)
```

3. Update service program COST to include module COST2, using the updated binder language source BNDUPD, located in source file QSRVSRC in library MYLIB:

```
CRTCPPMOD MODULE(MYLIB/COST2) SRCSTMF(cost2.cpp)
CRTSRVPGM SRVPGM(MYLIB/COST) MODULE(MYLIB/COST1 MYLIB/COST2)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(BND) DETAIL(*EXTENDED)
```

It is necessary to re-create the service program COST, using the two modules COST1 and COST2 and the updated version of the binder language BNDUPD, so that it supports the new `cost2()` function. Program COSTDPT1, which used COST before it was re-created, remains unchanged.

In order to update service program COST, it is necessary to re-create it from the two modules COST1 and COST2, using the updated version of the binder language BNDUPD. The **EXTENDED* option in the *DETAIL* parameter creates an extended output listing, so that you can look at the current and previous signature of COST.

4. Create program COSTDPT2 from source file `costdpt2.cpp`:

```
CRTCPPMOD MODULE(MYLIB/COSTDPT2) SRCSTMF(costdpt2.cpp)
CRTPGM PGM(MYLIB/COSTDPT2) MODULE(MYLIB/COSTDPT2) BNDSRVPGM(MYLIB/COST)
```

Running the Program

Run program COSTDPT1 from an IBM i command line using the CL command `CALL COSTDPT1`.

Run program COSTDPT2 from an IBM i command line using the CL command `CALL COSTDPT2`.

Running a Program

There are several ways to run a program in the ILE environment. You can use:

- A control language (CL) command:
 - Call (CALL) command
 - Transfer Control (TFRCTL) command
 - Start Programming Development Manager (STRPDM) command
 - user-defined CL command
- An ILE C/C++ program as a Command Processing Program (CPP)
- A high-level language CALL statement

Note: [“Using ILE C/C++ Call Conventions” on page 283](#) contains information on interlanguage calls.

- The EVOKE statement in an ICF file
- The REXX interpreter
- The QCAPEXC program
- The ILE Programmer Menu

This topic describes:

- [The ILE C/C++ runtime model](#)
- [Activations and activation groups](#)
- [Runtime functions and activation groups](#)
- [How to call programs](#)
- [Normal and abnormal end-of-program](#)

- [How to manage activation groups](#)
- [How to manage runtime storage](#)

The ILE C/C++ Runtime Model

The ILE C/C++ runtime model guarantees ISO C/C++ standard semantics when either of the following are true:

- All programs in an application are created with the Create Bound Program commands (CRTBNDC and CRTBNDCPP).
- The following options are used with the Create Program (CRTPGM) command:

<i>Table 7. CRTPGM Command Options</i>	
Option	Description
ACTGRP(*NEW)	A new activation group is created on every call of the created *PGM, and the activation group is destroyed when the program ends.
OPTION(*NODUPPROC)	No duplicate procedure definitions in the same bound program are allowed.
OPTION(*NODUPVAR)	No duplicate variable definitions in the same bound program are allowed.

- The following options are used with the Create Service Program (CRTSRVPGM) command:

<i>Table 8. CRTSRVPGM Command Options</i>	
Option	Description
ACTGRP(*CALLER)	When this service program is called, it is activated into the activation group of the calling program.
OPTION(*NODUPPROC)	No duplicate procedure definitions in the same bound program are allowed.
OPTION(*NODUPVAR)	No duplicate variable definitions in the same bound program are allowed.

Note: When a CRTPGM parameter does not appear in the Create Bound Program command invocation, the default is the CRTPGM parameter. For example, the parameter ACTGRP(*NEW) is the default for the CRTPGM command, and is used for the Create Bound Program command. You can change the CRTPGM parameter defaults using the Change Command Defaults (CHGCMDDF) command.

Activations and Activation Groups

After successfully creating an ILE C/C++ program, you can run your code. Activation is the process of getting an ILE C/C++ program or service program ready to run. When an ILE C/C++ program is called, the system performs activation. Because ILE C/C++ service programs are not called, they are activated during the call to an ILE C/C++ program that directly or indirectly requires their services.

Activations and activation groups provide the following functions and benefits:

- They help ensure that ILE C/C++ programs running in the same job run independently without intruding on each other by scoping resources to the activation group. Examples of programs running in the same job are commitment control, overrides, and shared files.
- They scope resources to the ILE C/C++ program.
- They uniquely allocate static data needed by the ILE C/C++ program or service program.
- They change symbolic links to ILE C/C++ service programs to physical addresses.

When activation allocates the storage necessary for the static variables that are used by an ILE C/C++ program or service program, the space is allocated from an activation group. At the time the ILE C/C++ program or service program is created, you specify the activation group that should be used at runtime.

Once an ILE C/C++ program is activated, it remains activated until the activation group is deleted. Even though they are activated, programs do not appear in the call stack unless they are running.

When an IBM i job is started, the system creates two activation groups for OPM programs. One activation group is reserved for IBM i system code and the other is used for all other OPM programs. You cannot delete the OPM default activation groups. The system deletes them when your job ends.

Note: OPM programs are not threadsafe. OPM programs should be migrated to ILE and made threadsafe before they are called in a multithreaded application. When it is necessary to call an OPM program in a multithreaded application, start another process to run the OPM program.

An activation group can continue to exist even when the `main()` function of an ILE C/C++ program is not on the call stack. This occurs when the ILE C/C++ program was created with a named activation group (specifying a name on the `ACTGRP` option of the `CRTPGM` command), and the `main()` function issues a return. This can also occur when the ILE C/C++ program performs a `longjmp()` across a control boundary by using a jump buffer that is set in an ILE C/C++ procedure. This procedure is higher in the call stack and before the nearest control boundary.

Runtime Library Functions and Activation Groups

The ILE C/C++ runtime library functions are bound to the application in the activation group in which the application is called. This means that:

- All program activations in the same activation group share one instance of the ILE C/C++ runtime library.
- The state of the ILE C/C++ runtime environment propagates across program call boundaries.

In other words, if one program in an activation group changes the state of the ILE C/C++ runtime, then all other programs in that activation group are affected. For example, other programs in the same activation group are affected by the locale setting of an application or the shift-in/shift-out states of the multibyte functions.

If the `ACTGRP` parameter of the `CRTPGM` command is specified to a value other than `*NEW`, the application's runtime behavior might not follow ISO C or ISO C++ standards. Non-ISO behavior may occur during:

- Program ending (`exit()`, `abort()`, `atexit()`)
- Signal handling (`signal()`, `raise()`)
- Multibyte string handling (`mblen()`)
- Any locale-dependent library functions (`isalpha()`, `qsort()`)

In the default activation groups, I/O files are not automatically closed. The I/O buffers are not flushed.

If `ACTGRP` is set to `*CALLER`, multiple calls of an ILE C/C++ program share one instance of the ILE C/C++ runtime library state in the same activation group. Through this option, ILE C/C++ programs can run within the OPM default activation groups. Certain restrictions exist for ILE C/C++ programs that run in the OPM default activation groups. For example, you are not allowed to register `atexit()` functions within the OPM default activation groups.

If the activation group is named:

- All calls to programs in this activation group within the same job share the same instance of the ILE C/C++ runtime library state.
- No constructors, destructors, or static initialization in the program are executed.

Note: Constructors, destructors, and static initializations are executed only when the activation group is created.

It is possible to create an ISO-compliant application whose programs are created with options other than ACTGRP(*NEW).

Note: It is the responsibility of the application designer to ensure that the sharing of resources, and runtime states across all programs in the activation group do not result in non-ISO behavior.

Calling Programs

When you call a program, the IBM i system locates the corresponding executable code and performs the instructions found in the program.

Note: Only programs can run independently. Service programs or other bound procedures must be called from a program that requires their services.

There are several ways to call a program:

- [Using the Call \(CALL\) command](#)
- [Using the Transfer Control \(TFRCTL\) command](#)
- [Creating a CL command to call a program](#)

Using the Call (CALL) Command

You can use the Call (CALL) command to run a program interactively, or as part of a batch job.

The syntax for this command is:

➤ CALL PGM — (*library-name/program-name*) ➤

For example, the command

```
CALL PGM(MYLIB/MYPROG)
```

invokes the program MYPROG located in the library MYLIB.

If the program object specified by *program-name* exists in a library that is contained in your library list, you can omit the library name in the command, and the syntax is:

➤ CALL — *program-name* ➤

For example, if MYLIB appears in your library list, you can simply enter:

```
CALL MYPROG
```

Note: If you need prompting for the command parameters, type CALL and press F4 (Prompt). If you need help on how to use the command, type CALL and press F1(Help).

Passing Parameters to the Called Program

When you request prompting with the Call command, a display appears that allows you to supply the parameters to the program you are calling.

You can also type the parameters directly onto the command line, following the Call command.

If the program requires only one parameter, enter:

```
CALL MYPROG 'parameter 1'
```

If the program requires more than one parameter, you must use the PARM keyword. For example:

```
CALL MYPROG PARM ('parameter 1' parameter 2')
```

Example1:

The following example shows an ILE C/C++ program T1520REP that requires parameters at runtime.

1. Suppose the source code is stored as a member T1520REP in file QACSRC of library QCPPLE. To create the program T1520REP, enter:

```
CRTBNDC PGM(MYLIB/T1520REP) SRCFILE(QCPPLE/QACSRC)
```

The source code is shown in [Figure 26 on page 45](#).

2. To run the program T1520REP, enter:

```
CALL PGM(MYLIB/T1520REP) PARM('Hello, World')
```

The output is:

```
Hello, World
Press ENTER to end terminal session.
```

The source file for program T1520REP is shown in the following figure:

```
/* Print out the command line arguments.          */
#include <stdio.h>
void main ( int argc, char *argv[] )
{
    int i;
    for ( i = 1; i < argc; ++i )
        printf( "%s\n", argv[i] );
}
```

Figure 26. T1520REP – ILE C Source to Pass Parameters to an ILE C Program

Example 2

The following example demonstrates how to pass the value 'Hello, World' to program XRUN1 which expects parameters at runtime.

Follow the steps below to create and run program XRUN1:

1. Compile the source shown above with default compiler options. From the command line, enter:

```
CRTBNDCPP PGM(MYLIB/XRUN1) SRCSTMF('xrun1.cpp')
```

The resulting program object is created in the default library (in this example, MYLIB).

2. To run the program from a command line, enter:

```
CALL PGM(MYLIB/XRUN1) PARM('Hello, World')
```

The output of program XRUN1 is:

```
Hello, World
Press ENTER to end terminal session.
```

The source file xrun1.cpp for program XRUN1 is shown in the following figure:

```
// xrun1.cpp
// Prints out command line arguments.

#include <iostream.h>
int main ( int argc, char *argv[] )
{
    int i;
    for ( i = 1; i < argc; ++i )
        cout << argv[i] << endl;
}
```

Figure 27. Source File for a Program that Passes the Value 'Hello, World' to Another Program

Call (CALL) Command Parameter Conversions

When you call a program from a CL command line, the parameters you pass on the Call command are changed, depending on how you state the parameters. [Table 9 on page 46](#) shows how parameters are converted.

Conversion Rules	Examples	Conversion Results
String literals are passed with a null terminating character.	CALL PGM(T1520REP) PARM(abc)	ABC\0 (converted to uppercase; passed as a string)
Numeric constants are passed as packed decimal digits.	CALL PGM(T1520REP) PARM('123.4')	123.4 (passed as a packed decimal (15,5))
Characters that are not enclosed in single quotation marks are: <ul style="list-style-type: none">• Folded to uppercase• Passed with a null character	CALL PGM(T1520REP) PARM(123.4)	123.4\0 (passed as a string)
Characters that are enclosed in single quotation marks are not changed. Mixed case strings are supported, and are passed with a null terminating character.	CALL PGM(T1520REP) PARM('abc') and CALL PGM(T1520REP) PARM('abc')	abc\0 (passed as a string) and abC\0 (passed as a string)

The REXX interpreter treats all REXX variables as strings (without a null terminator). REXX passes parameters to IBM i which then calls the ILE C/C++ program. Conversion to a packed decimal data type still occurs, and strings are null terminated.

Note: These changes only apply to calling a program from a command line, not to interlanguage calls. See [“Using ILE C/C++ Call Conventions” on page 283](#) for information on ILE C/C++ calling conventions.

Using the Process Commands (QCAPCMD) API

You can use the Process Commands (QCAPCMD) API to:

- Add the null character to arguments that are passed to an ILE C/C++ program.
- Check the syntax of a command string prior to running it, prompt the command and receive the changed command string, and run a command from an HLL (high-level language).

The QCAPCMD API is used to perform command analyzer processing on command strings. You can check or run CL commands from HLLs as well as check syntax for specific source definition types.

Using the Transfer Control (TFRCTL) Command

You can run an application from within a CL program that transfers control to your program using the Transfer Control (TFRCTL) command. This command:

1. Transfers control to the program specified on the command.
2. Removes the transferring CL program from the call stack.

In the following example, the TFRCTL command in a CL program RUNCP calls a C++ program XRUN2, which is specified on the TFRCTL command. RUNCP transfers control to XRUN2. The transferring program RUNCP is removed from the call stack.

[Figure 28 on page 47](#) illustrates the call to the CL program RUNCP, and the transfer of control to the C++ program XRUN2.

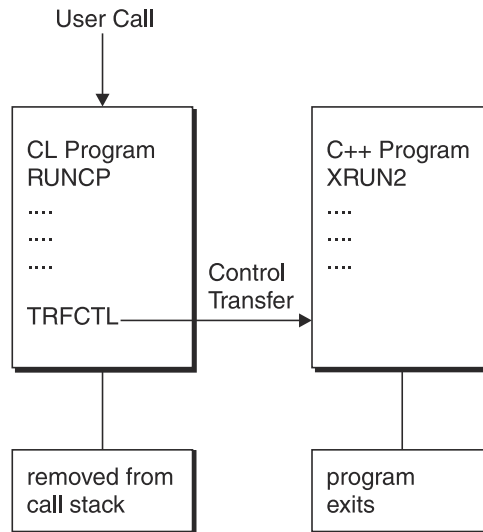


Figure 28. Calling Program XRUN2 Using the TRFCTL Command

Example: Creating and Running a Program that Uses the TRFCTL Command

To create and run programs RUNCP and XRUN2, follow the steps below:

1. Create the source file QCLSRC and enter the source code shown in [Figure 29 on page 47](#).
2. Create the CL program RUNCP. From the command line, enter:

```
CRTCLPGM PGM(MYLIB/RUNCP) SRCFILE(MYLIB/QCLSRC)
```

3. To create program XRUN2 in library MYLIB from source file xrun2.cpp (shown in [Figure 30 on page 48](#)), enter:

```
CRTBNDCPP PGM(MYLIB/XRUN2) SRCSTMF(xrun2.cpp)
```

4. Run program RUNCP from a command line, passing it the string "nails", with the command:

```
CALL PGM(MYLIB/RUNCP) PARM('nails')
```

The output from program XRUN2 is:

```
string = nails
Press ENTER to end terminal session.
```

Code Samples

```

/* Source for CL Program RUNCP                                     */
PGM          PARM(&STRING)                                         */
DCL          VAR(&STRING)    TYPE(*CHAR)  LEN(20)                 */
DCL          VAR(&NULL)     TYPE(*CHAR)  LEN(1)  VALUE(X'00')     */

/* ADD NULL TERMINATOR FOR THE ILE C++ PROGRAM                   */
CHGVAR      VAR(&STRING)  VALUE(&STRING *TCAT &NULL)             */
TRFCTL      PGM(MYLIB/XRUN2) PARM(&STRING)                       */

/* THE DSPJOBLOG COMMAND IS NOT CARRIED OUT SINCE                */
/* WHEN PROGRAM XRRUN2 RETURNS, IT DOES NOT RETURN TO THIS     */
/* CL PROGRAM.                                                    */
DSPJOBLOG                                       */
ENDPGM
  
```

Figure 29. Example of Source Code that Transfers Control to Another Program

Note: In the example [“Example: Creating and Running a Program that Uses the TRFCTL Command” on page 47](#), program RUNCP uses the TRFCTL command to pass control to the ILE C++ program XRUN2, which does not return control to RUNCP.

```

// xrun2.cpp
// Source for Program XRUN2
// Receives and prints a null-terminated character string

#include <iostream.h>

int main(int argc, char *argv[])
{
    int    i;
    char * string;
    string = argv[1];
    cout << "string = " <<  string << endl;
}

```

Figure 30. Example of Source Code that Receives and Prints a Null-Terminated Character String

Note: In the example “[Example: Creating and Running a Program that Uses the TFRCTL Command](#)” on page 47, program XRUN2 receives a null-terminated character string from the CL program and prints the string.

Creating a CL Command to Run a Program

You can also run a program from your own CL command. To create a command:

1. Enter a set of command statements into a source file.
2. Process the source file and create a command object (type *CMD) using the Create Command (CRTCMD) command.

The CRTCMD command definition includes the command name, parameter descriptions, and validity-checking information, and identifies the program that performs the function requested by the command.

3. Enter the command interactively, or in a batch job.

The program called by your command is run.

The following [example](#) illustrates how to run a program from a user-created command:

Program Description

A newly created command COST prompts for and accepts user input values. It then calls a C++ program CALCOST and passes it the input values. CALCOST accepts the input values from the command COST, performs calculations on these values, and prints results. [Figure 31 on page 48](#) illustrates this example.

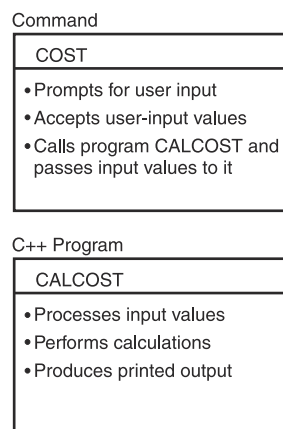


Figure 31. Calling Program CALCOST from a User-Defined Command COST

Instructions

To create and run the example, follow the steps below:

1. Enter the source code for the command prompt COST (shown in [Figure 32 on page 49](#)) into a source file QCMDSRC in library MYLIB, and save it as member COST:
2. To Create the command prompt COST. From the command line, enter:

```
CRTCMD CMD(MYLIB/COST) PGM(MYLIB/CALCOST) SRCFILE(MYLIB/QCMDSRC)
```

3. To create program CALCOST from the source file calcost.cpp (shown in [Figure 32 on page 49](#)), enter:

```
CRTBNDCPP PGM(MYLIB/CALCOST)
```

4. To run program CALCOST:

- a. Enter COST and press F4 (Prompt). The prompts ITEM, PRICE, and QUANTITY appear in order.
- b. When prompted, enter the data shown below:

```
Hammers
1.98
5000
```

The output of program CALCOST is:

```
It costs $11385.00 to buy 5000 HAMMERS
Press ENTER to end terminal session.
>
```

Code Samples

```
/* Source for Command Prompt COST */
CMD      PROMPT('CALCULATE TOTAL COST')
PARM     KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
         MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
PARM     KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
         RANGE(0.01 99999999.99) MIN(1) +
         ALWUNPRT(*YES) PROMPT('Unit price' 2)
PARM     KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
         9999) MIN(1) ALWUNPRT(*YES) +
         PROMPT('Number of items' 3)
```

Figure 32. Source Code for Command Prompt that Runs the CALCOST Program

```
// calcost.cpp
// Source for Program CALCOST

#include <iostream.h>
#include <string.h>
#include <bcd.h>

int main(int argc, char *argv[])
{
    char          *item_name;
    _DecimalT<10,2> *price;
    short int     *quantity;
    const _DecimalT<2,2> taxrate=_D("0.15");
    _DecimalT<17,2> cost;
    item_name = argv[1];
    price     = (_DecimalT<10,2> *) argv[2];
    quantity  = (short *) argv[3];
    cost = (*quantity)*(*price)*(_D(1.00+taxrate));
    cout << "\nIt costs $" << cost << " to buy "
         << *quantity << " " << item_name << endl;
}
```

Figure 33. Source Code for Program CALCOST

Note: This program receives the incoming arguments from the command COST, calculates a cost, and prints values. All incoming arguments are pointers.

Normal and Abnormal End-of-Program

When a program ends normally, the system returns control to the caller. The caller might be a workstation user or another program.

If a program ends abnormally during runtime, and the program had been running in a different activation group from its caller, the escape message CEE9901 is issued and control is returned to the caller:

```
Application error <msgid> unmonitored by <pgm> at  
statement <stmtid>, instruction <instruction>
```

A CL program can monitor for this exception by using the Monitor Message (MONMSG) command.

If the program and its caller are running in the same activation group and the program ends abnormally, the message that is issued depends on how the program ends. If it ends with a function check, CPF9999 is issued.

Note: For more information about escape messages, see Message Handling Terms and Concepts within the Message Handling APIs topic in the *APIs* section in the *Programming* category at the IBM i Information Center web site: <http://www.ibm.com/systems/i/infocenter>.

Managing Activation Groups

Activation groups make it possible for multiple ILE programs to run in the same job independently, without intruding on each other.

An activation group is a substructure of a job. It consists of system resources such as storage, commitment definitions, and open files. These resources are allocated to run one or more ILE or OPM programs. For example, the storage space for the static variables of a program is allocated from an activation group.

Once a program (type *PGM) is called, it remains activated until the activation group it runs in is deleted. Because service programs are not called directly, they are activated during the call to the program that requires their services.

Specifying an Activation Group

When an IBM i job is started, the system automatically creates two activation groups to be used by OPM programs. One activation group is reserved for IBM i system code. The other activation group is used for all other OPM programs. The symbol used to represent this activation group is *DFTACTGRP. You cannot delete the OPM default activation groups. The system deletes them when your job ends.

Note: OPM programs always run in the default activation group; you cannot change their activation group specification.

For ILE programs you specify the activation group that should be used at runtime through the ACTGRP parameter of the Create Program or Create Service Program commands. You can choose between:

- Running your program in a named activation group.
- Accepting the default activation group:
 - *NEW for programs
 - *CALLER for service programs
- Activating a program into the activation group of a calling program.

Running a Program in a Named Activation Group

To manage a collection of ILE programs and service programs as one application, you create a named activation group for them by specifying a user-defined name on the ACTGRP parameter.

The system creates the named activation group as soon as the first program that has specified this activation group is called. This group is then used by all programs and service programs that have specified its name.

A named activation group ends when it is deleted through the Reclaim Activation Group (RCLACTGRP) command. This command can only be used when the activation group is no longer in use. It also ends when you call the `exit()` function in your code.

When a named activation group ends, all resources associated with the programs and service programs of the group are returned to the system.

Note: Using named activation groups may result in non-ISO compliant runtime behavior. If a program created using named activation groups remains activated by a return statement, you encounter the following problems:

- Static variables are not re-initialized.
- Static constructors are not called again.
- Static destructors are not called on return.
- Other programs activated in the same activation group may terminate your program, although they seem to be independent.
- Your program is not portable, if you count on the behavior of the named activation group.

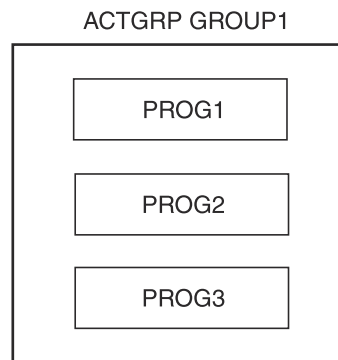


Figure 34. Running Programs in a Named Activation Group

In the following example, programs PROG1, PROG2, and PROG3 are part of the same application and run in the same activation group, GROUP1. Figure 34 on page 51 illustrates this scenario:

To create these programs in the same activation group, you specify GROUP1 on the *ACTGRP* parameter when you create each program:

```
CRTCPPMOD MODULE(PROG1) SRCSTMF(prog1.cpp)
CRTPGM PGM(PROG1) MODULE(PROG1) ACTGRP(GROUP1)
CRTCPPMOD MODULE(PROG2) SRCSTMF(prog2.cpp)
CRTPGM PGM(PROG2) MODULE(PROG2) ACTGRP(GROUP1)
CRTCPPMOD MODULE(PROG3) SRCSTMF(prog3.cpp)
CRTPGM PGM(PROG3) MODULE(PROG3) ACTGRP(GROUP1)
```

Running a Program in Activation Group **NEW*

To create a new activation group whenever your program is called, specify **NEW* on the *ACTGRP* parameter. In this case, the system creates a name for the activation group that is unique within your job.

**NEW* is the default value of the *ACTGRP* parameter on the *CRTPGM* command. An activation group created with **NEW* always ends when the last program associated with it ends.

Note: **NEW* is not valid for a service program, which can only run in the activation group of its caller, or in a named activation group.

If you create a program with *ACTGRP(*NEW)*, more than one user can call the program at the same time without using the same activation group. Each call uses a new copy of the program. Each new copy has its own data and opens its files.

In the following example, programs PROG4, PROG5, and PROG6 run in separate unnamed activation groups.

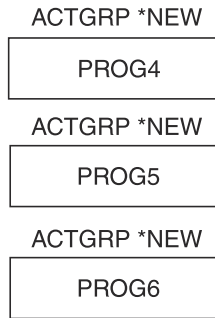


Figure 35. Running Programs in Unnamed Activation Groups

By default, each program is created into a different activation group, identified by the *ACTGRP* parameter (**NEW*).

```
CRTCPPMOD MODULE(PROG4) SRCSTMF(prog4.cpp)
CRTPGM PGM(PROG4) MODULE(PROG4) ACTGRP(*NEW)
CRTCPPMOD MODULE(PROG5) SRCSTMF(prog5.cpp)
CRTPGM PGM(PROG5) MODULE(PROG5) ACTGRP(*NEW)
CRTCPPMOD MODULE(PROG6) SRCSTMF(prog6.cpp)
CRTPGM PGM(PROG6) MODULE(PROG6) ACTGRP(*NEW)
```

Because **NEW* is the default, you obtain the same result with the following invocations:

```
CRTBNDCCP PGM(PROG4) SRCSTMF(prog4.cpp)
CRTBNDCCP PGM(PROG5) SRCSTMF(prog5.cpp)
CRTBNDCCP PGM(PROG6) SRCSTMF(prog6.cpp)
```

Note: If you invoke three modules in one command a single program object PROG is created in activation group **NEW*:

```
CRTCPPMOD MODULE(PROG7) SRCSTMF(prog7.cpp)
CRTCPPMOD MODULE(PROG8) SRCSTMF(prog8.cpp)
CRTCPPMOD MODULE(PROG9) SRCSTMF(prog9.cpp)
CRTPGM PGM(PROG) MODULE(PROG7 PROG8 PROG9)
```

Non-Standard Behavior with Named Activation Groups

If the *ACTGRP* parameter of the *CRTPGM* command is specified as a value other than **NEW*, the application's runtime behavior may not follow ISO semantics. Runtime and class libraries assume that programs are built with *ACTGRP(*NEW)*

Non-ISO behavior may occur during:

- Program termination - `exit()`, `abort()`, `atexit()`
- Signal handling - `signal()`, `raise()`
- Multibyte string handling - `mblen()`
- Any locale-dependent library functions - `isalpha()`, `qsort()`

In the default activation groups, I/O files are not automatically closed. The I/O buffers are not flushed unless explicitly requested.

Running a Program in Activation Group (**CALLER*)

You can specify that an ILE program or an ILE service program be activated within the activation group of a calling program, by setting *ACTGRP* to **CALLER*. With this attribute, a new activation group is never created when the program or service program is activated. Through this option, ILE C/C++ programs can run within the OPM default activation groups when the caller is an OPM program.

Certain restrictions exist for ILE C/C++ programs running in the OPM default activation groups. For example, you are not allowed to register `atexit()` functions within the OPM default activation groups.

If the activation group is named, all calls to programs in this activation group within the same job share the same instance of the ILE C/C++ runtime library state.

It is possible to create an ISO-compliant application whose programs are created with options other than *ACTGRP(*NEW)*. While non-ISO behavior may be desirable in certain cases, it is the responsibility of the application designer to ensure that the sharing of resources and runtime states across all programs in the activation group does not result in incorrect behavior.

In the following example, a service program SRV1 is activated into the respective activation groups of programs PROG7 and PROG8. PROG7 runs in a named activation group GROUP2, while PROG8 runs in an unnamed activation group **NEW*.

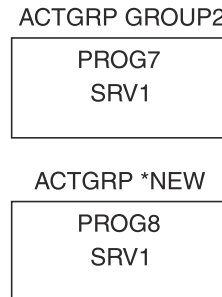


Figure 36. Running a Service Program in the Activation Groups of Calling Programs

By default, the service program SRV1 is created into the activation group of each calling program.

```
CRTCPPMOD MODULE(SRV1) SRCSTMF(srv1.cpp)
CRTSRVPGM SRVPGM(SRV1) MODULE(SRV1)
```

Presence of a Program on the Call Stack

Even though it is activated, a program does not appear on the call stack unless it is running. But an activation group can continue to exist even when the `main()` function of the program is not on the call stack.

This occurs when the program was created with a named activation group, and the `main()` function issues a return. It can also occur when the program performs a `longjmp()` across a control boundary by using a jump buffer that is set in an ILE C or C++ procedure. (This procedure is higher in the call stack and before the nearest control boundary.)

Deleting an Activation Group

When an activation group is deleted, its resources are reclaimed. The resources include static storage and open files. A **NEW* activation group is deleted when the program it is associated with returns to its caller.

Named activation groups are persistent. You must delete them explicitly. Otherwise they end only when the job ends. The storage associated with programs running in named activation groups is not released until these activation groups are deleted.

The OPM default activation group is also a persistent activation group. The storage associated with ILE programs running in the default activation group is released either when you sign off (for an interactive job) or when the job ends (for a batch job).

Reclaiming System Resources

You may encounter situations where system storage is exhausted, for example:

- If many ILE programs are activated (that is, called at least once).
- If ILE programs that use large amounts of static storage run in the OPM default activation group (storage is not reclaimed until the job ends).

- If many service programs are called into named activation groups (resources are only reclaimed when the job ends).

In such situations, you may want to reclaim system resources that are no longer needed for a program, but are still tied up because an activation group has not been deleted. You have the following options:

- Delete a named activation group that is not in use through the Reclaim Activation Group (RCLACTGRP) command .

The command provides options to either delete all eligible activation groups or to delete an activation group by name.

- Free resources for programs that are no longer active through the Reclaim Resources (RCLRSC) command.

Using the Reclaim Resources (RCLRSC) Command

The RCLRSC command works differently depending on how the program was created:

- For OPM programs, the RCLRSC command closes open files and frees static storage.
- For ILE programs that were activated into the OPM default activation group (because they were created with *CALLER), The RCLRSC command closes files and reinitializes storage. However, the storage is not released.
- For ILE programs associated with a named activation group, the RCLRSC command has no effect. You must use the RCLACTGRP command to free resources in a named activation group.

Managing Runtime Storage

ILE allows you to manage runtime storage directly from your program, by managing heaps. A heap is an area of storage used for allocations of dynamic storage. The amount of dynamic storage required by an application depends on the data being processed by the programs and procedures that use the heap. You manage heaps by using ILE bindable APIs.

You are not required to manage runtime storage explicitly. However, you may wish to do so if you want to make use of dynamically allocated storage. For example, if you do not know exactly how big an array should be, you could acquire the actual storage for the array at runtime, once your program determines how big the array should be.

There are two types of heaps available on the system:

- default heap
- user-created heap

You can use one or more user-created heaps to isolate the dynamic storage required by some programs and procedures within an activation group.

The rest of this section explains how to use a default heap to manage runtime storage in a program.

Managing the Default Heap

The first request for dynamic storage within an activation group results in the creation of a *default heap* from which the storage allocation takes place. Additional requests for dynamic storage are met by further allocations from the default heap. If there is insufficient storage in the heap to satisfy the current request for dynamic storage, the heap is extended, and the additional storage is allocated.

Allocated dynamic storage remains allocated until it is explicitly freed, or until the heap is discarded. The default heap is discarded only when the owning activation group ends.

Programs in the same activation group all use the same default heap. If one program accesses storage beyond what has been allocated, it can cause problems for another program.

For example, assume that two programs, PGM1 and PGM2 are running in the same activation group. 10 bytes are allocated for PGM1, but 11 bytes are changed by it. If the extra byte was in fact allocated for PGM2 problems may arise for PGM2.

Using Bindable APIs to Manage the Default Heap

You can use the following ILE bindable APIs on the default heap:

Free Storage (CEEFRST)

Frees one previous allocation of heap storage

Get Heap Storage (CEEGTST)

Allocates storage within a heap

Reallocate Storage (CEEZST)

Changes the size of previously allocated storage

Dynamically Allocating Storage at Runtime

C++

In an ILE C++ program, you manage dynamic storage belonging to the default heap using the operators `new` and `delete` to create and delete dynamic objects. Dynamic objects are never created and deleted automatically. Their creation can fail if there is not enough free heap space available, and your programs must provide for this possibility.

The following figures illustrate how to use the `new` and `delete` operators for dynamic storage allocation:

```
TClass *p;                // Define pointer
p = new TClass;           // Construct object
if (!p) {
    Error("Unable to construct object");
    exit(1);
}
...
delete p;                 // Delete object
```

Figure 37. Example of Dynamic Allocation and De-Allocation of Storage for a Class Object

```
TClass *array;            // Define pointer
array = new TClass[100];  // Construct array of 100 objects
if (!array) {
    Error("Unable to construct array");
    exit(1);
}
...
delete[] array;           // Delete array
```

Note: In this example, you use `delete[]` to delete the array. Without the brackets, `delete` deletes the entire array, but calls the destructor only for the first element in the array. If you have an array of values that do not have destructors, you can use `delete` or `delete[]`.

Figure 38. Example of Dynamic Allocation and De-Allocation of Storage for an Array of Objects

Overriding Replacement Functions

C++

The C++ standard allows an application to redefine a number of *replacement functions*. The program's definitions are used instead of the default versions supplied by the library. Such replacement occurs prior to program startup.

A C++ program may provide the definition for any of the eight dynamic memory allocation functions. These include:

- `void *operator new (size_t) throw(std::bad_alloc);`
- `void *operator new (size_t, const std::nothrow_t&) throw();`
- `void *operator new[] (size_t) throw(std::bad_alloc);`
- `void *operator new[] (size_t, const std::nothrow_t&) throw();`
- `void operator delete (void*) throw();`
- `void operator delete (void*, const std::nothrow_t&) throw();`

- `void operator delete [] (void*) throw();`
- `void operator delete [] (void*, const std::nothrow_t&) throw();`

Limitations

When overriding replacement functions, consider the following limitations:

- A program that contains a main function has to be compiled with C++ compiler.
- When the `main()` entry point is not a C++ module, the calls to global `new` or `delete` operators work only if they are in the same compilation unit (where the definition of the corresponding replacement functions are visible).
- Infinite recursion can occur when you use standard library objects in the implementation of replacement functions because the library makes extensive use of calls to the allocation operators.

Note: Avoid using `iostreams` for logging.

Overloading the new or delete Operator

C++ The ISO C++ Standard categorizes operator `new` and operator `delete` as replacement functions, which means that they can be redefined in a C++ program. However, the standard allows only one definition of an operator to be in effect during program execution.

Note: Visibility issues can arise if a program does both of the following:

- Overloads operator `new` or operator `delete`
- Uses multiple C++ translation units

For detailed information about visibility issues, see the *ILE C/C++ Language Reference*.

Example:

Suppose an application uses three C++ source files (`one.cpp`, `two.cpp`, and `three.cpp`):

- `One.cpp` contains the main function.
- `Two.cpp` contains a redefinition of operator `new` or operator `delete`.
- `Three.cpp` calls operator `new`.

After you compile the application, the redefined operator `new` (or operator `delete`) is visible to, and used for, all translation units.

Given the same three-translation unit scenario, suppose that `one.cpp` is compiled with the C compiler. The redefined operator is visible in translation unit `two.cpp` but not in `three.cpp`. Any calls to operator `new` (or operator `delete`) outside of translation unit `two.cpp` uses the standard version, not the user-defined version, of the operator.

Note: Because user-defined and standard operators have different signatures, no binder error or compiler warning is generated.

Improving Runtime Performance

When you examine a program to improve performance, look at those aspects which have a significant impact every time a program is run.

Often runtime performance can be improved through minor changes to your source programs. The amount of improvement each change provides depends on

- How your program is organized
- The functions and language constructs your program uses

Some changes may provide substantial performance improvement to your program, while others may offer almost no improvement.

Note: Some tips may contradict each other because they trade one advantage for another. For example, one tip is to reduce the size of the call stack by using static and global variables, while another tip is to improve execution startup performance by reducing the use of static and global variables.

Before trying to improve runtime performance, compile and benchmark your programs using full optimization. Use performance analysis tools to find out where your performance problems are, and then try and apply different appropriate tips to try and achieve the best performance for your program.

This topic discusses how you might try to improve performance with respect to:

- [Data types](#)
- [Classes](#)
- [Performance measurement](#)
- [Exception handling](#)
- [Function call performance](#)
- [Input and output considerations](#)
- [Pointers](#)
- [Shallow copy and deep copy](#)
- [Space considerations](#)
- [Activation groups](#)
- [Compiler options](#)
- [Runtime limits](#)

Choosing Data Types to Improve Performance

There are several ways to improve performance through data types. Replacing bit fields with other data types and minimizing the use of static and global variables are some of the ways.

Avoiding Use of the Volatile Qualifier

Only use the `volatile` qualifier when necessary. `volatile` specifies that a variable can be changed at any time, possibly by an external program, and therefore it is not a candidate for optimization.

Replacing Bit Fields with Other Data Types

Avoid using bit fields because it takes more time to access bit-fields than other data types such as `short` and `int`. Whenever possible, replace bit fields with other data types. If a bit field takes 16 bits and aligns on 2-byte boundary, you can replace it with the `short` data type.

Note: You can still obtain a runtime improvement if the bit-field is smaller than the integral type. The extra time required for bit-field manipulation code offsets the performance gain due to space saved in data.

Minimizing the Use of Static and Global Variables

Minimize the use of static and global variables, if possible. These are initialized whether or not you explicitly initialize them. By not using static and global variables, the performance improvement is obtained at activation group startup.

Using the Register Storage Class

Use the Register Storage class for a variable that is frequently used. Do not overuse the Register Storage class, so that the optimizer can place the most frequently used variables into the available hardware registers.

If you use the Register Storage class, you cannot rely on the value displayed from within the debugger because you may be referencing an older value that is still in storage.

Creating Classes to Improve Performance

C++

When you use class libraries to create classes, use a high level of abstraction. After you establish the type of access to your class, you can create more specific implementations. This can result in improved performance with minimal code changes.

When you define structures or data members within a class, define the largest data types first to align them on the largest natural boundary. Define pointers first to reduce the padding necessary to align them on quadword (16-byte) boundaries. Follow them, in order, with the double-word, and half-word items to avoid padding or improve load/store time.

Enabling Performance Measurement

You can use a native compiler option to include performance hooks in your generated code.

Using a Compiler Option to Enable Performance Measurement

The performance-measurement compiler option `ENBPFCOL()` allows you to specify whether or not the compiler should generate code (sometimes called performance hooks) into your compiled program or module. The performance hooks enable the Performance Explorer to analyze your programs. The default for this option specifies that program entry procedure level performance-measurement code is generated for your compiled module or program.

Compiling performance collection code into the module or program allows performance data to be gathered and analyzed. The insertion of the additional collection code results in slightly larger module or program objects and may affect performance to a small degree.

Types of performance data collected include:

- Pre-call and post-call information

This information is gathered immediately before and after calling any given functions. It provides a record of where a call was made, and information on the performance of the operation called.

- Procedure entry and exit information

This information is gathered immediately upon entry into a procedure and exit from that procedure. A snapshot is taken of the current performance statistics when entering a procedure, and a calculation is made of the differences in those statistics when exiting that procedure.

When performance collection code is generated into a leaf procedure, the procedure is changed so that it is no longer a leaf procedure. (A leaf procedure is one that does not call any other procedures.) This is because the leaf procedure now contains hooks to call the performance collection routines. This can be a time-consuming process.

See the *ILE C/C++ Compiler Reference* for information on these options.

Minimizing Exception Handling

To minimize exception handling, you can try:

- [Turning on return codes during record I/O](#)
- [Turning off C2M messages during record I/O](#)
- [Using direct monitor handlers](#)
- [Minimizing percolation of exceptions](#)

See "[Handling Exceptions in a Program](#)" on page 245 for information on handling exceptions.

Turning on Return Codes during Record I/O

Exceptions are expensive to process.

If you use record I/O, you can minimize exceptions by using the `rtncode=y` option on `_Ropen()`. Exceptions are not generated for the following conditions:

- "Record not found" (CPF5006)
- "End-of-File" (CPF5001)

When these conditions occur, the `num_bytes` field of the `_RIOFB_T` structure is updated and `errno` is set, but no exceptions are generated. For the "Record not found" condition, the `num_bytes` field is set to zero. For the "End-of-File" condition, the `num_bytes` field is set to EOF.

Turning Off C2M Messages during Record Input and Output

To turn off C2M messages during record I/O, set the variable `_C2M_MSG` (in `<recio.h>`) to zero. If `_C2M_MSG` is set to a different value, record I/O sends C2M messages to your program when it detects any of the following errors: C2M3003, C2M3004, C2M3005, C2M3009, C2M3014, C2M3015, C2M3040, C2M3041, C2M3042 and C2M3044.

Note: Removing data truncation messages with signal handlers or message handlers is no longer necessary when the C2M messages are turned off during record I/O.

Using a Direct Monitor Handler

When an exception occurs, the compiler first attempts to use any direct monitor handler. If there is no direct monitor handler, the exception is mapped to a signal at runtime, and the corresponding signal handler is called. By using the `#pragma exception_handler` directive to enable a direct monitor handler, you avoid the process for both the signal mapping and search for a signal handler.

For all exceptions specified by the `#pragma exception_handler` directive, the direct monitor handler marks each exception as handled; otherwise the exception is percolated again.

For information about `#pragma exception_handler`, see the *ILE C/C++ Compiler Reference*.

For more information about using direct monitor handlers, see the *ILE C/C++ Compiler Reference*.

Minimizing Percolation of Exceptions

Try to handle an exception in the place it occurs. There is some processing overhead incurred with exception percolation.

Example of Exception Percolation for a Sample ILE C Source Code

The following figure shows an example of ILE C source code for handling exceptions. Below the figure is an example of an exception that can occur and the steps the code takes to handle the exception.

```

#include <stdio.h>
#include <except.h>
#include <signal.h>
#include <lecond.h>
void handler1(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    printf("In handler1: will not handle the exception\n");
}
void handler2(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    printf("In handler2: will not handle the exception\n");
}
void handler3(_FEEDBACK *condition, _POINTER *token, _INT4 *result_code,
              _FEEDBACK *new_condition)
{
    printf("In handler3: will not handle the exception\n");
}
void handler4(_INTRPT_Hndlr_Parms_T * __ptr 128 parms)
{
    printf("In handler4: will not handle the exception\n");
}
void fred(void)
{
    _HDLR_ENTRY hdlr = handler3;
    char *p = NULL;
    #pragma exception_handler(handler2, 0, 0, \
                              _C2_MH_ESCAPE | _C2_MH_FUNCTION_CHECK)
    CEEHDLR(&hdlr, NULL, NULL);
    #pragma exception_handler(handler1, 0, 0, _C2_MH_ESCAPE)
    *p = 'x'; /* exception */
}
int main(void)
{
    signal(SIGSEGV, SIG_DFL);
    #pragma exception_handler(handler4, 0, 0, \
                              _C2_MH_ESCAPE | _C2_MH_FUNCTION_CHECK)
    fred();
}

```

Figure 39. T1520XH7 – ILE C Source for Exception Handling

The sequence of exceptions and handling actions that occur when the source code in [Figure 39 on page 60](#) is run is:

1. An escape exception occurs in function `fred()`.
2. `handler1` gets control because it is monitoring for an escape message, it is the closest nested monitor to the exception, and it has highest priority because it is a direct handler.
3. `handler2` gets control because it is monitoring for an escape message, and has a higher priority than a CEEHDLR.
4. `handler3` gets control (from CEEHDLR).
5. signal handler gets control. Even though it is registered in `main`, `signal` is scoped to the activation group and therefore will get control. It gets control after `handler1`, `handler2`, and `handler3` because it has a lower priority than either direct handlers or CEEHDLRs. Because the action is `SIG_DFL`, the exception is not handled.
6. The exception is percolated to `main()`.
7. `handler4` gets control.
8. The exception is still not handled. Thus, when it hits the control boundary (the PEP for `main()`), it is turned into a function check and is re-driven.
9. `handler1` does NOT get control, because it is not monitoring for a function check.
10. `handler2` gets control because it is monitoring for function check.
11. `handler3` gets control because CEEHDLRs get control for all `*ESCAPE`, `*STATUS`, `*NOTIFY`, and Function Check messages.
12. signal handler does NOT get control because `signal` does not recognize function checks.
13. The function check is percolated to `main()`.
14. `handler4` gets control because it is monitoring for function check.

15. The function check percolates to the control boundary and causes the ending.
16. (CEE9901) *ESCAPE is sent to the caller of `main()`.

Reducing the Number of Function Calls and Arguments


Extra processing is involved in accessing the return value of a program call.

You can reduce the number of function calls and arguments by:

- [Inlining function calls](#)
- [Using static class members or global variables](#)
- [Passing arguments in registers](#)
- [Using prototypes to minimize function call processing](#)

Inlining Function Calls

When a function is called in a few places but executed many times, changing the function to an inline function typically saves many function calls and results in performance improvement. You might be able to improve performance by changing function calls to inline functions or macro expressions, provided such a change does not increase the size of the program object and cause enough page faults to slow the program down. To optimize performance, strike a balance between program size and inlining or macro expressions. See Table 10 on page 73.

Note:  In C++, macro expressions are not recommended. Instead, use the `inline` keyword and turn on inlining.


The `INLINE` compile time option allows you to request that the compiler replace a function call with that function's code in place of the function call. If the compiler allows the inlining to take place, the function call is replaced by the machine code that represents the source code in the function definition.

Inlining is a method that allows you to improve the runtime performance of a C or C++ program by eliminating the function call overhead. Inlining allows for an expanded view of the program for optimization. Exposing constants and flow constructs on a global scale allows the compiler to make better choices during optimization.

For information about inlining and expanding macros, see:

- *ILE C/C++ Language Reference*
- *ILE C/C++ Compiler Reference*

Using Static Class Member Functions or Global Variables

 You might be able to improve runtime performance of a C++ program by using static class members to pass an argument to a function.

In a C or C++ program, an alternative to passing an argument to a function is to have the variable defined as being global and to have the function use the global variable.

Note: Using more global variables increases the amount of work that has to be done at activation group startup to allocate and initialize the global variables, which can inhibit optimization.

For information about class member functions and global variables, see:

- [“Minimizing the Use of Static and Global Variables” on page 57](#)
- *ILE C/C++ Language Reference*
- *ILE C/C++ Compiler Reference*

Passing Arguments in Registers

Function call performance can be improved if the system has all of the arguments passed in registers. Because there are only a limited number of registers, in order to increase the chance of having all

arguments passed in registers, combine several arguments into a class and pass the address of the class to the function. Because an address is being passed, pass-by-reference semantics are used, which may not have been the case when the arguments were being passed as individual variables.

For more information about passing arguments in registers, see:

- *ILE C/C++ Language Reference*
- *ILE C/C++ Compiler Reference*

For more information about storage classes, see:

- [“Using the Register Storage Class” on page 57](#)
- *ILE C/C++ Language Reference*

Using Prototypes to Minimize Function Call Processing

A *function prototype* consists of the function return type, the name of the function, and the parameter list. An un-prototyped function has its signature inferred by the data model in effect at the time of its first reference.

Note: C++ requires full prototype declarations. ISO C allows non-prototyped functions.

C++ When calling a program dynamically from a C++ program using extern OS linkage, prototype the program to return void rather than int. Extra processing is involved in accessing the return value of a program call. Passing the address of storage that can hold a return value in the call's argument list is better from a performance viewpoint.

For information about function prototypes, see the *ILE C/C++ Language Reference*.

Choosing Input and Output Functions to Improve Performance

This section covers some Input and Output issues.

Using Record Input and Output Functions

Using record I/O functions instead of stream I/O functions can greatly improve I/O performance. Instead of accessing one byte at a time, record I/O functions access one record at a time.

The two types of record I/O supported by the ILE C/C++ runtime libraries are ISO C record I/O and ILE C record I/O.

ISO C Record I/O

If you use an ISO C record I/O in your program, you must specify `type = record` in the open mode parameter of `fopen()` when you open a file, and you must use the FILE data type. The following figure provides an example.

```
#include <stdio.h>
#define MAX_LEN 80
int main(void)
{
    FILE *fp;
    int len;
    char buf[MAX_LEN + 1];
    fp = fopen("MY_LIB/MY_FILE", "rb, type = record");
    while ((len = fread(buf, 1, MAX_LEN, fp)) != 0)
    {
        buf[len] = '\0';
        printf("%s\n", buf);
    }
    fclose(fp);
    return 0;
}
```

Figure 40. Example: Using ISO C Record I/O

ILE C Record I/O

If you use ILE C record I/O in your program, you must:

- Use the ILE C record I/O functions (for example, functions that begin with `_R`)
- Use the `_RFILE` data type.

The example in [Figure 40](#) on [page 62](#) can be rewritten as follows:

```
#include <stdio.h>
#include <recio.h>
#define MAX_LEN 80
int main(void)
{
    _RFILE *fp;
    _RIOFB_T *iofb;
    char buf[MAX_LEN + 1];
    fp = _Ropen("MY_LIB/MY_FILE", "rr");
    iofb = _Rreadn(fp, buf, MAX_LEN, __DFT);
    while ( iofb->num_bytes != EOF )
    {
        buf[iofb->num_bytes] = '\0';
        printf("%s\n", buf);
        iofb = _Rreadn(fp, buf, MAX_LEN, __DFT);
    }
    _Rclose(fp);
    return 0;
}
```

Figure 41. Example: Using ILE C Record I/O

Using Input and Output Feedback Information

`_RIOFB_T` is a structure that contains I/O feedback information from ILE C record functions, for example, the number of bytes that are read or are written. By default, the ILE C record I/O functions update the fields in `_RIOFB_T` after a record I/O operation is performed.

If your program does not use all these values, you can improve your application's performance by opening a file as shown in the following figure:

```
fp = _Ropen("MY_LIB/MY_FILE", "rr, riofb = N");
```

Figure 42. I/O Feedback Information

By specifying `riofb = N`, only the `num_bytes` field (the number of bytes read or written in the `_RIOFB_T` structure) is updated. If you specify `riofb = Y`, all fields in the `_RIOFB_T` structure are updated.

Blocking Records

You can improve record I/O performance by blocking records. When blocking is specified, the first read causes a whole block of records to be placed into a buffer. Subsequent read operations return a record from the buffer until the buffer is empty. At that time, the next block is fetched.

If you wish to block records when the `FILE` data type is used, open the file with `blksize=value` specified, where `value` indicates the block size. If `blksize` is specified with a value of 0, a block size is calculated for you when you open a file.

If you wish to block records when the `_RFILE` data type is used, specify `blkrcd = Y` when you open the file.

Similar rules apply when blocking records for write operations.

Manipulating the System Buffer

You can improve I/O performance of your ILE C/C++ programs by performing read and write operations directly to and from the system buffer, without the need for an application-defined buffer. This system

access is referred to as locate mode. The following illustrates how to directly manipulate the system buffer when reading a source physical file.

```
fp = _Ropen("MY_LIB/MY_FILE", "rr, blkrcd = Y, riofb = N");
while ( (_Rreadn(fp, NULL, 92, __DFT))->num_bytes != EOF )
{
    printf("%75.75s\n", ((char *) (*(fp->in_buf))) + 12);
}
_Rclose(fp);
```

Figure 43. Using the System Buffer

The example code above prints up to 75 characters of each record that is contained in the file. The second parameter for the `_Rreadn()`, `NULL`, allows you to manipulate the record in the system buffer. An `_RFILE` structure contains the `in_buf` and `out_buf` fields, which point to the system input buffer and system output buffer, respectively. The example above prints each record by accessing the system's input buffer.

Directly manipulating the system buffer provides a performance improvement when you process very long records. It also provides a significant performance improvement when you use InterSystem Communications Function (ICF) files. Usually, you only need to access the last several bytes in an ICF file and not all the other data in the record. By using the system buffer directly, the data that you do not use for ICF files need not be copied.

The system buffer should always be accessed through the `in_buf` and `out_buf` pointers in the `_RFILE` structure that is located in the `<recio.h>` header file. Unpredictable results can occur if the system buffer is not accessed through the `in_buf` and `out_buf` pointers.

Opening Files Once for Both Input and Output

If your application writes data into a file and then reads the data back, you can improve performance by opening the file only once, instead of the usual two times to complete both input and output. The following illustrates how a file is opened twice and closed twice:

```
fp = _Ropen("MY_LIB/MY_FILE", "wr"); /* Output only.*/
/* Code to write data to MY_FILE */
_Rclose(fp);
fp = _Ropen("MY_LIB/MY_FILE", "rr"); /* Other code in your application. */
/* Input only.*/
/* Code to read data from MY_FILE. */
_Rclose(fp);
```

Figure 44. Example: Opening a File Twice

By changing this example to the following, one call to `_Ropen`, and one call to `_Rclose` is saved:

```
fp = _Ropen("MY_LIB/MY_FILE", "ar+"); /* Input and output.*/
/* Code to write data to MY_FILE. */
/* Other code in your application. */
/* Code to read data from MY_FILE. */
/* Use either _Rreadf or _Rlocate
/* with the option __FIRST. */
_Rclose(fp);
```

Figure 45. Example: Opening a File Once

Minimizing the Use of Shared Files

You can improve performance by not opening the same file more than once in an application. You can allocate the file pointers as global (external) variables, opening the files once, and not closing the file until the end of the application.

Minimizing the Number of File Opens and Closes

Open and close are very expensive operations. You can improve performance by opening and closing files only as often as necessary. You can use a class to encapsulate I/O operations such as opening the files once, and not closing the file until the end of the program.

Defining Tape Files to Improve Performance

You can improve the performance of programs that use tape files by using fixed-length record tape files instead of variable-length tape files.

Improving Performance when Using Stream Input and Output Functions

Although using ILE C record I/O functions improves performance more effectively than stream I/O functions, there are still ways to improve performance when using stream I/O.

You can use IFS stream files, with performance similar to record I/O functions, by specifying SYSIFCOPT(*IFSIO) on the CRTCMOD or CRTBNDC commands.

You should use the macro version of `getc` instead of `fgetc()` to read characters from a file. See [“Using Static Class Member Functions or Global Variables”](#) on page 61. The macro version of `getc()` reads all the characters in the buffer until the buffer is empty. At this point, `getc()` calls `fgetc()` to get the next record.

For the same reason, you should use `putc()` instead of `fputc()`. The macro version of `putc()` writes all the characters to the buffer until the buffer is full. At this point, `putc()` calls `fputc()` to write the record into the file.

Because stream I/O functions cause many function calls; reducing their use in your application improves performance. The following illustrates calls to `printf()`:

```
printf("Enter next item.\n");
printf("When done, enter 'done'.\n");
```

Figure 46. Using `printf()`

The two calls to `printf()` can be combined into a single call so that one call is saved as follows:

```
printf("Enter next item.\n"
      "When done, enter 'done'.\n");
```

Figure 47. Using `printf()` to Reduce Function Calls

Using C++ Input and Output Stream Classes



Use overloaded shift `<<` `>>` operators on the standard streams, instead of their C equivalents.

Using Physical Files Instead of Source Physical Files

To improve performance, use physical files instead of source physical files for your data.

When a source physical file is used for stream I/O, the first 12 bytes of each record are not visible to your application. They are used to store the record number and update time. These 12 bytes are an extra load that the ILE C stream I/O functions must manipulate. For example:

- When performing output, these 12 bytes must be initialized to zero.
- When performing input, these 12 bytes must be fetched even though they are not passed to your application.

Because the ILE C stream I/O functions dynamically create a source physical file when opening a text file that does not already exist for output, create the file as a physical file before you start your application.

Specifying Library Names

You should specify the name of the library in which the file resides. If you do not specify a library name when processing a file, the library list is searched for the file. The search time can be lengthy, depending on the number of libraries and the objects that they contain.

Using Pointers to Improve Performance

Using and comparing pointers can impact performance.

Avoiding Use of Open Pointers

Avoid using open pointers. Open pointers inhibit optimization. Note that pointers to void (`void*`) are open pointers in ILE C/C++.

Avoiding Pointer Comparisons

Because pointers take up 16 bytes of space, pointer comparisons are less efficient than comparisons using other data types. You might want to replace pointer comparisons with comparisons using other data types, such as `int`.

The following figure shows a program that constructs a linked list, processes all the elements in the list, and then frees the linked list:

```

#include <string.h>
#include <stdlib.h>
#include <recio.h>

#define MAX_LEN 80

struct link
{
    struct link *next;
    char record[MAX_LEN];
};

int main(void)
{
    struct link *start, *ptr;
    _RFILE *fp;
    return 0;
int i;

// Construct the linked list and read in records.

fp = _Ropen("MY_LIB/MY_FILE", "rr, blkrcd = Y");
start = (struct link *) malloc(sizeof(struct link));
start->next = NULL;
ptr = start;
for ( i = (_Ropnfbk(fp))->num_records; i > 0; --i )
    {
        _Rreadn (fp, NULL, MAX_LEN, _DFT);
        ptr = ptr->next = (struct link *) malloc(sizeof(struct link));
        memcpy(ptr->record,(void const *) *(fp->in_buf), MAX_LEN);
        ptr->next = NULL;
    }
ptr = start->next;
free(start);
start = ptr;
_Rclose(fp);
return 0;

// Process all records.

for ( ptr = start; ptr != NULL; ptr = ptr->next )
    {

// Code to process the element pointed to by ptr.

    }

// Free space allocated for the linked list.

while ( start != NULL )
    {
        ptr = start->next;
        free(start);
        start = ptr;
    }
}

```

Figure 48. Example of a Program that Uses Linked Lists

Each element in the link list holds one record from a file:

In the preceding program, pointer comparisons are used when processing elements and freeing the linked list. The program can be rewritten using a `short` type member to indicate the end of the link list. As a result, you change pointer comparisons to integer comparisons, as shown in the following figure:

```

#include <string.h>
#include <stdlib.h>
#include <recio.h>

#define MAX_LEN 80
int i;

struct link
{
    struct link *next;
    short last;
    char record[MAX_LEN];
};

int main(void)
{
    struct link *start, *ptr;
    _RFILE *fp;
    return 0;

    // Construct the linked list and read in records.

    fp = _Ropen(" MY_LIB/MY_FILE", "rr, blkrcd = Y");
    start = (struct link *) malloc(sizeof(struct link));
    start->next = NULL;
    ptr = start;
    for ( i = (_Ropnfbk(fp)->num_records; i > 0; --i )
    {
        _Rreadn(fp, NULL, MAX_LEN, __DFT);
        (struct link *) malloc(sizeof(struct link));
        memcpy(ptr->record, (void const *) *(fp->in_buf), MAX_LEN);
        ptr->last = 0;
    }
    ptr->last = 1;
    ptr = start->next;
    free(start);
    start = ptr;
    _Rclose(fp);

    // Process all records.

    if ( start != NULL )
    {
        for ( ptr = start; !ptr->last; ptr = ptr->next )
        {
            // Code to process the element pointed to by
            // code to process the element
            //(last element) pointed.
            // Free space allocated for the linked list.

            while ( !start->last )
            {
                ptr = start->next;
                free(start);
                start = ptr;
            }
            free(start);
        }
    }
}

```

Figure 49. Example of Source Code that Uses a short Type Member to End a Linked List

Reducing Indirect Access through Pointers

You can improve performance by reducing indirect access through pointers. Each level of indirection adds some overhead:

```

for ( i = 0; i < n; i++ )
{
    x->y->z[i] = i;
}

```

Performance in the above example improves if it is rewritten as:

```
temp = x->y;
for ( i = 0; i < n; i++ )
{
    temp->z[i] = i;
}
```

Using Shallow Copy instead of Deep Copy

Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on.

Note: If a program points to an object more than once, you must use deep copy. Objects that use shallow copy can destroy objects pointed to more than once.

Minimizing Space Requirements

You can improve the performance of a program by reducing the space it requires. Reducing the space requirement helps reduce page faults, segment faults, and effective address overflows.

Choosing Appropriate Data Types

Choosing the appropriate data type can reduce program space requirements and help improve program performance. When choosing data types, consider all the platforms that your code must support. You may not know all the data types and sizes at the beginning of your code design. Because the data types can hold the same size data on various platforms, you can use typedefs, enums, or classes depending on the use of the data type. If possible use `short` instead of `int`, and `float` instead of `double`. The compiler uses 2 bytes for `short`, 4 bytes for `int`, and 8 bytes for `double`.

Minimizing Dynamic Memory Allocation Calls

You can improve performance by reducing the number of times you dynamically allocate memory. Every time you call the `new` operator a certain amount of space is allocated from the heap. This space is always aligned at 16 bytes, which is suitable for storage of any object type. In addition, 32 extra bytes are taken from the dynamic heap for bookkeeping. This means that even if you only want one byte, 48 bytes are allocated from the dynamic heap, 32 bytes for bookkeeping and 15 bytes for padding. When the current space allocation in the heap is used up, storage allocation is slower:

```
ptr1 = new char[12];
ptr2 = new char[4];
```

In the code above, 96 bytes are taken from the heap (including 64 bytes for bookkeeping and 16 bytes for padding) and `new` is used twice. This code can be rewritten as:

```
ptr1 = new char[16];
ptr2 = ptr1 + 12;
```

Only 48 bytes are taken from the heap and the `new` operator is only used once. Because you reduce the dynamic space allocation requirement, less storage is taken from the heap. You may gain other benefits such as a reduction in page faults. Because there are fewer calls to the `new` operator, function call overhead is reduced as well.


Note: If you allocate space by incrementing pointers, you must guarantee the proper alignment when you allocate pointers (or aggregates which can contain pointers) because pointers require 16-byte alignments. There is a performance degradation for data types such as `float` if they are not allocated on their natural boundaries because these data types have a natural alignment of word or doubleword.


Arranging Variables to Reduce Padding

Reducing space wasted on padding by rearranging variables is another way to reduce your program's space requirement.

With the exception of packed decimal data types, variables are padded the same in both C and C++:

- A `char` type variable takes one byte
- A `short` type variable takes 2 bytes
- An `int` type variable takes 4 bytes
- A `long` type variable takes 4 bytes
- A `longlong` type variable takes 8 bytes
- A `float` type variable takes 4 bytes
- A `double` type variable takes 8 bytes
- A pointer takes 16 bytes
- A `_DecimalT` template class object takes 1 to 16 bytes

 A C packed decimal type variable can be 1–32 bytes in size.

 A C++ packed decimal type variable can be 1–16 bytes in size.

By rearranging variables, wasted space created by padding can be minimized, as shown in [Figure 50 on page 71](#).


```

class OrderT
{
    float value;      // Four bytes.
    char flag1;      // One byte plus one byte.
    short num;       // Two bytes.
    char flag2;      // One byte plus three bytes.
}
orderT;

```

```

class ItemT
{
    char *name;           // 16 bytes.
    int number;          // 4 bytes plus 12 bytes.
    char *address;       // 16 bytes.
    double value;        // 8 bytes plus 8 bytes.
    char *next;          // 16 bytes.
    short rating;        // 2 bytes plus 14 bytes.
    char *previous;      // 16 bytes.
    _DecimalT<25,5> tot_order; // 13 bytes plus 3 bytes.
    int quantity;        // 4 bytes.
    _DecimalT<12,5> unit_price; // 7 bytes plus 5 bytes.
    char *title;         // 16 bytes.
    char flag;           // 1 byte plus 15 bytes.
}
itemT;

```

Note: The structure of the ItemT class takes 176 bytes, of which 57 bytes are used for padding.

The ItemT class can be rearranged as:

```

class ItemT
{
    char *name;           // 16 bytes
    char *address;       // 16 bytes
    char *next;          // 16 bytes
    char *previous;      // 16 bytes
    char *title;         // 16 bytes
    double value;        // 8 bytes
    int quantity;        // 4 bytes
    int number;          // 4 bytes
    short rating;        // 2 bytes
    char flag;           // 1 byte
    _DecimalT<25,5> tot_order; // 13 bytes
    _DecimalT<12,5> unit_price; // 7 bytes plus 9 bytes
}
itemT;

```

Note: After rearrangement, the ItemT class takes only 128 bytes, with 9 bytes for padding. The saving of space is even more substantial when you are rearranging arrays of similar structure type.

Figure 50. Example of Minimizing Padding by Rearranging Variables

As a general rule, the space used for padding can be minimized if 16-byte variables are declared first, 8-byte variables are declared second, 4 byte variables are declared third, 2-byte variables are declared fourth, and 1-byte variables are declared fifth. `_DecimalT` template class objects should be declared last, after all other variables have been declared. The same rule can be applied to structure or class definitions.

To show the layout (including padding) of module structures, in both packed and normal alignment, use either the `*AGR` or the `*STRUCREF` compiler option.

Note: `*AGR`, which gives a map of all structures, overrides `*STRUCREF`, which gives a map of referenced structures.

Removing Observability

A module has *observability* when it contains data that allows it to be changed without being compiled again. Two types of data can make a module observable:

Create Data

This data is necessary to translate the code into machine instructions. The object must have this data before you can change the optimization level. It is represented by the **CRTDTA* value on the *RMVOBS* parameter of the Change Program (CHGPGM) command.

Debug Data

This data enables an object to be debugged. It is represented by the **DBGDTA* value on the *RMVOBS* parameter of the CHGPGM command.

The addition of these types of data increases the size of the object. Consequently, you may at some point want to remove the data in order to reduce object size. However, once the data is removed, so is the object's observability. To regain it, you must recompile the source and re-create the program.

To remove either kind of data from a program or module, use the CHGMOD or the CHGPGM command. Again, once you have removed the data, it is not possible to change the object in any way unless you re-create it. Therefore, ensure that you have access to all source required to create the program, or that you have a comparable program object with create data.

Compressing Objects

The Create Data (**CRTDTA*) value associated with an ILE program or module may make up more than half of the object's size. By removing or compressing this data, you reduce the secondary storage requirements for your programs significantly.

An alternative is to compress the object through using the Compress Object (CPROBJ) command. A compressed object takes up less system storage than an uncompressed one. When the compressed program is called, the part of the object containing the executable code is automatically decompressed. You can also decompress an object by using the Decompress Object (DCPOBJ) command.

Optimizing Use of Activation Groups

Using activation groups can impact performance.

Calling Functions in Other Activation Groups

Within the same job, calling a function that runs in a different activation group degrades the performance of the call significantly (the call takes approximately twice as long).

If a service program was created to run in a named activation group (using the *ACTGRP(name)* parameter of the *CRTSRVPGM* command) then any calls to that function from a program or service program would be calling *across an activation group* and would therefore be slower. Sometimes it makes sense to run programs or service programs in other activations groups (for storage isolation, exception handling) but it should be noted that call-performance suffers in that arrangement.

Reducing Program Startup Time

When a new ILE program is first called, the system needs to perform some initialization to prepare the program to run. Part of this initialization requires creating an activation group for all of the program storage, resolving all service programs bound to the program, getting program arguments and so on. Several recommendations for improving start-up time can be drawn from these initialization steps:

- Reduce the use of global variables.
- Reduce the number of service programs bound to the program. The more service programs used by an ILE program, the more time is required to start up the program. It is often better to have a few large service programs than many small ones. For example, the C runtime libraries comprise a small number of service programs.

Minimizing Use of Virtual Functions



There is a performance impact if you use virtual functions because virtual functions are compiled to be indirect calls, which are slower than direct calls. You may be able to minimize this performance impact depending on your program design by using a minimum number of parameters on the virtual functions.

Choosing Compiler Options to Optimize for Speed or Size

There are several ways to improve compile time performance. These include both front end and back end compile time activities.

Table 10 on page 73 describes different compiler options to make your program run faster, and to make your compiled program smaller. Note that sometimes you have to decide which is more important to you, program size or program speed. In some cases optimizing for one aspect means the other suffers.

Optimization is the process through which the system looks for processing shortcuts that reduce the amount of system resources necessary to produce output. Processing shortcuts are translated into machine code, allowing the procedures in a module to run more efficiently. A highly optimized program or service program should run faster than it would without optimization.

To control the level of optimization, use the OPTIMIZE option on the Create Module and Create Bound Program commands. Changing the desired optimization level requires recompiling your source code. Changing the optimization of a module can also be accomplished through a Change Module (CHGMOD) command.

Note: You cannot use the Change Module (CHGMOD) command to change the optimization level without recompiling your source code.

You should be aware of the following limitations when working with optimized code:

- In general, the higher the optimizing request, the longer it takes to create an object.
- At higher levels of optimization, the values of fields may not be accurate when they are displayed in a debug session, or after the program recovers from an exception.
- Optimized code may have altered breakpoints and step locations used by the source debugger because the optimization changes may rearrange or eliminate some statements.

To circumvent this restriction while debugging, you can lower the optimization level of a module to display fields accurately as you debug a program, and then raise the level again afterwards, to improve the program efficiency as you get the program ready for production.

Use the guidelines in Table 10 on page 73, except where they are contradicted. Intrinsic functions may improve performance, but they increase the size of your module.

Option	Optimize for Speed	Optimize for Size
OPTIMIZE 10//Default value OPTIMIZE 20 OPTIMIZE 30 OPTIMIZE 40 Turns on optimization.	Yes	Yes
INLINE(*OFF) Turns off inlining. May reduce module size, especially if the inlined functions consist of small pieces of code.	No	Yes

Table 10. Compiler Options for Performance (continued)

Option	Optimize for Speed	Optimize for Size
<p data-bbox="228 285 386 310">INLINE(*ON)</p> <p data-bbox="228 338 1076 401">Turns on inlining. Saves many function calls when a function is called in a few places but executed many times.</p>	Yes	No
<p data-bbox="228 441 423 466">DBGVIEW(*NONE)</p> <p data-bbox="228 493 1040 556">Does not generate debug information, which would increase module size.</p>	No	Yes

Setting Runtime Limits

- The maximum amount of storage of any single variable (such as a string or array) is 16 773 104 bytes
- The maximum length of a command passed to the system function is 32 702 bytes
- The maximum size of dynamic heap storage is 4 gigabytes

A very large memory allocation may cause a system crash if there is insufficient auxiliary storage on your system. A 4 gigabyte memory allocation requires more than 4 gigabytes of available DASD. The Work System Status (WRKSYSSTS) command shows auxiliary storage usage.

- The maximum size of a single heap allocation is 16 711 568 bytes
- The maximum auto storage is 16 MB and there is a recursion limit of approximately 21743 levels deep

Example: Creating an ILE C Application

The example in this section demonstrates some typical steps in creating a sample ILE C application.

The application is a small transaction-processing program that takes item names, price, and quantity as input. As output, the application displays the total cost of the items on the screen and updates an audit trail of the transaction.

This topic describes, for the sample application, the following:

- Process flow
- ILE activation group
- Resource requirements
- Task summary
- Step-by-step instructions
- Source code

Process Flow

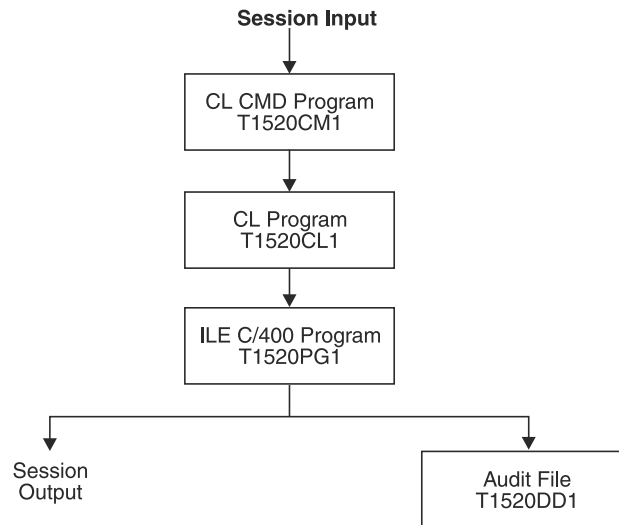


Figure 51. Sample Application: High-Level Input/Processing/Output Flow

Figure 51 on page 75 shows:

Session input

Data entered during a terminal session:

- Name of the item being ordered
- Price per unit
- Quantity of units being ordered

CL CMD Program T1520CM1

A developer-created CL command that accepts user input and passes it to CL Program T1520CL1.

CL Program T1520CL1

A CL program that processes the input, and passes it to ILE C Program T1520PG1.

ILE C Program T1520PG1

An ILE C program that processes the input and directs output to the user's terminal and to an externally described file. The ILE C program consists of two modules: T1520IC1 and T1520IC2, as shown in Figure 52 on page 76. Module T1520IC1 provides the user entry procedure `main()`, which calls the `calc_and_format()` procedure.

Service Program T1520SP1

An ILE service program that makes the `write_audit_trail()` procedure available for a program to import, as shown in Figure 52 on page 76.

Service Program T1520SP2

An ILE service program that makes the tax rate data item available for a program to import, as shown in Figure 52 on page 76.

Session Output

The following appears on the screen:

- A statement: "*(Quantity of units being ordered) (Name of item being ordered) plus tax = (Calculated cost to user)*"
- A prompt: "Press ENTER to end terminal session."

Audit File T1520DD1

A log that is updated with each transaction. The DDS source, shown in Figure 53 on page 81, defines the data fields and relationships (that is, layout) of the audit file.

ILE Activation Group

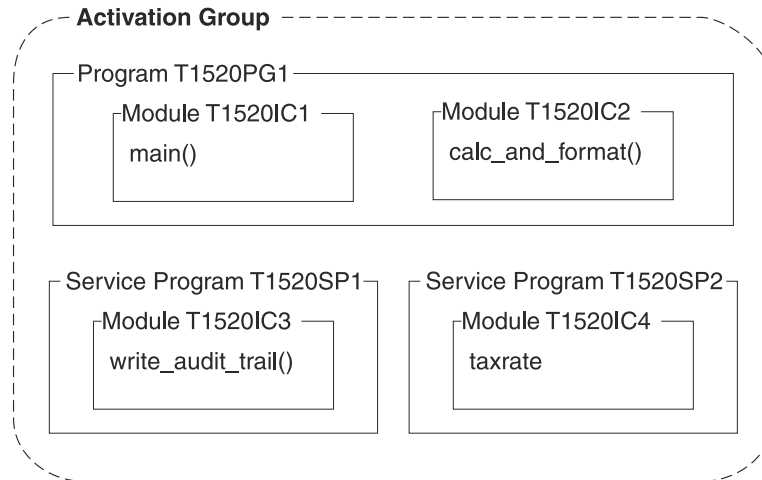


Figure 52. ILE Activation Group of the Sample Application

When the CL Command program calls the CL program, all the resources necessary to run these programs are allocated in the default activation group.

When the CL program calls the ILE C program, a new activation group is started, as shown in [Figure 52 on page 76](#), because the ILE C program is created with the `ACTGRP(*NEW)` parameter. The ILE C service programs are also activated in this new activation group because they are created with the `ACTGRP(*CALLER)` parameter.

In other words, the ILE C program and ILE C service programs are activated within one activation group because the programs are developed as one cooperative application.

Note: To isolate programs from other programs running in the same job you can select different activation groups. For example, a complete customer solution may be provided by integrating software packages from four different vendors. Different activation groups ease the integration by isolating the resources associated with each vendor package.

Resource Requirements

To create the sample application, you need to create the following resources:

- A binding directory for the ILE C application modules and service programs
- Source code for:
 - Data Description Specification (DDS) [T1520DD1](#) for the output file.
 - CL Program [T1520CL1](#)
 - CL CMD Program [T1520CM1](#)
 - ILE C Program T1520PG1 modules [T1520IC1](#) and [T1520IC2](#)
 - Service program T1520SP1 module [T1520IC3](#) and its binder language [QSRVSRC](#)
 - Service program T1520SP2 module [T1520IC4](#) and its binder language [QASRVSR](#)

Note: This example is for illustration purposes. It is not necessary to create a binding directory for an ILE C program of this size. You might not want to break up ILE C service programs by data and function as shown.

Task Summary

Table 11 on page 77 lists the tasks you must perform to create the sample ILE application. Each task number is linked to the corresponding procedure step. Each component name is linked to the figure that contains its source code.

Table 11. Summary of Tasks Required to Create Sample ILE Application

Task		Component
<u>1</u>	Create the physical file to contain the audit log. The DDS source defines the fields for the audit file.	T1520DD1
<u>2</u>	Create the CL program that passes required parameters to the ILE C program T1520PG1.	T1520CL1
<u>3</u>	Create the CL command prompt that collects data from the user's terminal session.	T1520CM1
<u>4</u>	Create the module that provides the UEP (main() function), which: <ul style="list-style-type: none"> • Receives the user input from the CL program T1520CL1 • Calls <code>calc_and_format()</code> function in module T1520IC2, which: <ul style="list-style-type: none"> – Calculates an item's total cost – Calls the <code>write_audit_trail()</code> function in module T1520IC3 	T1520IC1
<u>5</u>	Create the module that provides the called function <code>calc_and_format()</code> , which completes the tax calculation by: <ul style="list-style-type: none"> • Receiving arguments from module T1520IC1 • Importing the tax rate data item from an ILE C service program Note: The function <code>calc_and_format()</code> also formats the total cost.	T1520IC2
<u>6</u>	Create the module that provides the <code>write_audit_trail()</code> function. This module creates the ILE C service program T1520SP1.	T1520IC3
<u>7</u>	Create the module that exports the tax rate data. This module creates the ILE C service program T1520SP2.	T1520IC4
<u>8</u>	Create a source physical file QSRVSRC that contains the binder language to export the procedure <code>write_audit_trail</code> from ILE Service Program T1520SP1.	QSRVSRC
<u>9</u>	Create the source physical file QASRVSRC that contains the binder language to export the data item <code>taxrate</code> from ILE service program T1520SP2.	QASRVSRC
<u>10</u>	Create the binding directory that contains the service programs T1520SP1 and T1520SP2 and add the service program names to the directory.	T1520BD1
<u>11</u>	Create the ILE C service program T1520SP1 from: <ul style="list-style-type: none"> • Module T1520IC3, which exports the tax rate data • The physical file (QSRVSRC), which contains the binder language source¹ 	T1520IC3QSRVSRC
<u>12</u>	Create the ILE C service program T1520SP2 from: <ul style="list-style-type: none"> • Module T1520IC4, which exports the procedure <code>write_audit_trail</code>. • The physical file (QASRVSRC), which contains the binder language source¹ 	T1520IC4QSRVSRC
<u>13</u>	Create the ILE C program T1520PG1 from the following components: <ul style="list-style-type: none"> • T1520IC1 • T1520IC2 	T1520IC1TC1520IC2
<u>14</u>	Test the program by running it.	

¹A tool is provided in the QUSRTOOL library to help generate the binder language for one or more modules. See member TBNINFO in the file QUSRTOOL/QATTINFO.

Instructions to Create the Sample Application

1. Create the physical file T1520DD1, which contains the audit log entries, by entering the following command:

```
CRTPF FILE(MYLIB/T1520DD1) SRCFILE(QCPPL/QADDSSRC) MAXMBS(*NOMAX)
```

Note: Figure 53 on page 81 shows the source code in T1520DD1.

2. Create the CL program T1520CL1, which passes parameters to the ILE C program T1520PG1 by entering the following command:

```
CRTCLPGM PGM(MYLIB/T1520CL1) SRCFILE(QCPPL/QACLSRC)
```

Note: Figure 54 on page 81 shows the source code in T1520CL1.

3. Create the CL command prompt T1520CM1, which collects data for item name, price, and quantity by entering the following command:

```
CRTCMD CMD(MYLIB/T1520CM1) PGM(MYLIB/T1520CL1) SRCFILE(QCPPL/QACMSRC)
```

Note: Figure 55 on page 82 shows the source code in T1520CM1.

4. Create the module T1520IC1, which provides the main () function by entering the following command:

```
CRTCMOD MODULE(MYLIB/T1520IC1) SRCFILE(QCPPL/Q CSRC) OUTPUT(*PRINT) DBGVIEW(*ALL)
```

Notes:

- Figure 56 on page 83 shows the source code in T1520IC1.
 - OUTPUT(*PRINT) specifies that you want a compiler listing.
 - The parameter DBGVIEW(*ALL) specifies that you want a root source view and a listing view, along with debug data, to debug this module. See [“Working with Source Debug Sessions”](#) on page 99 for information on debug views and debug data.
5. Create the module T1520IC2, which calculates the tax and formats the total cost for output, by entering the following command:

```
CRTCMOD MODULE(MYLIB/T1520IC2) SRCFILE(QCPPL/QACSRC) OUTPUT(*PRINT)  
DBGVIEW(*ALL)
```

Notes:

- Figure 57 on page 84 shows the source code in T1520IC2.
 - DBGVIEW(*ALL) specifies that you want a root source view and a listing view, along with debug data to debug this module.
6. Create the module T1520IC3, which updates the audit trail in audit file T1520DD1, by entering the following command:

```
CRTCMOD MODULE(MYLIB/T1520IC3) SRCFILE(QCPPL/QACSRC)  
OUTPUT(*PRINT) DBGVIEW(*SOURCE) OPTION(*SHOWUSR)
```

Notes:

- [“Source Code to Write an Audit Trail”](#) on page 84 shows the source code in T1520IC3.
- The DBGVIEW(*SOURCE) OPTION(*SHOWUSR) parameters specifies that you want an include view containing the root source member, user include files, and debug data to debug this module.
- The OPTION(*SHOWUSR) parameter expands the type definitions generated by the compiler from the DDS source file MYLIB/T1520DD1, as specified on the #pragma mapinc directive, into the compiler listing and the include debug view.

7. Create the module T1520IC4, which exports the tax rate data, by entering the following command:

```
CRTCMOD MODULE(MYLIB/T1520IC4) SRCFILE(QCPPL/QACSRC) OUTPUT(*PRINT)
OPTION(*XREF) DBGVIEW(*SOURCE)
```

Notes:

- [Figure 58 on page 86](#) shows the source code in T1520IC4.
 - The OPTION(*XREF) parameter generates a cross reference table containing the list of identifiers in the source code with the numbers of the lines in which they appear. The table provides the class, length and type of the variable taxrate. The class is an external definition. The length is 2. The type is a constant decimal (2,2). The use of this option in this example is for illustrative purposes. Typically you use this option when there are several variable references or executable statements.
 - The DBGVIEW(*SOURCE) parameter creates a root source view and debug data to debug this module. If you do not specify DBGVIEW(*SOURCE), you can debug the modules that reference taxrate, but you cannot display taxrate during that debug session nor can you debug this module that defines taxrate.
8. Create a source physical file QSRVSRC, which contains the binder language to export the procedure write_audit_trail() from ILE Service Program T1520SP1, by entering the following command :

```
CRTSRCPF FILE(MYLIB/QSRVSRC) MBR(T1520SP1)
```

Note: [Figure 59 on page 86](#) shows the source code in MYLIB/QSRVSRC.

9. Create the source physical file QASRVSRC, which contains the binder language to export the data item taxrate, from ILE service program T1520SP2, by entering the following command:

```
CRTSRCPF FILE(MYLIB/QASRVSRC) MBR(T1520SP2)
```

Note: [Figure 60 on page 86](#) shows the source code in MYLIB/QASRVSRC.

10. Create the binding directory T1520BD1 in library MYLIB and add the two service program names (T1520SP1 and T1520SP2) to it.

a. To create the binding directory, enter the following command:

```
CRTBNDDIR BNDDIR(MYLIB/T1520BD1)
```

b. To add the service program names, enter the following commands:

```
ADDBNDDIRE BNDDIR(MYLIB/T1520BD1) OBJ((MYLIB/T1520SP1 *SRVPGM))
ADDBNDDIRE BNDDIR(MYLIB/T1520BD1) OBJ((MYLIB/T1520SP2 *SRVPGM))
```

Note: The service program names T1520SP1 and T1520SP2 can be added even though the service program objects do not yet exist.

Note: These instructions assume that the library MYLIB already exists.

11. Create the ILE C service program T1520SP1 from module T1520IC3 and the binder source language in QSRVSRC by entering the following command:

```
CRTSRVPGM SRVPGM(MYLIB/T1520SP1) MODULE(MYLIB/T1520IC3 MYLIB/T1520IC4)
SRCFILE(MYLIB/QSRVSRC) SRCMBR(*SRVPGM) BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
```

Notes:

- “Source Code to Write an Audit Trail” on [page 84](#) shows the source code in T1520IC3 and [Figure 58 on page 86](#) shows the source code in T1520IC4.
- Service program T1520SP1 needs both module T1520IC3 and module T1520IC4 because it exports the procedure write_audit_trail to satisfy an import request for function write_audit_trail in module T1520IC1, and the write_audit_trail procedure uses the data item taxrate, which is defined in module T1520IC4.

12. Create the ILE C service program T1520SP2 from module T1520IC4 and the binder source language in QASRVSRC by entering the following command:

```
CRTSRVPGM SRVPGM(MYLIB/T1520SP2) MODULE(MYLIB/T1520IC4) SRCFILE(MYLIB/QASRVSRC)
SRCMBR(*SRVPGM) BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
```

Note: Service program T1520SP2 exports the data item `taxrate` to satisfy an import request for variable `taxrate` in module T1520IC2.

13. Create the ILE C program T1520PG1 from components T1520IC1 and T1520IC2 by entering the following command:

```
CRTPGM PGM(MYLIB/T1520PG1) MODULE(MYLIB/T1520IC1 MYLIB/T1520IC2) ENTMOD(*ONLY)
BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
```

Notes:

- [Figure 56 on page 83](#) shows the source code in T1520IC1 and [Figure 57 on page 84](#) shows the source code in T1520IC2.
 - Module T1520IC1 contains an import procedure request named `write_audit_trail`, which matches an export request in T1520SP1, using symbol `write_audit_trail`. The binder matches:
 - The import request from T1520IC1 for the procedure `write_audit_trail` with the corresponding export from T1520SP1
 - The import request from T1520IC2 for the data item `taxrate` with the corresponding export from T1520SP2
 - The `ENTMOD(*ONLY)` parameter specifies that only one module from the list of modules can have a PEP. An error is issued if more than one module is found to have a PEP. If the `ENTMOD(*FIRST)` parameter is used, the first module found from a list of modules that has a PEP is selected as the PEP. All other modules with PEPs are ignored.
 - The default `ACTGRP(*NEW)` parameter is used to define a new activation group. The program T1520PG1 is associated with a new activation group. Service programs T1520SP1 and T1520SP2 were created with activation group `*CALLER`. These service programs become part of the callers activation group using the resources of one activation group for the purposes of developing one cooperative program. Service program T1520SP1 and T1520SP2 are bound to the program being activated. These service programs are also activated as part of the dynamic call processing.
14. Run the program T1520PG1:

Note: Ensure that the library MYLIB is on the LIBL library list.

- a. Enter the command T1520CM1 and press F4 (Prompt).
- b. As you are prompted by T1520CM, type the following data:

```
Hammers
1.98
5000
Nails
0.25
2000
```

The output is:

```
5000 HAMMERS plus tax =          $11,385.00
Press ENTER to end terminal session.
>
2000 NAILS plus tax =           $575.00
Press ENTER to end terminal session.
```

The physical file T1520DD1 is updated with the following data:

```
SMITHE  HAMMERS          0000000198500015          $11,385.00072893
SMITHE  NAILS           0000000025200015           $575.00072893
```

Note: Each entry is identified with the user's ID. In this case, SMITHE was the user.

Source Code Samples

The figures in this section contain the source code used to create the ILE sample application.

Source Code for an Audit Log File

Audit file T1520DD1 is shown in [Figure 51 on page 75](#). The DDS source defines the fields for the audit file.

```
R T1520DD1R
  USER          10          COLHDG('User')
  ITEM          20          COLHDG('Item name')
  PRICE        10S 2       COLHDG('Unit price')
  QTY           4S        COLHDG('Number of items')
  TXRATE       2S 2       COLHDG('Current tax rate')
  TOTAL        21          COLHDG('Total cost')
  DATE         6          COLHDG('Transaction date')
K  USER
```

Figure 53. DDS Source for Audit File T1520DD1

Source Code Pass Terminal Session Input to an ILE Program

CL program T1520CL1 is shown in [Figure 51 on page 75](#). It passes required parameters to the ILE C program T1520PG1.

```
PGM          PARM(&ITEMIN &PRICE &QUANTITY) 1
DCL          VAR(&USER)          TYPE(*CHAR)  LEN(10)
DCL          VAR(&ITEMIN)        TYPE(*CHAR)  LEN(20)
DCL          VAR(&ITEMOUT)       TYPE(*CHAR)  LEN(21)
DCL          VAR(&PRICE)         TYPE(*DEC)   LEN(10 2)
DCL          VAR(&QUANTITY)      TYPE(*DEC)   LEN(2 0)
DCL          VAR(&NULL)          TYPE(*CHAR)  LEN(1)  VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */ 2
CHGVAR      VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
/* GET THE USERID FOR THE AUDIT TRAIL */ 3
RTVJOBA     USER(&USER)
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF      FILE(T1520DD1) TOFILE(*LIBL/T1520DD1) +
            MBR(T1520DD1) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL        PGM(T1520PG1) PARM(&ITEMOUT &PRICE &QUANTITY +
            &USER)
DLTOVR      FILE(*ALL)
ENDPGM
```

Figure 54. T1520CL1 – CL Source to Pass Variables to an ILE C Program

Note:

1. This program passes (by reference) the CL variables for item name, price, quantity, and user ID to an ILE C program T1520PG1. Variables in a CL program are passed by reference to allow an ILE C program to change the contents in the CL program.
2. The variable *ITEMOUT* is null-terminated in the CL program T1520CL1. Passing CL variables passed from CL to ILE C are not automatically null-terminated on a compiled CL call. See [“Using ILE C/C++ Call Conventions” on page 283](#) for information about null-terminated strings for compiled CL calls and command line CL calls.
3. The Retrieve Job Attributes (RTVJOBA) command obtains the user ID for the audit trail.

Source Code to Define a CL Command to Collect Session Data

CL CMD program T1520CM1 is shown in [Figure 51 on page 75](#).

This developer-defined command:

- Prompts the user to enter, in the following order: Item name, Unit price, and Number of items
- Stores the input data in the following keyword parameters: ITEM, PRICE, and QUANTITY.

Note: These keyword parameters were defined in the DDS file [T1520DD1](#).

```
CMD      PROMPT('CALCULATE TOTAL COST')
PARM    KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
        MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
PARM    KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
        RANGE(0.01 99999999.99) MIN(1) +
        ALWUNPRT(*YES) PROMPT('Unit price' 2)
PARM    KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
        9999) MIN(1) ALWUNPRT(*YES) +
        PROMPT('Number of items' 3)
```

Figure 55. T1520CM1 – CL Command Source to Receive Input Data

Source Code for a User Entry Procedure (UEP)

In the source code for T1520IC1, the `main()` function:

- Receives the user ID, item name, quantity, and price from a CL program
- Calls `calc_and_format()` function in module T1520IC2, which:
 - Calculates an item's total cost
 - Calls the `write_audit_trail()` function from module T1520IC2, which writes the transaction to an audit file

```

/* This program demonstrates how to use multiple modules, service */
/* programs and a binding directory. This program accepts a user ID, */
/* item name, quantity, and price, calculates the total cost, and */
/* writes an audit trail of the transaction. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>
int calc_and_format (decimal(10,2),
                    short int,
                    char[]);
void write_audit_trail (char[],
                       char[],
                       decimal(10,2),
                       short int,
                       char[]);
int main(int argc, char *argv[])
{
    /* Incoming arguments from a CL program have been verified by */
    /* the *CMD and null ended within the CL program. */
    char *user_id;
    char *item_name;
    short int *quantity;
    decimal (10,2) *price;
    char formatted_cost[22];
    /* Incoming arguments are all pointers. */
    item_name = argv[1];
    price = (decimal (10, 2) *) argv[2];
    quantity = (short *) argv[3];
    user_id = argv[4];
    /* Call an ILE C function that returns a formatted cost. */
    /* Function calc_and_format returns true if successful. */
    if (calc_and_format (*price, *quantity, formatted_cost))
    {
        write_audit_trail (user_id,
                           item_name,
                           *price,
                           *quantity,
                           formatted_cost);
        printf("\n%d %s plus tax = %-s\n", *quantity,
              item_name,
              formatted_cost);
    }
    else
    {
        printf("Calculation failed\n");
    }
    return 0;
}

```

Figure 56. ILE C Source to Call Functions in Other Modules

Note:

1. The main() function in this module is the user entry procedure (UEP), which is the target of a dynamic program call from the CL program T1520CL1. The UEP receives control from the program entry procedure (PEP). This module has a PEP that is generated by the ILE C compiler during compilation. The PEP is an entry point for the ILE C/C++ program on a dynamic program call from the CL program T1520CL1. The PEP is shown in the call stack as _C_pep.
2. The main() function in this module receives the incoming arguments from the CL program T1520CL1 that are verified by the CL command prompt T1520CM1.
3. All the incoming arguments are pointers. The variable item_name is null terminated within the CL program T1520CL1.
4. The main() function in this module calls calc_and_format in module T1520IC2 to return a formatted cost. If the calc_and_format returns successful a record is written to the audit trail by write_audit_trail in the service program T1520SP1.
5. The function write_audit_trail is not defined in this module (T1520IC1), so it must be imported.

Source Code to Calculate Tax and Format Cost for Output

Module T1520IC2 is shown in [Figure 52 on page 76](#). It provides the calc_and_format() function.

```

/* This function calculates the tax and formats the total cost. */
/* The function calc_and_format() returns 1 if successful and 0 */
/* if it fails. */
#include <stdio.h>
#include <string.h>
#include <decimal.h>
/* Tax rate is imported from the service program T1520SP2. */1
const extern decimal (2,2) taxrate;
int calc_and_format (decimal (10,2) price,
                    short int    quantity,
                    char          formatted_cost[22])
{
    decimal (17,4) hold_result;
    char          hold_formatted_cost[22];
    int          i,j;
    memset(formatted_cost, ' ', 21);
    hold_result = (decimal(4,0))quantity *
    price * (1.00D+taxrate); /* Calculate the total cost. */
    if (hold_result < 0.01D || hold_result > 1989800999801.02D)
    {
        printf("calc out of range:%17.4D(17,4)\n", hold_result);
        return(0);
    }
    /* Format the total cost. */
    sprintf(hold_formatted_cost, "%21.2D(17,4)", hold_result);
    j = 0;
    for (i=0; i<22; ++i)
    {
        if (hold_formatted_cost[i] != ' ' &
            hold_formatted_cost[i] != '0')
        {
            hold_formatted_cost[j] = '$';
            break;
        }
        j = i;
    }
    for (i=j=21; i>=0; --i)
    {
        if (j < 0) return(0);
        if (hold_formatted_cost[i] == '$')
        {
            formatted_cost[j] = hold_formatted_cost[i];
            break;
        }
        if (i<16 & !((i-2)%3))
        {
            formatted_cost[j] = ',';
            --j;
        }
        formatted_cost[j] = hold_formatted_cost[i];
        --j;
    }
    /* End of for loop, 21->0. */
    return(1);
}

```

Figure 57. Sample ILE C Source to Calculate Tax and Format Cost for Output

Note:

1. The function `calc_and_format` in this module calculates and formats the total cost. To do the calculation, the data item `taxrate` is imported from service program `T1520SP2`. This data item must be imported because it is not defined in this module (`T1520IC2`).

Source Code to Write an Audit Trail

The function `write_audit_trail` in module `T1520IC3` writes the audit trail for the ILE C program `T1520PG1`. Module `T1520IC3` is used to create service program `T1520SP1`, shown in [Figure 52 on page 76](#).

Use the following source:

```

/* This function writes an audit trail. To write the audit trail */
/* the file field structure is retrieved from the DDS file */
/* T1520DD1 and the taxrate data item is imported from service */
/* program T1520SP2. Retrieves the file field structure. */

```

```

#pragma mapinc("myinc", "MYLIB/T1520DD1(*all)", "both", "p z","")
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <decimal.h>
#include <recio.h>
#include <xxcvt.h>

/* These includes are for the call to QWCCVTD T API to get */ 1
/* the system date to be used in the audit trail. */
#include <QSYSINC/H/QWCCVTD T>
#include <QSYSINC/H/QUSEC>
/* DDS mapping of the audit file, T1520DD1. */ 2
#include "myinc"
/* Tax rate is imported from service program T1520SP2. */ 3
const extern decimal (2,2) taxrate;
void write_audit_trail (char          user_id[10],
                       char          item_name[],
                       decimal (10,2) price,
                       short int     quantity,
                       char          formatted_cost[22])
{
    char char_item_name[21];
    char char_price[11];
    char temp_char_price[11];
    char char_quantity[4];
    char char_date[6];
    char char_taxrate[2];
    /* Qus_EC_t is defined in QUSEC. */
    Qus_EC_t errcode;
    char get_date[16];
    int i;
    double d;
    /* File field structure is generated by the #pragma */
    /* mapinc directive. */
    MYLIB_T1520DD1_T1520DD1R_both_t buf1;
    _RFILE *fp;
    /* Get the current date. */
    errcode.Bytes_Provided = 0;
    QWCCVTD T ("*CURRENT ", "", "*MDY ", get_date, &errcode);
    memcpy (char_date, &(get_date[1]), 6);
    /* Loop through the item_name and remove the null terminator. */
    for (i=0; i<=20; i++)
    {
        if (item_name[i] == '\0') char_item_name[i] = ' ';
        else char_item_name[i] = item_name[i];
    }
    /* Convert packed to zoned for audit file. */
    d = price;
    QXXD TOZ (char_price, 10, 2, d);
    QXXI TOZ (char_quantity, 4, 0, quantity);
    d = taxrate;
    QXXD TOZ (char_taxrate, 2, 2, d);
    /* Set up buffer for write. */
    memset(&buf1, ' ', sizeof(buf1));
    memcpy(buf1.USER, user_id, 10);
    memcpy(buf1.ITEM, char_item_name, 20);
    memcpy(buf1.PRICE, char_price, 11);
    memcpy(buf1.QTY, char_quantity, 4);
    memcpy(buf1.TXRATE, char_taxrate, 2);
    memcpy(buf1.TOTAL, formatted_cost, 21);
    memcpy(buf1.DATE, char_date, 6);
    if ((fp = _Ropen("MYLIB/T1520DD1", "ar+") == NULL)
    {
        printf("Cannot open audit file\n");
    }
    _Rwrite(fp, (void *)&buf1, sizeof(buf1));
    _Rclose(fp);
}

```

Note:

1. This source requires two members, QUSEC and QWCCVTD T, that are in the QSYSINC/H file. The QSYSINC library is automatically searched for system include files as long as the OPTION(*STDINC) parameter (the default) is specified on the CRTBNDC or CRTCMOD command.
2. The include name myinc is associated with the temporary source member created by the compiler when it generates the type definitions for the #pragma mapinc directive. See [Using Externally](#)

Described Files in a Program” on page 179 for information on how to use the `#pragma mapinc` directive.

3. To write the audit trail, the tax rate is imported from the service program T1520SP2.

Source Code to Export Tax Rate Data

Module T1520IC4 is used to create service program T1520SP2.

```
/* Export the tax rate data.                               */
#include <decimal.h>
const decimal (2,2)  taxrate = .15D;
```

Figure 58. T1520IC4 – ILE C Source to Export Tax Rate Data

Note: Because it is coded in a separate module, the data item `taxrate` can be imported by both or either of the following:

- The `calc_and_format` function in service program T1520IC2
- The `write_audit_trail` in T1520IC3.

Binder Language to Export Tax Rate Data

```
STRPGMEXP PGMLVL(*CURRENT) EXPORT SYMBOL('taxrate')
ENDPGMEXP
```

Figure 59. Binder Language Source to Export Tax Rate Data

Note:

1. The Start Program Export (STRPGMEXP) command identifies the beginning of a list of exports from the service program T1520SP1.
2. The Export Symbol (EXPORT) command identifies the symbol name `taxrate` to be exported from the service program T1520SP1.
3. The symbol name `taxrate`, identified between the STRPGMEXP PGMLVL(*CURRENT) and ENDPGMEXP pair, defines the public interface to the service program T1520SP2. Only those procedures and data items exported from the module objects making up the ILE C service program can be exported from this service program.
4. The symbol name `taxrate` is enclosed in apostrophes to maintain its lowercase format. Without apostrophes it is converted to uppercase characters. (The binder would search for TAXRATE, which it would not find.)
5. The symbol name `taxrate` is also used to create a signature. The signature validates the public interface to the service program T1520SP2 at activation. This ensures that the ILE C service program T1520SP1 and the ILE C program T1520PG1 can use service program T1520SP2 without being re-created.

Binder Language to Export the write-audit-trail Procedure

```
STRPGMEXP PGMLVL(*CURRENT) EXPORT SYMBOL('write_audit_trail')
ENDPGMEXP
```

Figure 60. Binder Language Source to Export write_audit_trail Procedure

Note:

1. The Start Program Export (STRPGMEXP) command identifies the beginning of a list of exports from the service program T1520SP2.
2. The Export Symbol (EXPORT) command identifies the symbol name `write_audit_trail` to be exported from the service program T1520SP2.

3. The symbol name `write_audit_trail`, identified between the `STRPGMEXP PGMLVL(*CURRENT)` and `ENDPGMEXP` pair, defines the public interface to the service program `T1520SP2`. Only those procedures and data items exported from the module objects making up the ILE C service program can be exported from this service program. If you cannot control the public interface, runtime or activation errors may occur.
4. The symbol name `write_audit_trail` is enclosed in apostrophes to maintain its lowercase format. Without apostrophes it is converted to uppercase characters. (The binder would search for `WRITE_AUDIT_TRAIL`, which it would not find.)
5. The symbol name `write_audit_trail` is also used to create a signature. The signature validates the public interface to the service program `T1520SP2` at activation. This ensures that the ILE C service program `T1520SP1` and the ILE C program `T1520PG1` can use service program `T1520SP2` without being re-created.

Debugging Programs

This topic describes how to:

- [Use the ILE source debugger \(its options, language syntax, and commands\)](#)
- [Bind a module with debug data into a program and create a listing view for debugging](#)
- [Prepare and compile your program to include debugging data](#)
- [Use debugging sessions](#)
- [Use breakpoints to aid debugging](#)
- [Use watches to aid debugging](#)
- [Step through a program](#)
- [Debug variables](#)

The ILE Source Debugger

The ILE source debugger helps you locate programming errors in ILE C/C++ programs and service programs.

Before you can use the ILE source debugger, you must use one of the non-default debug options (DBGVIEW) when you compile a source file. Once you set breakpoints or other ILE source debugger options, you can start your debug session and call the program.

This topic describes:

- [Debug data options](#)
- [Debug language syntax and its limitations](#)
- [Debug commands](#)
- [Examples of data definitions to illustrate what can be done with the ILE source debugger and ILE C applications](#)

Debug Data Options

The type of debug data that can be associated with a module is referred to as a debug view.

The storage requirements for a module or program vary somewhat, depending on the type of debug data included. The debug options are listed below. Secondary storage requirements increase as you work down the list:

1. DBGVIEW(*NONE) (No debug data)
2. DBGVIEW(*STMT) (Statement view)
3. DBGVIEW(*SOURCE) (Source view)
4. DBGVIEW(*LIST) (Listing view)
5. DBGVIEW(*ALL) (All views)

Debug Language Syntax

Limitations of the C debug expression grammar include:

- **Type casts:** Array and function designator type casts are prohibited.
- **Function calls:** Function calls cannot be used in debug expressions.
- **Decimal types:** Decimal types are supported for display only. They cannot be used in debug expressions.

Precedence of operators and type conversion of mixed types conforms to the ISO C standard. For more information, see the *ILE C/C++ Language Reference*.

Limitations of the Debug Language Syntax

The ILE source debugger has the following limitations:

- Function calls cannot be used in debug expressions. This is a limitation of the debug expression grammar.
- Precedence of operators and type conversion of mixed types conform to the C and C++ language standards.
- The maximum size of variables that can be displayed is 65535 characters:
 - With the `:c` and `:x` formatting overrides, if no count is entered, the command stops after one byte.
 - With the `:s` formatting override, if no count is entered, the command stops after 30 bytes or a NULL, whichever is encountered first.
 - With the `:f` formatting override, if no count is entered, the command stops after 1024 bytes or a NULL, whichever is encountered first.
- The maximum number of classes that can be inherited as virtual base classes in a single compilation unit is 512.

Debug Commands

Many debug commands are available for use with the ILE source debugger.

For example, if you enter `break 10` on the debug command line (and line 10 is a debuggable statement), the debugger adds an unconditional breakpoint to line 10 of your source.

Note:

1. If line 10 is a blank line or a comment statement, the debugger will give an error.
2. If line 10 is not a debuggable statement, such as a typedef statement, it will set the break point to the very next debuggable statement.
3. Pressing the F6 key (while the cursor is on a debuggable command line) sets or clears a break point.

Debug data is created when you compile a module with one of the following debug options:

- `*STMT`
- `*SOURCE`
- `*LIST`
- `*ALL`

The debug commands and their parameters are entered on the Debug command line shown at the bottom of the Display Module Source display or the Evaluate Expression display. They can be entered in uppercase, lowercase, or mixed case.

The online information describes the debug commands, and shows their allowed abbreviations.

The debug commands are as follows:

ATTR

Displays the attributes of variables. The attributes are the size and type of the variable as recorded in the Debug Symbol table.

BREAK

Permits you to enter either an unconditional or conditional job breakpoint at a position in the program being tested. To enter a conditional job breakpoint, enter `BREAK line-number WHEN expression`.

CLEAR

Removes conditional and unconditional breakpoints; removes one or all active watch conditions.

DISPLAY

Displays the names and definitions assigned by using the Equate command. It also allows you to display a different source module than the one that is currently shown on the Display Module Source display. The module object must exist in the current program object.

EQUATE

Assigns an expression, variable, or debug command to a name for shorthand use.

EVAL

Displays or changes the value of a variable; displays the value of expressions, records, structures, or arrays.

QUAL

Defines the scope of variables that appear in subsequent EVAL or WATCH commands.

SET

Changes debug options, such as the ability to update production files; specifies whether find operations are to be case sensitive; enables OPM source debug support.

STEP

Runs one or more statements of the procedure that is being debugged.

TBREAK

Permits you to enter either an unconditional or conditional breakpoint in the current thread in a position, in the program being tested.

THREAD

Allows you to either open the Work with Debugged Threads display or change the current thread.

WATCH

Requests a breakpoint when the contents of a specified storage location is changed from its current value.

FIND

Searches in the module that is currently displayed for a specified line number or string of text. The text search can be specified in a forward or backward direction from the position of the cursor on the displayed view text. If the cursor is not on the view text, the search starts at the first position of the top line of text on the current screen. When the string is successfully found, the cursor will be positioned on the first character of the found string.

The last Find command that is entered can be repeated by using the F16 Repeat Find key.

UP

Moves the displayed window of source towards the beginning of the view by the number of lines entered.

DOWN

Moves the displayed window of source towards the end of the view by the number of lines entered.

LEFT

Moves the displayed window of source to the left by the number of characters that are entered.

RIGHT

Moves the displayed window of source to the right by the number of characters that are entered.

TOP

Positions the view to show the first line.

BOTTOM

Positions the view to show the last line.

NEXT

Positions the view to the next breakpoint in the source that is currently displayed.

PREVIOUS

Positions the view to the previous breakpoint in the source that is currently displayed.

HELP

Shows the online help information for the available source debugger commands.

Examples of Using Debug Expressions in ILE C Programs

Examples of ILE C source code and the ILE debug expressions that can be used to diagnose and correct programming errors are provided.

In the examples:

- The ILE C source code to be evaluated is presented in figures.
- Information shown in an ILE Source Debugger session is presented in screen figures. Each screen shows the EVAL debug command and the information it retrieves and displays.

Examples of Program Definitions and Corresponding Debug Expressions

The following ILE C source code includes data definitions of pointers, simple variables, structures, unions, and enumerations. To illustrate what can be done with the ILE source debugger and ILE C applications, the corresponding debug expressions and information are shown in the following sections:

- [“Evaluating Pointers to Find and Correct Errors” on page 93](#)
- [“Evaluating Simple Expression to Find and Correct Errors” on page 94](#)
- [“Evaluating Bit Fields to Find and Correct Errors” on page 95](#)
- [“Evaluating Structures and Unions to Find and Correct Errors” on page 95](#)
- [“Evaluating Enumerations to Find and Correct Errors” on page 96](#)

```
#include <stdio.h>
#include <decimal.h>
#include <pointer.h>
/** POINTERS **/
_SYSPTR pSys;          /* System pointer */
_SPCPTR pSpace;        /* Space pointer */
int (*fnctr)(void);    /* Function pointer */
char *pc1;             /* Character pointer*/
char *pc2;             /* Character pointer*/
int *pi1;              /* Integer pointer */
char arr1[] = "ABC";   /* Array */
/** SIMPLE VARIABLES **/
int i1;                /* Integer */
unsigned u1;           /* Unsigned Integer */
char c1;               /* Character */
float f1;              /* Float */
_Decimal(3,1) dec1;    /* Decimal */
/** STRUCTURES **/
struct {               /* Bit fields */
    int b1 : 1;
    int b4 : 4;
} bits;
struct x{              /* Tagged structure */
    int x;
    char *p;
};
struct y{              /* Structure with */
    int y;              /* structure member */
    struct x x;
};
typedef struct z {     /* Structure typedef*/
    int z;
    char *p;
} z;
z zz;                  /* Structure using typedef */
z *pZZ;

/* Same */
typedef struct c {     /* Structure typedef */
    unsigned a;
    char *b;
} c;
c d;                    /* Structure using typedef */
/** UNIONS **/
union u{              /* Union */
    int x;
    unsigned y;
};
union u u;             /* Variable using union */
```

```

union u *pU;          /* Same */
/** ENUMERATIONS **/
enum number {one, two, three};
enum color {red,yellow,blue};
enum number number = one;
enum color color = blue;
/** FUNCTION **/
int ret100(void) { return 100;}
main(){
    struct y y, *pY;
    bits.b1 = 1;
    bits.b4 = 2;
    i1 = ret100();
    c1 = 'C';
    f1 = 100e2;
    dec1 = 12.3;
    pc1 = &c1;
    pi1 = &i1;
    d.a = 1;
    pZZ = &zz;
    pZZ->z=1;
    pY = &y;
    pY->x.p=(char*)&y;
    pU = &u;
    pU->x=255;
    number=color;
    fncptr = &ret100;
    pY->x.x=1;          /* Set breakpoint here */
}
int main(void) { ... . . .
return(0); }

```

Evaluating Pointers to Find and Correct Errors

A pointer type variable holds the address of a data object or a function. A pointer can refer to an object of any one data type (except a bit field or a reference). A pointer is classified as a scalar type, which means that it can hold only one value at a time.

For information about using pointers in your programs, see:

- [“Using pointers in a Program” on page 276](#)
- [“Using Pointers to Improve Performance” on page 66](#)
- *ILE C/C++ Language Reference*

Use the EVAL debug command to display or change the value of a pointer variable or array. Messages with multiple line responses launch the Evaluate Expression display, which also shows a history of the debug commands entered and the results from these commands. To return to the Display Module Source display, press the Enter key. You can use the Enter key as a toggle switch between displays.

Note: Single-line responses are shown on the Display Module Source message line.

The following figures show examples of debug expressions for pointers.

Evaluate Expression	
Previous debug expressions	
>eval pc1	
pc1 = SPP:C0260900107C0000	Displaying pointers
>eval pc2=pc1	
pc2=pc1 = SPP:C0260900107C0000	Assigning pointers
>eval *pc1	
*pc1 = 'C'	Dereferencing pointers
>eval &pc1	
&pc1 = SPP:C026090010400000	Taking an address
>eval *&pc1	
*&pc1 = SPP:C0260900107C0000	Can build expressions with normal C precedence
>eval *(short *)pc1	
*(short *)pc1 = -15616	Casting
Bottom	
Debug . . . -----	
F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right	

Figure 61. Examples of Using Pointers in Debug Sessions, Screen 1

```

                                Evaluate Expression
Previous debug expressions
>eval arr1
arr1 = SPP:C026090010700000      Unqualified arrays are treated
                                as pointers
>eval *arr1
*arr1 = 'A'                      Dereferencing applies the array type
                                (character in this example)
>eval *arr1:s
*arr1:s = "ABC"                 If the expression is an lvalue
                                you can override the formatting
>eval pc1=0
pc1=0 = SYP:*NULL              Setting a pointer to null by assigning 0
>eval fncptr
fncptr = PRP:A0CD0004F0100000   Function pointers
>eval *pY->x.p
*pY->x.p = ' '                  Using the arrow operator

                                Bottom
Debug . . . -----
F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right

```

Figure 62. Examples of Using Pointers in Debug Sessions, Screen 2

Evaluating Simple Expression to Find and Correct Errors

Expressions are sequences of operators, operands, and punctuators that specify a computation. The evaluation of expressions is based on the operators that the expressions contain and the context in which they are used.

For information about using expressions and operators in your programs, see:

- [“Using pointers in a Program” on page 276](#)
- [“Using BCD Macros to Port Coded Decimal Objects to ILE C++” on page 298](#)
- [“Using Packed Decimal Data in a C Program” on page 354](#)
- [“Using Packed Decimal Data in a C++ Program” on page 365](#)
- *ILE C/C++ Language Reference*

Use the EVAL debug command to display or change the value of a simple expression or operator. Messages with multiple line responses launch the Evaluate Expression display, which also shows a history of the debug commands entered and the results from these commands. To return to the Display Module Source display, press the Enter key. You can use the Enter key as a toggle switch between displays.

Note: Single-line responses are shown on the Display Module Source message line.

This following figure shows examples of debug expressions for simple operations (for example, assignment, arithmetic, or relational operations).

```

                                Evaluate Expression
Previous debug expressions
>eval i1==u1 || i1<u1
i1==u1 || i1<u1 = 0             Logical operations
>eval i1++
i1++ = 100                     Unary operators occur in proper order
>eval i1
i1 = 101                       Increment has happened after i1 was used
>eval ++i1
++i1 = 102                     Increment has happened before i1 was used
>eval u1 = -10
u1 = -10 = 4294967286          Implicit conversions happen
>eval (int)u1
(int)u1 = -10
>eval dec1
dec1 = 12.3                    Decimal types are displayed but cannot
                                be used in expressions

                                Bottom
Debug . . . -----
F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right

```

Figure 63. Examples of Simple Operations Used in Debug Expressions

Evaluating Bit Fields to Find and Correct Errors

Both C and C++ allow integer members to be stored into memory spaces smaller than the compiler would ordinarily allow. These space-saving structure members are called bit fields, and their width in bits can be explicitly declared. Bit fields are used in programs that must force a data structure to correspond to a fixed hardware representation and are unlikely to be portable.

For information about using bit fields, see:

- [“Choosing Data Types to Improve Performance”](#) on page 57
- *ILE C/C++ Language Reference*

Use the EVAL debug command to display or change the value of a bit field. Messages with multiple line responses launch the Evaluate Expression display, which also shows a history of the debug commands entered and the results from these commands. To return to the Display Module Source display, press the Enter key. You can use the Enter key as a toggle switch between displays.

Note: Single-line responses are shown on the Display Module Source message line.

The following figure shows examples of debug expressions for bit fields.

```

                                Evaluate Expression

Previous debug expressions

>eval bits                        You can display an entire structure
  bits.b1 = 1
  bits.b4 = 2
>eval bits.b4 = bits.b1           You can work with a single member
  bits.b4 = bits.b1 = 1
>eval bits.b1 << 2                Bit fields are fully supported
  bits.b1 << 2 = 4
>eval bits.b1 = bits.b1 << 2     You can overflow, but no warning is
  bits.b1 = bits.b1 << 2 = 4     generated
>eval bits.b1
  bits.b1 = 0


                                Bottom
Debug . . . -----
F3=Exit  F9=Retrieve  F12=Cancel  F18=Command entry  F19=Left  F20=Right
```

Figure 64. Examples of Using Bit Fields in Debug Expressions

Evaluating Structures and Unions to Find and Correct Errors

A *structure* is a class declared with the class_key struct. The members and base classes of a structure are public by default.

A *union* is a class declared with the class_key union. The members of a union are public by default; a union holds only one data member at a time.

No conversions between structure or union types are allowed, except for the following:  In C, an assignment conversion between compatible structure or union types is allowed if the right operand is of a type compatible with that of the left operand.

For information about using structures and unions in your programs, see:

- [“Porting Programs from Another Platform to ILE”](#) on page 295
- [“Using BCD Macros to Port Coded Decimal Objects to ILE C++”](#) on page 298
- [“Using Externally Described Files in a Program”](#) on page 179
- [“Interlanguage Data-Type Compatibilities”](#) on page 425
- *ILE C/C++ Language Reference*

Use the EVAL debug command to display or change the value of a structure or union. Messages with multiple line responses launch the Evaluate Expression display, which also shows a history of the debug commands entered and the results from these commands. To return to the Display Module Source display, press the Enter key. You can use the Enter key as a toggle switch between displays.

Note: Single-line responses are shown on the Display Module Source message line.

The following shows examples of debug expressions for structures and unions.

```

Evaluate Expression

Previous debug expressions

>eval (struct z *)&zz
(struct z *)&zz = SPP:C005AA0010D000000 You can cast with typedefs
>eval *(c *)&zz
*(c *)&zz.a = 1           You can cast with tags
*(c *)&zz.b = SYP:*NULL

>eval u.x = -10
u.x = -10 = -10         You can assign union members
>eval u
u.y = 4294967286        You can display and the union will be
u.x = -10                formatted for each definition

Debug . . . Bottom
-----
F3=Exit  F9=Retrieve  F12=Cancel  F18=Command entry  F19=Left  F20=Right

```

Figure 65. Examples of Using Structures and Unions in Debug Expressions

Evaluating Enumerations to Find and Correct Errors

An *enumeration* is a data type consisting of a set of values that are named integral constants. It is also referred to as an enumerated type because you must list (enumerate) each of the values in creating a name for each of them. A named value in an enumeration is called an enumeration constant. In addition to providing a way of defining and grouping sets of integral constants, enumerations are useful for variables that have a small number of possible values. For information about using enumerations in your programs, see:

- [“Interlanguage Data-Type Compatibilities” on page 425](#)
- *ILE C/C++ Language Reference*

The following figure shows debug expressions for enumerations.

Evaluate Expression

Previous debug expressions

>eval color color = blue (2)	Both the enumeration and its value are displayed
>eval number number = three (2)	
>eval (enum color)number (enum color)number = blue (2)	Casting to a different enumeration
>eval number = 1 number = 1 = two (1)	Assigning by number
>eval number = three number = three = three (2)	Assigning by enumeration
>eval arr1[one] arr1[one] = 'A'	Using in an expression

Bottom

Debug . . .

F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right

Figure 66. Examples of Using Enumerations in Debug Expressions

Examples of Displaying System and Space Pointers in the ILE Source Debugger

The ILE C source code in [Figure 67](#) on page 98 sets up system and space pointers for an example of how they can be displayed with the debugger. The corresponding debug expressions and information are shown in [Figure 68](#) on page 98

```

#include <stdio.h>
#include <mispace.h>
#include <pointer.h>
#include <miscobj.h>
#include <except.h>
#include <lecond.h>
#include <leenv.h>
#include <qtedbgs.h> /* From qsysinc */
/* Link up the Create User Space API */
#pragma linkage(CreateUserSpace,OS)
#pragma map(CreateUserSpace,"QUSCRTUS")
void CreateUserSpace(char[20],
                    char[10],
                    long int,
                    char,
                    char[10],
                    char[50],
                    char[10],
                    _TE_ERROR_CODE_T *
                    );
/* Link up the Delete User Space API */
#pragma linkage(DeleteUserSpace,OS)
#pragma map(DeleteUserSpace,"QUSDLTUS")
void DeleteUserSpace(char[20],
                    _TE_ERROR_CODE_T *
                    );
/* Link up the Retrieve Pointer to User Space API */
#pragma linkage(RetrievePointerToUserSpace,OS)
#pragma map(RetrievePointerToUserSpace,"QUSPTRUS")
void RetrievePointerToUserSpace(char[20],
                                char **,
                                _TE_ERROR_CODE_T *
                                );
int main (int argc, char *argv[])
{
    char *pBuffer;
    _SYSPTR pSYSptr;
    _TE_ERROR_CODE_T errorCode;
    errorCode.BytesProvided = 0;
    CreateUserSpace("QTEUSERSPCQTEMP ",
                  "QTESSPC ",
                  10,
                  0,
                  "*ALL ",
                  " ",
                  "*YES ",
                  &errorCode
                  );

    /* call RetrievePointerToUserSpace - Retrieve Pointer to User Space */
    /*! (pass: Name and library of user space, pointer variable */
    /*! return: nothing (pointer variable is left pointing to start*/
    /*! of user space) */
    RetrievePointerToUserSpace("QTEUSERSPCQTEMP ",
                              &pBuffer,
                              &errorCode);

    /* convert the space pointer to a system pointer */
    pSYSptr = _SETSPFP(pBuffer);

    printf("Space pointer: %p\n",pBuffer);
    printf("System pointer: %p\n",pSYSptr);

    return 0;
}

```

Figure 67. System and Space Pointers in ILE C Source Code

The following figure illustrates how the debugger displays system and space pointers.

Evaluate Expression	
Previous debug expressions	
>eval pSYSptr	System pointers are formatted
pSYSptr =	SYP:QTEUSERSPC :1934:QTEMP :111111110
	00111100
>eval pBuffer	
pBuffer =	SPP:071ECD0002000000 Space pointers return 6 bytes that can be used in System Service Tools
Debug . . .	Bottom

F3=Exit F9=Retrieve F12=Cancel F18=Command entry F19=Left F20=Right	

Figure 68. Example of System and Space Pointer Display

Preparing a Program for Debugging

This topic describes how to:

- Set up a test library to avoid modification of production files.

- Bind a module with debug data into a program and create a listing view for debugging.

Before you can use the ILE source debugger, the program has to contain debug data. Then you can start your debug session. After you set breakpoints or other ILE source debugger options, you can call the program.

Note: For information about setting breakpoints and other ILE source debugger options, see [“Using Breakpoints to Aid Debugging”](#) on page 105.

Setting Up a Test Library



Attention: While debugging and testing your programs, ensure that your library list is changed to direct the programs to a test library containing test objects so that any existing production data is not affected.

To prevent production libraries from being modified unintentionally, do one of the following:



- Use Start Debug (STRDBG) with the `UPDPROD` parameter set to `*NO` (the default).
- Use Change Debug (CHGDBG) with the `UPDPROD` parameter set to `*NO` (the default).
- Specify `SET UPDPROD NO` in the Display Module Source display.

Note: If you start a debug session with the `UPDPROD` parameter set to `(*YES)`, the debug session will be able to access objects in production libraries.

Creating a Listing View for Debugging

A *listing view* contains text similar to the text in the compiler listing produced by the compiler. To create a listing view for debugging, use the `DBGVIEW(*LIST)` option when you create the module or program.

Note:

1.  The ILE C compiler creates the listing view by copying the lines in a section of the listing into the module.
2.  The ILE C++ compiler creates the listing view by copying the text of the appropriate source files into the module.

The listing view is not linked to the source files upon which it is based.

For example, to create a listing view to debug a program created from the source file `myfile.cpp`, enter:

```
CRTBNDCPP MYFILE SRCSTMF('/home/myfile.cpp') DBGVIEW(*LIST)
```

Note: The maximum line length for a listing view is 255 characters.

Working with Source Debug Sessions

Use the ILE source debugger to locate programming errors in ILE C/C++ programs and service programs.

Before you can use the ILE source debugger, you must use one of the non-default debug options (`DBGVIEW`) when you compile a source file. Next, you can start your debug session. Once you set breakpoints or other ILE source debugger options, you can call the program.

This topic describes how to:

- Start a debug session
- Add and remove programs from a debug session
- Set and change debug options
- View the program source from a debug session

Starting a Source Debug Session

You use the Start Debug (STRDBG) command to start the ILE source debugger. Once the debugger is started, it remains active until you enter the End Debug (ENDDDBG) command.

Note: You must have *USE object authority to use the STRDBG command and *CHANGE authority for the objects that are to be debugged.

Initially you can add as many as 20 programs to a debug session by using the Program (PGM) parameter on the STRDBG command. You can add any combination of OPM or ILE programs. Whether you can use the ILE source debugger to debug all of them depends on

- how the OPM programs were compiled
- the debug environment settings

You can also add as many as 20 service programs to a debug session by using the Service Program (SRVPGM) parameter on the STRDBG command.

Before you can use the ILE source debugger to debug an ILE C/C++ program or service program, a valid debug view must be specified when the module or program is created. Valid debug views include: *SOURCE, *LIST, *STMT, or *ALL.

You can create several views for each module that you want to debug. They are:

- Root source view

A *root source view* contains text from the root source file. This view does not contain any include file expansions.

You can create a root source view by using the *SOURCE or *ALL options on the DBGVIEW parameter for either the CRTCMOD/CRTCPMOD or CRTBNDC/CRTBNDCPP command when you create the module or the program, respectively.

The ILE C/C++ compiler creates the root source view while the module object (*MODULE) is being compiled. The root source view is created using references to locations of text in the root source file rather than copying the text of the file into the view. For this reason, do not modify, rename, or move root source files between the module creation of these files and the debugging of the module created from these files.

- Statement view. This is the view obtained when the source is compiled with *STMF option on the DBGVIEW parameter on either C or C++ command. This will allow to debug programs with statement numbers and symbolic identifiers which can be referenced in the listing.

You can create a statement view by using the *STMF option on the DBGVIEW parameter on either the CRTCMOD/CRTCPMOD or CRTBNDC/CRTBNDCPP command when you create the module or the program, respectively.

- Include source view

An *include source view* contains text from the root source file, as well as the text of all included files that are expanded into the text of the source. This view does not contain any macro expansion. When you use this view, you can debug the root source file and all included files.

You can create an include source view to debug a module by using the *SOURCE or *ALL option on the DBGVIEW parameter, and *SHOWINC on the OPTION parameter.

- Listing view

A *listing view* contains text similar to the compiler listing produced by the ILE C/C++ compiler specified in the OUTPUT() command parameter.

You can create a listing view to debug a module by using the *LIST or *ALL options when you compile the module. You can also specify at least one of *SHOWINC, *SHOWUSR, *SHOWSYS, and *SHOWSKP on the OPTION parameter, depending on the listing view that you want to see.

Note:  *SHOWSKP is valid for ILE C only.

The first program specified on the STRDBG command is shown when it has debug data. In the case of ILE program, the entry module is shown when it has debug data; otherwise, the first module bound to the ILE program with debug data is shown. To debug an OPM program with ILE source debugger, the following conditions must be met:

- If the program is an OPM RPG or COBOL program, it was compiled with OPTION(*LSTDBG).
- If the program is an OPM CL program, it was compiled with OPTION(*SRCDBG).
- The ILE debug environment is set to accept OPM programs. You can do this by specifying OPMSRC(*YES) on the STRDBG command. (The system default is OPMSRC(*NO).)

If these two conditions are not met, then debug the OPM program with the OPM system debugger.

To start a debug session for the sample debug program DEBUGEX which calls the OPM program RPGPGM, enter:

```
STRDBG PGM(MYLIB/DEBUGEX MYLIB/RPGPGM) OPMSRC(*YES)
```

DBGVIEW(*NONE) is the default DBGVIEW option. No debug data is created when the module is created.

Once you have created a module with debug data or debug views, and bound it into a program object (*PGM), you can start to debug your program.

Example:

This example shows you how to create three modules with debug views and start a debug session.

1. To create module T1520IC1 with a root source view, enter:

```
CRTCMOD MODULE(MYLIB/T1520IC1) SRCFILE(QCPPL/QACSRC) DBGVIEW(*SOURCE)
```

A root source view and debug data is created to debug module T1520IC1.

2. To create module T1520IC2 with all three views, enter:

```
CRTCMOD MODULE(MYLIB/T1520IC2) SRCFILE(QCPPL/QACSRC) DBGVIEW(*ALL)  
OPTION(*SHOWINC)
```

All views and debug data are created to debug module T1520IC2.

3. To create module T1520IC3 with both root source and include view, enter:

```
CRTCMOD MODULE(MYLIB/T1520IC3) SRCFILE(QCPPL/QACSRC) DBGVIEW(*SOURCE)  
OPTION(*SHOWINC)
```

An include view containing the root source file, user include files, and debug data is created to debug module T1520IC3.

4. To create program T1520PG1, enter:

```
CRTPGM PGM(MYLIB/T1520PG1) MODULE(MYLIB/T1520IC1 MYLIB/T1520IC2) ENTMOD(*ONLY)  
BNDDIR(MYLIB/T1520BD1) DETAIL(*FULL)
```

Note: The creation of this program requires modules, service programs, and a binding directory. See [“Creating a Program in Two Steps” on page 15](#).

5. To start a debug session for program T1520PG1, enter:

```
STRDBG PGM(MYLIB/T1520PG1)
```

The Display Module Source display appears as shown:

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC1
 1  /* This program demonstrates how to use multiple modules, service */
 2  /* programs and a binding directory. This program accepts user ID, */
 3  /* item name, quantity and price, calculates the total cost and */
 4  /* writes an audit trail of the transaction.                        */
 5
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include <string.h>
 9  #include <decimal.h>
10      11 int  calc_and_format (decimal(10,2),
11                                     short int,
12                                     char[]);
13
14 void write_audit_trail (char[],
15
Debug . . . -----
-----
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch  F24=More keys

```

The module T1520IC1 is shown. It is the module with the main() function.

You can start a debug session for OPM programs or a combination of ILE and OPM programs by typing:

```
STRDBG PGM(MYLIB/T1520RP1 MYLIB/T1520CB1 MYLIB/T1520IC5) DSPMODSRC(*YES)
```

The parameter DSPMODSRC(*YES) specifies that you want the ILE source debug program display panel to be shown at start debug time. The DSPMODSRC parameter accepts the *PGMDEP value as a default. This value indicates that if any program in the STRDBG PGM list is an ILE program, the source display panel is shown.

Adding and Removing Programs from a Debug Session

If you have *CHANGE authority, you can add programs and service programs to a debug session, or remove them from a current debug session.

For ILE programs, use option 1 (Add program) on the Work with Module List display (F14) of the DSPMODSRC command. To remove an ILE program or service program, use option 4 (Remove program) on the same display. When an ILE program or service program is removed, all breakpoints for that program are removed. There is no limit to the number of ILE programs or service programs that can be included in a debug session at one time.

For OPM programs, you have two choices depending on the value specified for OPMSRC. If you specified OPMSRC(*YES), by using either STRDBG, the SET debug command, or Change Debug (CHGDBG) options, then you add or remove an OPM program using the Work with Module Display. (Note that there will not be a module name listed for an OPM program.) There is no limit to the number of OPM programs that can be included in a debug session when OPMSRC(*YES) is specified.

If you specified OPMSRC(*NO), then you must use the Add Program (ADDPGM) command or the Remove Program (RMVPGM) command. Only 20 OPM programs can be in a debug session at one time when OPMSRC(*NO) is specified.

Note: You cannot debug an OPM program with debug data from both an ILE and an OPM debug session. If OPM program is already in an OPM debug session, you must first remove it from that session before adding it to the ILE debug session or stepping into it from a call statement. Similarly, if you want to debug it from an OPM debug session, you must first remove it from an ILE debug session.

Example:

This example shows you how to add an ILE C service program to, and remove an ILE C program from a debug session.

Note: Assume the ILE C program T1520ALP is part of this debug session, and the program has been debugged. It can be removed from this debug session.

1. To add programs to or remove programs from a debug session type:


```
DSPMODSRC
```

and press Enter. The Display Module Source display appears.

2. Press F14 (Work with module list) to show the Work with Module List display.
3. On this display type 1 (Add program) on the first line of the list to add programs and service programs to a debug session. To add service program T1520SP1, type T1520SP1 for the *Program/module* field, MYLIB for the *Library* field, change the default program type from *PGM to *SRVPGM and press Enter.
4. On this display type 4 (Remove program) on the line next to each program or service program that you want to remove from the debug session.
5. To remove program T1520ALP, type 4 next to T1520ALP, and press Enter.
6. Press F12 (Cancel) to return to the Display Module Source display.

If an ILE C/C++ program with debug data is in a debug session, the module with the `main()` function is shown (if it has a debug view). Otherwise, the first module bound to the ILE C/C++ program with debug data is shown.

Setting or Changing Debug Options During a Session

After you start a debug session, you can set or change the following debug options:

- Whether database files can be updated while debugging your program. (This option corresponds to the UPDPROD parameter of the STRDBG command.)
- Whether text searches using FIND are case sensitive.
- Whether OPM programs are to be debugged using the ILE source debugger. (This corresponds to the OPMSRC parameter.)

Changing the debug options by using the SET debug command affects the value for the corresponding parameter, if any, specified on the STRDBG command. You can also use the Change Debug (CHGDBG) command to set debug options.

Example: Adding an OPM Program to an ILE Debug Session

This example shows you how to allow the ILE source debugger to add an OPM program to an ILE debug session.

Suppose you are in a debug session working with an ILE program and you decide you should also debug an OPM program that has debug data available. To enable the ILE source debugger to accept OPM programs, follow these steps:

1. If, after you enter STRDBG, the current display is not the Display Module Source display, enter:

```
DSPMODSRC
```

The Display Module Source display appears

2. Enter

```
SET
```

3. When the Set Debug Options display appears, type Y (Yes) for the OPM source debug support field, and press Enter to return to the Display Module Source display.

You can now add the OPM program, either by using the Work with Module display, or by processing a call statement to that program.

Example: Setting Debug Options during a Debug Session

This example shows how to set the Update production files debug option during a debug session.

1. To set debug options from a debug session, enter:

```
DSPMODSRC
```

2. When the Display Module Source display appears, enter

```
SET
```

3. When the Set Debug Options display appears, type Y (Yes) for the *Update production files* field, and press Enter to return to the Display Module Source display. The database files in production libraries are updated while the job is in debug mode.

Viewing the Program Source

The Display Module Source display shows the source of an ILE program object one module at a time. The source of an ILE module object can be shown if the module object was compiled using one of the following debug view options:

- DBGVIEW(*SOURCE)
- DBGVIEW(*COPY) - ILE RPG only
- DBGVIEW(*LIST)
- DBGVIEW(*ALL)

The source of an OPM program can be shown if the following conditions are met:

1. The OPM program was compiled with OPTION(*LSTDBG).
2. The ILE debug environment is set to accept OPM programs; that is the value of OPMSRC is *YES. (The system default is OPMSRC(*NO).)

Once you have displayed a view of a module, you may want to display a different module or see a different view of the same module (if you created the module with several different views). The ILE source debugger remembers the last position in which the module is displayed, and displays it in the same position when a module is redisplayed. Lines that have breakpoints set are highlighted. When a breakpoint, step, or message causes the program to stop and the display to be shown, the statement where the breakpoint occurred is highlighted.

Displaying Other Modules in Your Program

You may want to set some debug options in other modules of your program. You can do this by changing the module that is shown on the Display Module Source display to specify the preferred module.

You can change the module that is shown on the Display Module Source display by using:

- The Work with Module list display
- The Display Module debug command

If you use the Display Module debug command with an ILE program object, the entry module with a root source, COPY, or listing view is shown (if it exists). Otherwise the first module object bound to the program object with debug data is shown. If you use this option with an OPM program object, then the source or listing view is shown (if available).

Example: Changing the Module Displayed in a Session

This example shows you how to change from the module shown on the Display Module Source display to another module in the same program using Display Module debug command.

1. While in a debug session, enter DSPMODSRC. The Display Module Source display is shown.
2. On the debug command line, enter: `display module T1520IC2`

The module T1520IC2 is displayed.

Displaying a Different View Of a Module

Several different views of a module are available depending on the values you specify when you create the module. They are:

- Root source view
- Include source view
- Listing view

Example:

This example shows you how to change the view of the module shown on the Display Module Source display.

1. To change the view of the module on the Display Module Source display type DSPMODSRC, and press Enter. The Display Module Source display is shown.
2. Press F15 (Select view). The Select View window is as shown:

```

                Display Module Source
.....
                Select View
:
: Current View . . . : ILE C root source view
:
: Type option, press Enter.
:   1=Select
:
: Opt   View
:  1      ILE C root source view
:  1      ILE C include view
:
:
:                                     Bottom
:
: F12=Cancel
:
:
:
:
:
:                                     More...
:
: Debug . . . -----
: F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
: F12=Resume      F17=Watch variable       F18=Work with watch F24=More keys

```

The current view is listed at the top of the window, and the other views that are available are shown below. Each module in a program can have a different set of views available, depending on the debug options used to create it.

3. Enter 1 next to the ILE C Include view. The Display Module Source display appears showing the module with an include source view. The source of the include view will be shown at a statement position that is equivalent to the statement position in the current view.

Using Breakpoints to Aid Debugging

The flow of a program can be controlled with breakpoints.

When a breakpoint stops the program, the Display Module Source display appears. Use this display to evaluate variables, set more breakpoints, and run any of the source debugger commands. The appropriate module is shown with the source positioned to the line where the condition occurred. The cursor will be positioned on the line where the breakpoint occurred if the cursor was in the text area of the display the last time the source was displayed. Otherwise, it is positioned on the debug command line.

If you change the view of the module after setting breakpoints, then the line numbers of the breakpoints are mapped to the new view by the source debugger.

This topic describes:

- [Breakpoint types](#)
- [How to set conditional and unconditional breakpoints in an unthreaded program](#)

- [How to set conditional thread breakpoints](#)
- [How to test breakpoints](#)
- [How to remove all breakpoints](#)

Types Of Breakpoints

The type of breakpoint determines the scope of the flow that it controls.

Job and Thread Breakpoints

There are two types of breakpoints: job and thread.

- Typically, you use breakpoints to halt processing of a program, or job.
- Each *thread* in a threaded application may have its own thread breakpoint.

Both job and thread breakpoints can be either unconditional or conditional.

In general, there is one set of debug commands and function keys for job breakpoints and another for thread breakpoints.

For the rest of this section on breakpoints, the term *breakpoint* refers to both job and thread, unless specifically mentioned otherwise.

Conditional and Unconditional Breakpoints

You can set unconditional and conditional breakpoints. An *unconditional breakpoint* stops the program at a specific statement. A *conditional breakpoint* stops the program when a specific condition at a specific statement is met.

Setting Breakpoints

To work with a module you can: use either of the following:

- F13 (Work with module breakpoints)
- F6 (Add/Clear breakpoint)

You can set conditional and unconditional breakpoints by using the BREAK debug command.

Note: You can also add breakpoints with the BREAK or TBREAK debug commands. For information on using the BREAK command, see [“Setting Unconditional Breakpoints from the Command Line”](#) on page 107. For information on using the TBREAK command, see [“Setting Conditional Thread Breakpoints”](#) on page 108.

You can remove conditional and unconditional breakpoints by using the CLEAR debug command.

Setting Unconditional Breakpoints from the Display Module Source Display

Example:

This example shows you how to set an unconditional breakpoint using F6 (Add/clear breakpoints).

1. Enter DSPMODSRC. The Display Module Source display is shown.
2. To display the module source that you want to modify, type `display module name`, where *name* is the file name of the module you want to modify, and press Enter.
3. For each unconditional breakpoint you want to set:
 - a. Place the cursor on the line that should follow the new breakpoint.
 - b. Press F6 (Add/Clear breakpoint).

Note: If there is no breakpoint on the line you specify, an unconditional breakpoint is set on that line. If there is a breakpoint on the line you specify, it is removed (even if it is a conditional breakpoint).

The following example shows an unconditional breakpoint set at line 50 of module T1520PG1:

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
 46      {
 47      if (j<0) return(0);
 48      if (hold_formatted_cost[i] == '$')
 49      {
50          formatted_cost[j] = hold_formatted_cost[i];
 51          break;
 52      }
 53      if (i<16 && !((i-2)%3))
 54      {
 55          formatted_cost[j] = ',';
 56          --j;
 57      }
 58      formatted_cost[j] = hold_formatted_cost[i];
 59      --j;
 60      }
Debug . . . -----
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch  F24=More keys
Breakpoint added to line 50
```

Note: To remove a breakpoint, use the CLEAR command. For example, `clear 50` removes the breakpoint at line 50.

4. After all breakpoints are set:
 - a. Press F12 (Cancel) to leave the Work with Module Breakpoints display.
 - b. Press F3 (End Program) to leave the ILE source debugger. Your breakpoints are not removed.

Setting Unconditional Breakpoints from the Command Line

To set an unconditional breakpoint using the BREAK debug command, enter `BREAK line-number` on the debug command line. The variable *line-number* is the line number in the currently displayed view of the module on which you want to set a breakpoint.

Note: To remove an unconditional breakpoint using the CLEAR debug command, enter: `CLEAR line-number` on the debug command line.

Setting Conditional Breakpoints for a Macro

To set a breakpoint on the first statement of a multi-statement macro, position the cursor on the line containing the macro invocation, not the macro expansion.

Example:

This example shows you how to set a conditional breakpoint using F13 (Work with module breakpoints).

1. To work with a module, enter `DSPMODSRC`. The Display Module Source display is shown.
2. To display the module source that you want to modify, type `display module name`, where *name* is the file name of the module you want to modify, and press Enter.
3. For each conditional breakpoint you want to set:
 - a. Place the cursor on the line that should follow the new breakpoint.
 - b. Press F13 (Work with module breakpoints). The Work with Module Breakpoints display appears.
 - c. Place the cursor on the first line of the list, type 1 (Add), and press Enter. For example, to set a conditional breakpoint at line 35 when *i* is equal to 21:
 - i) In the *Line* type, enter 35.
 - ii) In the *Condition* field, type `i==21`.
 - iii) Press Enter

Note: If you do not want to switch panels, you can set the same breakpoint from the Display Module Source command line by typing:

```
break 35 when i==21
```

A conditional breakpoint is set on line 35.

```

                                Work with Module Breakpoints
Program . . . . : T1520PG1          Library . . . . : MYLIB
Module . . . . : T1520IC2         Type . . . . . : *PGM
Type options, press Enter.
  1=Add   4=Clear
Opt   Line   Condition
  1     35    i==21
  -     50

```

Note: An existing breakpoint is always replaced by a new breakpoint entered at the same location.

4. After all breakpoints are set:

- a. Press F12 (Cancel) to leave the Work with Module Breakpoints display.
- b. Press F3 (End Program) to leave the ILE source debugger. Your breakpoints are not removed.

Setting Conditional Breakpoints for a Statement

You can set a conditional breakpoint to a statement. For example, if you have a compiler listing that contains line numbers and statement numbers, you can use the statement syntax to set a breakpoint on a specific statement when there are several statements on a single line.

```
Line  Stmt   Source
33    24     i=j; j=0;
34    26     array[i] = cost;
```

Break myfunction/25 sets a breakpoint on the statement `j=0` assuming this is in `myfunction`. If you then enter `Break 33`, a breakpoint is set at statement 24, `i=j`.

To set a breakpoint on the first statement of a multi-statement macro, position the cursor on the line containing the macro invocation, not the macro expansion.

Setting Conditional Thread Breakpoints

You can set or remove a conditional thread breakpoint by using:

- The Work with Module Breakpoints display
- The TBREAK debug command to set a conditional thread breakpoint in the current thread
- The CLEAR debug command to remove a conditional thread breakpoint

Setting a Conditional Thread Breakpoint from the Work with Module Breakpoints Display

To set a conditional thread breakpoint using the Work with Module Breakpoints display:

1. Press F13 to display Work with Module Breakpoints and press Enter.
2. Type 1 (Add) in the *Opt* field and press Enter.
3. Fill in the remaining fields as if it were a conditional job breakpoint.

To remove a conditional thread breakpoint using the Work with Module Breakpoints display: Type 4 (Clear) in the *Opt* field next to the breakpoint you want to remove and Press Enter.

Setting a Conditional Thread Breakpoint from the Command Line

You use the same syntax for the TBREAK debug command as you would for the BREAK debug command. The difference between these commands is that the BREAK debug command sets a conditional job breakpoint at the same position in all threads, while the TBREAK debug command sets a conditional thread breakpoint in the current thread.

Note: To remove a conditional thread breakpoint, use the CLEAR debug command. When a conditional thread breakpoint is removed, it is removed for the current thread only.

Testing Breakpoints

1. Call the program.
2. When an unconditional breakpoint is reached, the program stops and the Display Module Source display is shown again.
3. When a conditional breakpoint is reached, the expression is evaluated before the statement is run.
 - If the result is true (in the example, if `i` is equal to 21), the program stops, and the Display Module Source display is shown.
 - If the result is false, the program continues to run.

Removing All Breakpoints

You can remove all breakpoints, conditional and unconditional, from a program that has a module shown on the Display Module Source display by using the CLEAR PGM debug command. To use the debug command, type CLEAR PGM on the debug command line and press Enter. The breakpoints are removed from all of the modules bound to the program.

Using Watches to Aid Debugging

Use a *watch condition* to monitor changes in the current value of a variable or an expression which determines the address of a storage location. Setting watch conditions is similar to setting conditional breakpoints, with one important difference:

- Watch conditions stop the program as soon as the value of a variable changes from its current value.
- Conditional breakpoints stop the program only if the condition stated in the associated expression is satisfied when the statement is executed.

The debugger watches a variable through the content of a *storage address*, computed at the time the watch condition is set. When the content at the storage address is changed from the value it had when the watch condition was set or when the last watch condition occurred, a breakpoint is set, and the program stops.

Note: After a watch condition has been registered, the new content at the watched storage location is saved as the new current value of the corresponding variable. The next watch condition will be registered if the new content at the watched storage location changes subsequently.

This topic describes:

- Characteristics of watches
- How to set and remove watch conditions
- An example of setting a watch condition on a variable
- How to display active watches

Characteristics and Limitations Of Watches

When using watches, keep the following watch characteristics in mind:

- Watches are monitored on a system-wide basis, with a maximum number of 256 watches that can be active simultaneously. This number includes watches set by the system.

Depending on overall system use, you may be limited in the number of watch conditions you can set at a given time. If you try to set a watch condition while the maximum number of active watches across the system is exceeded, you will receive an error message and the watch condition is not set.

Note: If a variable crosses a page boundary, two watches are used internally to monitor the storage locations. Therefore, the maximum number of variables that can be watched simultaneously on a system-wide basis ranges from 128 to 256.

- Watch conditions can be set only when a program is stopped under debug, and the variable to be watched is in scope. If this is not the case, an error message is issued when a watch is requested, indicating that the corresponding call stack entry does not exist.
- Once the watch condition is set, the address of a storage location that is watched does not change. Therefore, if a watch is set on a temporary location, it could result in spurious watch-condition notifications.

An example of this is the automatic storage of an ILE C/C++ procedure, which can be reused after the procedure ends.

A watch condition may be triggered even though the watched variable is no longer in scope. You must not assume that a variable is in scope just because a watch condition has been reported.

- Two watch locations in the same job must not overlay in any way. Two watch locations in different jobs must not start at the same storage address; otherwise, overlap is allowed. If these restrictions are violated, an error message is issued.

Note: Changes that are made to a watched storage location are ignored if they are made by a job other than the one that set the watch condition.

- Eligible programs are automatically added to the debug session if they cause the watch-stop condition.
- When multiple watch conditions are hit on the same program statement, only the first one will be reported.
- You can set watch conditions when you are using service jobs for debugging, that is when you debug one job from another job.
- If a program in your session changes the content of a watched storage location and a watch command is successfully run, your application is stopped and the Display Module Source display is shown.

If the program has debug data and a debug view is available, the debug data is shown. The source line highlighted is the next statement to run (after the statement that changed the storage location). A message indicates which watch condition was satisfied.

Note: If the program cannot be debugged, the text area of the display is blank.

Setting and Removing Watch Conditions

Your program must be stopped under debug, and the variable you want to watch must be in scope before you can set a watch condition:

- To watch a global variable, you must ensure that the program in which the variable is defined is active before setting the watch condition.
- To watch a local variable, you must step into the function in which the variable is defined before setting the watch condition.

Setting watch conditions

You can set a watch condition by using:

- F17 (watch variable) to set a watch condition for the variable under the cursor.
- The WATCH debug command with or without its parameters.

Using the WATCH Debug Command

If you use the WATCH command, it must be entered as a single command; no other debug commands are allowed on the same command line.

- To access the Work with Watch display shown below, enter WATCH without any parameters.

```

Work with Watch
System:  DEBUGGER

Type options, press Enter.
4=Clear 5=Display
Opt   Num   Variable      Address      Length
-     1     salary        080090506F027004  4

Command
===>
F3=Exit F4=Prompt F5=Refresh F9=Retrieve F12=Cancel
Bottom

```

The Work with Watch display shows all watches currently active in the debug session. You can clear, and display watches from this display. When you select 5=Display, the Display Watch window that is shown below displays information about the currently active watch.

```

Work with Watch
Display Watch
.....:
.....:                               DEBUGGER
:
: Watch Number .....: 1
: Address .....: 080090506F027004
: Length .....: 4
: Number of Hits ..: 0
:
: Scope when watch was set:
:   Program/Library/Type:  PAYROLL  ABC  *PGM
:
:   Module...:  PAYROLL
:   Procedure:  main
:   Variable..:  salary
:
: F12=Cancel
:
.....:
Command
===>
F3=Exit F4=Prompt F5=Refresh F9=Retrieve F12=Cancel
Bottom

```

- To specify a variable to be watched, enter the following debug command:

```
WATCH variable
```

This command requests a breakpoint to be set if the value of *variable* is changed from its current value. For example, watch *V*, where *V* is a variable.

- To specify an expression to be watched, enter the following debug command:

```
WATCH expression
```

This command requests a breakpoint to be set if the value of *expression* is changed from its current value.

Note: *expression* is used to determine the address of the storage location to watch and must resolve to a location that can be assigned to, for example: watch $(p+2)$, where *p* is a pointer.

The scope of the expression variables in a watch is defined by the most recently issued QUAL command.

- To set a watch condition and specify a watch length, enter WATCH *expression* : *watch-length*.

Each watch allows you to monitor and compare a maximum of 128 bytes of contiguous storage. If the maximum length of 128 bytes is exceeded, the watch condition will not be set, and the debugger issues an error message.

By default, the length of the expression type is also the length of the watch-comparison operation. The `watch-length` parameter overrides this default. It determines the number of bytes of an expression that should be compared to determine if a change in value has occurred.

For example, if a 4-byte binary integer is specified as the variable, without the `watch-length` parameter, the comparison length is four bytes. However, if the `watch-length` parameter is specified, it overrides the length of the expression in determining the watch length.

Removing Watch Conditions

Watches can be removed in the following ways:

- The `CLEAR` command that is used with the `WATCH` keyword selectively ends one or all watches. For example, to clear the watch that is identified by `watch-number`, enter:

```
CLEAR WATCH watch-number
```

The watch number can be obtained from the Work with Watches display.

To clear all watches for your session, enter:

```
CLEAR WATCH ALL
```

Note: While the `CLEAR PGM` command removes all breakpoints in the program that contains the module being displayed, it has no effect on watches. You must explicitly use the `WATCH` keyword with the `CLEAR` command to remove watch conditions.

Automatic Removal Of Watch Conditions

Watches are also removed in the following ways:

- The `CL End Debug (ENDDBG)` command removes watches that are set in the local job or in a service job.

Note: `ENDDBG` will be called automatically in abnormal situations to ensure that all affected watches are removed.

- The initial program load (IPL) of your IBM i platform removes all watch conditions system-wide.

Example Of Setting a Watch Condition

In this example, you watch a variable `salary` in program `MYLIB/PAYROLL`. To set the watch condition, type `WATCH salary` on a debug line, accepting the default value for the `watch-length`.

If the value of the variable `salary` changes subsequently, the application stops, and the Display Module Source display is as shown:

Display Module Source

```

Program:  PAYROL          Library:  MYLIB          Module:  PAYROLL
52  for  (cnt=0;
53      cnt<EMPMAX &&
54      scanf("%s%s%f%d%d", payptr->first, payptr->last,
55            &(payptr->wage), &eflag, &(payptr->hrs))!=EOF;
56      cnt++, payptr++)
57  {
58      payptr->exempt=eflag;
59  }
60  empsort(payfile, cnt);
61  for  (index=1, payptr=payfile; index<=cnt; index++,payptr++) {
62      if  (payptr->exempt==1) {
63          salary = 40*(payptr->wage);
64          numexempt++; }
65      else
66          salary = (payptr->hours)*(payptr->wage);

```

More...

```

Debug . . . -----
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch  F24=More keys
Watch number 1 at line 64, variable: salary

```

- The line number of the statement where the change to the watch variable was detected is highlighted. This is typically the first executable line *following* the statement that changed the variable.
- A message indicates that the watch condition was satisfied.

If a text view is not available, a blank Display Module Source display is shown, with the same message as above in the message area.

Display Module Source

(Source not available)

```

F3=End program  F12=Resume  F14=Work with module list  F18 Work with watch
F21=Command entry  F22=Step into  F23=Display output
Watch number 1 at instruction 18, variable: salary

```

The following programs cannot be added to the ILE debug environment:

- ILE programs without debug data
- OPM programs with non-source debug data only
- OPM programs without debug data

In the first two cases, the stopped statement number is passed. In the third case, the stopped MI instruction is passed. The information is displayed at the bottom of a blank Display Module Source display as shown above. Instead of the line number, the statement or the instruction number is given.

Displaying Active Watches

To display a system-wide list of active watches and show which job set them, type DSPDBGWCH on the command line. This command brings up the Display Debug Watches display that is shown below.

Display Debug Watches

```

System:  DEBUGGER
-----Job-----
MYJOBNAME1  MYUSERPRF1  123456      1      4      080090506F027004
JOB4567890  PRF4567890  222222     1      4      09849403845A2C32
JOB4567890  PRF4567890  222222     2      4      098494038456AA00
JOB        PROFILE   333333     14     4      040689578309AF09
SOMEJOB    SOMEPROFIL  444444     3      4      005498348048242A
Bottom
Press Enter to continue
F3=Exit  F5=Refresh  F12=Cancel

```

Note: This display does not show watch conditions that are set by the system.

Stepping Through Programs

The STEP command of the ILE source debugger allows you to run a specified number of statements of a program, and then return to the Display Module Source display at the position of the next statement to be run. The cursor is positioned on this statement if the cursor was in the text area of the display the last time the source was displayed. Otherwise, it is positioned on the debug command line. The program begins at the statement where the program stopped. Setting a breakpoint causes the program to stop before the statement is run. The default number of statements to run is one.

This topic describes how to:

- Step over programs
- Step into programs
- Step over procedures
- Step into procedures

Stepping Over Programs

You can step over programs by using:

- F10 (Step) on the Display Module Source display
- Step Over debug command

Using F10 to Step Over Programs

Use F10 (Step) on the Display Module Source display to step over a called program in a debug session. If the next statement to be run is a CALL statement to another program, pressing F10 (Step) causes the called program to run to completion before the calling program is stopped again.

Using the STEP OVER Debug Command

Use the Step Over debug command to step over a called program in a debug session. To use the Step Over debug command, enter `STEP number-of-statements OVER`. The variable *number-of-statements* is the number of statements of the program that you want to run in the next step before the program is halted again.

If this variable is omitted, the default is 1. If one of the statements that are run contains a call to another program, the ILE source debugger steps over the called program.

Stepping into Programs

Step into programs by using:

- F22 (Step into) on the Display Module Source display
- The STEP INTO debug command.

Using F22 to Step into Programs

Use F22 (Step into) on the Display Module Source display to step into a called program in a debug session. If the next statement to be run is a Call (CALL) statement to another program, pressing F22 causes the first executable statement in the called program to be run. The called program is then shown in the Display Module Source display.

Note: The called program must have debug data associated with it in order for it to be shown in the Display Module Source display.

Using the STEP INTO Debug Command

Use the STEP INTO debug command to step into a called program in a debug session. To use the STEP INTO debug command, enter:

```
STEP number-of-statements INTO
```

The variable *number-of-statements* is the number of statements of the program that you want to run in the next step before the program is halted again. If this variable is omitted, the default is 1.

Stepping into Called Programs

If one of the statements being run contains a Call (CALL) statement to another program, the source debugger steps into the called program. Each statement in the called program is counted in the step. If the step ends in the called program, the called program is shown in the Display Module Source display. For example, if you enter `STEP 5 INTO`, the next five statements of the program are run. If the third statement is a Call (CALL) statement to another program, two statements of the calling program are run and the first two statements of the called program are run.

Note: The step is counted as a statement.

The `STEP INTO` command works with the Call (CALL) command as well. You can take advantage of this to step through your program after calling it. After starting the source debugger, from the initial Display Module Source display, enter `STEP 1 INTO` and press the Enter key. This sets the step count to 1.

Example of Stepping into a Program Using F22

Use F22 (Step Into) to step into program CPGM from the program DEBUGEX.

1. Assume that the Display Module Source display shows the source for DEBUGEX.
2. To set an unconditional breakpoint at line 92, which is the last executable statement before the call to function `CalcTax()` in program CPPPGM, type `Break 92` and press Enter.
3. Press F3 (End Program) to leave the Display Module Source display.
4. Call the program. The program stops at breakpoint 92, as shown in [Figure 69 on page 115](#).

DEBUGEX Before Stepping Into CPGM

```

                                Display Module Source
Program:  DEBUGEX      Library:  MYLIB      Module:  DEBUGEX
 88      cout << "Please enter amount" << endl;
 89      cin >> input;
 90      if (input > MINIMUM) {
 91          // call function CalcTax in separate program CPPPGM
 92          retval1 = CalcTax(input);
 93          if (retval1 > LIMIT)
 94              retval2 = CalcSurtax(input)
 95      }
 96      cout << "Total tax is " << retval1 = retval2 << endl;
 97      }
 98
 99
100
101
102
                                More...
Debug . . . -----
F3=End program   F6=Add/Clear breakpoint   F10=Step   F11=Display variable
F12=Resume       F17=Watch variable       F18=Work with watch   F24=More keys
Breakpoint at line 90
```

Figure 69. Module Source Display for DEBUGEX

5. Press F22 (Step into). One statement of the program is run, and then the Display Module Source display of CPGM is shown.

The program stops at the first executable statement of CPGM (line 13).

Note: You cannot specify the number of statements to step through when you use F22. Pressing F22 performs a single step.

```

Display Module Source
Program:  CPGM          Library:  MYLIB
 1      *=====
 2      * CPGM - Program called by DEBUGEX to illustrate the
 3      *          STEP functions of the ILE source
 4      *debugger
 5      * This program receives a parameter input from DEBUGEX,
 6      * calculates a tax amount, and then returns
 7      *=====
 8
 9      double CalcTax(double input)
10{
11      double tax;
12
13      tax= input * TAXRATE
14
15      return taxrate;
                                           Bottom
Debug . . . -----
-----
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch  F24=More keys
Step completed at line 13.

```

Figure 70. Module Source Display After Stepping into CPGM

If there is no debug data available, you see a blank Display Module Source display with a message indicating that the source is not available.

Stepping into an OPM Program

When calls to other functions are encountered, you can step into an OPM program if it has debug data available and if the debug session accepts OPM programs for debugging.

If the ILE source debugger is not set to accept OPM programs, or if there is no debug data available, then you see a blank Display Module Source display with a message indicating that the source is not available. An OPM program has debug data if it was compiled with OPTION(*LSTDBG).

The default step mode is step over.

Stepping Over Procedures

If you specify `over` on the STEP debug command, calls to procedures and functions count as single statements. This is the default STEP mode. Stepping through four statements of a program could result in running 20 statements if one of the four is a call to a procedure with 16 statements. You can start the step-over function by using:

- The STEP OVER debug command
- F10 (Step)

Example:

This example shows you how to use F10 (Step) to step over one statement at a time in your program.

1. To work with a module type DSPMODSRC and press Enter. The Display Module Source display is shown.
2. Enter `display module T1520IC2`.
3. To set an unconditional breakpoint at line 50, enter `Break 50` on the debug command line.
4. To set a conditional breakpoint at line 35, enter `Break 35 when i==21` on the debug command line.
5. Press F12 (Resume) to leave the Display Module Source display.
6. Call the program. The program stops at breakpoint 35 if `i` is equal to 21, or at line 50 whichever comes first.
7. To step over a statement, press F10 (Step). One statement of the program runs, and then the Display Module Source display is shown. If the statement is a function call, the function runs to completion. If

the called function has a breakpoint set, however, the breakpoint will be hit. At this point you are in the function and the next step will take you to the next statement inside the function.

Note: You cannot specify the number of statements to step through when you use F10. Pressing F10 performs a single step.

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
47      if (j<0) return(0);
48      if (hold_formatted_cost[i] == '$')
49      {
50          formatted_cost[j] = hold_formatted_cost[i];
51          break;
52      }
53      if (i<16 && !((i-2)%3))
54      {
55          formatted_cost[j] = ',';
56          --j;
57      }
58      formatted_cost[j] = hold_formatted_cost[i];
59      --j;
60  }
61
Debug . . . -----
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch  F24=More keys
Breakpoint at line 50.
```

8. To step over 5 statements, enter `step 5 over` on the debug command line. The next five statements of your program run, and then the Display Module Source display is shown.

If the third statement is a call to a function, the first two statements run, the function is called and returns, and the last two statements run.

9. To step over 11 statements, enter `step 11 over` on the debug command line. The next 11 statements of your program runs. The Display Module Source display is shown.

Stepping into Procedures

There is an automatic feature for stepping. This feature automatically puts a service program into debug. This happens if the service program that is stepped into from another program in debug:

- Has debug data
- Is not in debug
- Contains a procedure

The service program is added to debug for the user, and the DSPMODSRC panel shows the procedure in the service program. From this point, modules in the service program can be accessed using the Work with Modules display just like modules in programs the user added to debug.

If you specify *INTO* on the STEP debug command, each statement in a procedure or function that is called counts as a single statement. You can start the step into function by using:

- The STEP INTO debug command
- F22 (Step into)

Example:

This example shows you how to use F22 (Step Into) to step into one procedure.

1. Enter DSPMODSRC. The Display Module Source display is shown.
2. To set an unconditional breakpoint at line 50, enter `Break 50` on the debug command line.
3. To set a conditional breakpoint at line 35, enter `Break 35 when i==21` on the debug command line.
4. Press F12 (Resume) to leave the Display Module Source display.
5. Call the program. The program stops at breakpoint 35 if *i* is equal to 21 or at line 50 whichever comes first.

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
 47      if (j<0) return(0);
 48      if (hold_formatted_cost[i] == '$')
 49      {
 50          formatted_cost[j] = hold_formatted_cost[i];
 51          break;
 52      }
 53      if (i<16 && !((i-2)%3))
 54      {
 55          formatted_cost[j] = ',';
 56          --j;
 57      }
 58      formatted_cost[j] = hold_formatted_cost[i];
 59      --j;
 60  }
 61
Debug . . . -----
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
Breakpoint at line 50.

```

6. Press F22 (Step into). One statement of the program runs, and then the Display Module Source display is shown. If the statement is a procedure or function call, the program stops at the first statement of the procedure or function.

Note: You cannot specify the number of statements to step through when you use F22. Pressing F22 performs a single step.

7. To step into 5 statements, enter `step 5 into` on the debug command line.

The next five statements of your program are run, and then the Display Module Source display is shown. If the third statement is a call to a function, the first two statements of the calling procedure run, and the first two statements of the function run.

Note: The step is counted as a statement.

8. To step into 11 statements, enter `step 11 into` on the debug command line. The next 11 statements of your program runs. The Display Module Source display is shown.

Debugging Variables

You can display the value of scalar variables, expressions, structures, arrays, or `errno` and change the value of scalar variables or `errno` using the EVAL debug command.

This topic describes how to:

- [Display values of scalar variables, expressions, structures, arrays, or `errno` during a debug session](#)
- [Change the value of a variable by using the EVAL debug command](#)
- [Change the value of scalar variables or `errno` during a debug session](#)
- [Equate a shorthand name with a variable, expression, or command during a debug session](#)

Sample source is provided that illustrates uses of the EVAL debug command.

Displaying the Value Of a Variable

To display or change the value of a variable:

- The module that is shown on the Display Module Source display must be bound to a program that is in a debug session.
- The program must be called and stopped at a breakpoint or step location.

The scope of the variables used in the EVAL debug command is defined by using the QUAL debug command.

- The EVAL debug command
- F11 (Display variable)

You can use the Enter key as a toggle switch between displays.

Note: You can change variables by using the EVAL debug command with assignment.

Using F11 to Display Variables

The easiest way to display data or an expression is to use F11 (Display variable) on the Display Module Source display. Place your cursor on the variable that you want to display and press F11. The current value of the variable is shown on the message line at the bottom of the Display Module Source display.

In cases where you are evaluating structures, records, classes, arrays, pointers, enumerations, bit fields, unions or functions, the message returned when you press F11 (Display variable) may span several lines. Messages that span several lines are shown on the Evaluate Expression display to show the entire text of the message. Once you have finished viewing the message on the Evaluate Expression display, press Enter to return to the Display Module Source display.

The Evaluate Expression display also shows all the past debug commands that you entered and the results from these commands. You can search forward or backward on the Evaluate Expression display for a specified string, or text and retrieve or re-issue debug commands.

Example:

This example shows you how to use the F11 (Display variable) to display a variable.

1. Enter DSPMODSRC. The Display Module Source display is shown.
2. Enter display module T1520IC2.
3. Place the cursor on the variable hold_formatted_cost on line 50 and press F11 (Display variable). A pointer to the array is shown on the message line in the following.

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
47      if (j<0) return(0);
48      if (hold_formatted_cost[i] == '$')
49      {
50          formatted_cost[j] = hold_formatted_cost[i];
51          break;
52      }
53      if (i<16 && !((i-2)%3))
54      {
55          formatted_cost[j] = ',';
56          --j;
57      }
58      formatted_cost[j] = hold_formatted_cost[i];
59      --j;
60  }
61

                                                                    More...
Debug . . . -----
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
hold_formatted_cost = SPP:C048BD0003F0
```

Messages with multiple line responses will cause the Evaluate Expression display to be shown. This display will show all response lines. It also shows a history of the debug commands entered and the results from these commands. To return to the Display Module Source display, press the Enter key. You can use the Enter key as a toggle switch between displays. Single-line responses will be shown on the Display Module Source message line.

You can also use the EVAL debug command to determine the value of an expression. For example, if j has a value of 1024, enter eval (j * j)/512 on the debug command line. You use the QUAL debug command to determine the line or statement number within the function that you want the variables scoped to for the EVAL debug command. The Evaluate Expression display shows (j * j)/512 = 2048.

Changing the Value of a Variable

You can change variables by using the EVAL debug command with assignment. To specify the scope of the EVAL command, use a QUAL command.

Example:

This example shows you how to use the EVAL debug command to assign an expression to a variable.

1. Enter DSPMODSRC. The Display Module Source display is shown.
2. Enter `display module T1520IC2`.
3. To specify the scope of the EVAL command you can use a QUAL command. For example, `QUAL 48` will qualify the EVAL command to the scope that line 48 is located at. Line 48 is the number within the function to which you want the variables scoped for the following EVAL debug command.

Note: You do not always have to use the QUAL debug command before the EVAL debug command. An automatic QUAL is done when a breakpoint is encountered or a step is done. This establishes the default for the scoping rules to be the current stop location.

4. To change an expression in the module shown enter: `EVAL x=<expr>`, where `x` is the variable name and `<expr>` is the expression you want to assign to variable `x`.

For example, `"EVAL hold_formatted_cost [1] = '#'"` changes the array element at 1 from \$ to # and shows `"hold_formatted_cost[1]= '#' = '#':"` on the Display Module Source display as shown:

```

                                Display Module Source
Program:  T1520PG1      Library:  MYLIB      Module:  T1520IC2
 47      if (j<0) return(0);
 48      if (hold_formatted_cost[i] == '$')
 49      {
 50          formatted_cost[j] = hold_formatted_cost[i];
 51          break;
 52      }
 53      if (i<16 && !((i-2)%3))
 54      {
 55          formatted_cost[j] = ',';
 56          --j;
 57      }
 58      formatted_cost[j] = hold_formatted_cost[i];
 59      --j;
 60  }
 61

Debug . . . -----
F3=End program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable      F18=Work with watch  F24=More keys
hold_formatted_cost[1]= '#' = '#'
```

Changing the Value of a Scalar Variable

Change the value of scalar variables using the EVAL debug command with an assignment operator (=). The program must be called and stopped at a breakpoint or step location to change the value. To change the value of a variable, enter:

```
EVAL variable-name = value
```

where `variable-name` is the name of the variable that you want to change and `value` is an identifier, literal, or constant value that you want to assign to variable `variable-name`.

Example:

```
EVAL COUNTER=3
```

changes the value of COUNTER to 3 and shows

```
COUNTER=3 = 3
```

on the message line of the **Display Module Source** display.

When you assign values to a character variable, the following rules apply:

- If the length of the source expression is less than the length of the target expression, the data is left justified in the target expression and the remaining positions are filled with blanks.
- If the length of the source expression is greater than the length of the target expression, the data is left justified in the target expression and truncated to the length of the target expression.

The scope of the variables used in the EVAL debug command is defined by using the QUAL debug command. To change a variable at line 48, enter QUAL 48. Line 48 is the number within a function to which you want the variables scoped for the EVAL debug command.

Note: You do not always have to use the QUAL debug command before the EVAL debug command. An automatic QUAL is done when a breakpoint is encountered or a step is done. This establishes the default for the scoping rules to be the current stop location.

The example below shows the results of changing the array element at 1 from \$ to #.

```
EVAL hold_formatted_cost [1] = '#'  
    hold_formatted_cost[1]= '#' = '#':  
  
//Code evaluated before statement 51 where a breakpoint is set  
47     if (j<0) return(0);  
48     if (hold_formatted_cost[i] == '$')  
49     {  
50         formatted_cost[j] = hold_formatted_cost[i];  
51         break;  
52     }  
53     if (i<16 && !((i-2)%3))  
54     {  
55         formatted_cost[j] = ',';  
56         --j;  
57     }  
58     formatted_cost[j] = hold_formatted_cost[i];  
59     --j;  
60 }  
61
```

Figure 71. Using EVAL to Change a Variable

Equating a Name with a Variable, Expression, or Debug Command

You can equate a name with a variable, expression, or debug command for shorthand use. You can then use that name alone or within another expression. If you use it within another expression, the value of the name is determined before the expression is evaluated. Equates stay active until a debug session ends or a name is removed.

Example:

This example shows you how to use the Equate debug command with a variable name.

1. Enter DSPMODSRC. The Display Module Source display is shown.
2. To equate an expression, enter `equate <name> <definition>` where `<name>` is a character string that contains no blanks and `<definition>` is a character string separated from `<name>` by at least one blank. The character strings can be in uppercase, lowercase, or mixed case. The length of the character strings combined is limited to 144 characters, which is the length of the command line. After any Equates have been expanded, the length is limited to 150 characters, which is the maximum command length. For example, enter `equate dv display variable`.

If a definition is not supplied, and a previous Equate debug command has defined the name, the previous definition is removed. If the name was not previously defined, an error message is shown.

To see the Equates that are defined for this debug session, enter: `display equate`. A list of the active Equates is shown on the Evaluate Expression display.

Displaying a Structure

The following example shows a structure with two elements being displayed. Each element of the structure is formatted according to its type and displayed.

1. Enter DSPMODSRC. The Display Module Source display is shown.
2. Set a breakpoint at line 9.
3. Press F12 (Resume) to leave the Display Module Source display.
4. Call the program. The program stops at the breakpoint at line 9.
5. Enter `eval test` on the debug command line, as shown:

```

Display Module Source
Program:  TEST1          Library:  DEBUG          Module:  MAIN
 1 struct {
 2   char charValue;
 3   unsigned long intValue;
 4 } test;
 5
 6 int main(){
 7   test.intValue = 10;
 8   test.charValue = 'c';
 9   test.charValue = 11;
10   return 0;
11 }
                                                    Bottom
Debug . . . eval test_____
F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch F24=More keys
```

6. Press Enter to go to the next display. The Evaluate Expression Display shows the entire structure as shown:

```

Evaluate Expression
Previous debug expressions
> BREAK 9
> EVAL test
   test.charValue = 'c'
   test.intValue = 10
```

7. Press Enter from the Evaluate Expression Display to return to the Display Module Source screen.

Displaying Variables As Hexadecimal Values

The following example shows the steps and syntax used to dump hexadecimal variables.

1. Enter DSPMODSRC. The Display Module Source display appears, as shown below.
2. Set a breakpoint at line 9.
3. Press F12 (Resume) to leave the Display Module Source display.
4. Call the program. The program stops at the breakpoint at line 9.
5. Enter `eval test: x 32` on the debug command line, as shown below.

```

                                Display Module Source
Program:  TEST1                Library:  DEBUG           Module:  MAIN
 1  struct {
 2     char charValue;
 3     unsigned long intValue;
 4  } test;
 5
 6  int main(){
 7     test.intValue = 10;
 8     test.charValue = 'c';
 9     test.charValue = 11;
10     return 0;
11  }
                                Bottom
Debug . . . eval test: x 32_-----
-----
F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable      F18=Work with watch  F24=More keys

```

6. The Evaluate Expression display appears. As requested, 32 bytes are shown, but only the first 8 bytes are meaningful. The left column is an offset in hex from the start of the variable. The right column is an EBCDIC character representation of the data. If no length is specified after the 'x', the size of the variable is used as the length. A minimum of 16 bytes is displayed. Press the Enter key to return to the Display Module Source display.

```

                                Evaluate Expression
Previous debug expressions
> BREAK 9
> EVAL test: x 32
000000  83000000 0000000A 00000000 00000000  - c.....
000100  00000000 00000000 00000000 00000000  - .....

```

Displaying Null-Ended Character Arrays

The following example shows the display of a character string. The array must be dereferenced by the '*' operator. If the * operator is not entered, the array is displayed as a space pointer. If the dereferencing operator is used, but the 's' is not appended to the expression, only the first array element is displayed.

1. While in a debug session, enter DSPMODSRC. The Display Module Source display is shown.
2. Set a breakpoint at line 6.
3. Press F12(Resume) to leave the Display Module Source Display.
4. Call the program. The program stops at the breakpoint at line 6.
5. Enter eval *array1: s on the debug command line, as shown:

```

                                Display Module Source
Program:  TEST3                Library:  DEBUG           Module:  MAIN
 1  #include <string.h>
 2  char array1 [11];
 3  int i;
 4  int main(){
 5     strcpy(array1,"0123456789");
 6     i = 0;
 7     return 0;
 8  }
                                Bottom
Debug . . . eval *array1: s_-----
-----
F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable      F18=Work with watch  F24=More keys

```

The following shows the value of the array. A string length of up to 65535 can follow the s character. Formatting will stop at the first null character encountered. If no length is specified, formatting will stop after 30 characters or the first null, whichever is less.

```

                                Display Module Source
Program:  TEST3                Library:  DEBUG           Module:  MAIN
 1  #include <string.h>
 2  char array1 [11];
 3  int i;
 4  int main(){
 5      strcpy(array1,"0123456789");
 6      i = 0;
 7      return 0;
 8  }

                                Bottom
Debug . . . -----
F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume      F17=Watch variable  F18=Work with watch  F24=More keys
*array1: s = "0123456789"

```

The following example shows the usage of the `:f` syntax to specify that the newline character (`x'15'`) should be scanned for while displaying string output. If the end of the display line occurs, the output is wrapped to the next display line.

When the `:f` formatting code is used, the text string will display on the current line until a newline is encountered. If no newline character is encountered before the end of the display screen line, the output is wrapped until a newline is found. DBCS SO/SI characters are added as necessary to make sure they are matched.

An example of `:f` format code usage is shown:

```

int main()
{
    char testc[]={ "This is the first line.\nThis is the second line."
                  "\nThis is the third line."};
    int i;
    i = 1;
}

```

This program will result in the following screen output:

```

> EVAL *testcs 100
    *testcs 100 =
    "This is the first line. This is the second line. This is the"
    "third line."
> EVAL *testcf 100
    *testcf 100 =
    This is the first line.
    This is the second line.
    This is the third line.

```

Displaying Character Arrays

The following example shows the usage of the `:c` syntax to format an expression as characters. The array must be dereferenced by the `*` operator. If the `*` operator is not entered, the array will be displayed as a space pointer. If the dereferencing operator is used, but the `:c` is not appended to the expression, only the first array element is displayed. The default length of the display is 1.

1. While in a debug session, type `DSPMODSRC`. The Display Module Source display is shown.
2. Set a breakpoint at line 6.
3. Press `F12`(Resume) to leave the Display Module Source Display.
4. Call the program. The program stops at the breakpoint at line 6.
5. Enter `eval *array1: c 11` on the debug command line, as shown:

```

                                Display Module Source
Program:  TEST3                Library:  DEBUG           Module:  MAIN
1  #include <string.h>
2  char array1 [11];
3  int i;
4  int main(){
5  strcpy(array1,"0123456789");
6  i = 0;
7  return 0;
8  }

                                Bottom
Debug . . . eval *array1: c 11
-----
F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable  F18=Work with watch  F24=More keys

```

The following illustrates displaying 11 characters, including a null character. The null character appears as a blank.

```

                                Display Module Source
Program:  TEST3                Library:  DEBUG           Module:  MAIN
1  #include <string.h>
2  char array1 [11];
3  int i;
4  int main(){
5  strcpy(array1,"0123456789");
6  i = 0;
7  return 0;
8  }

                                Bottom
Debug . . . -----
F3=Exit program  F6=Add/Clear breakpoint  F10=Step  F11=Display variable
F12=Resume       F17=Watch variable  F18=Work with watch  F24=More keys
*array1: c 11 = '0123456789 ' ...

```

Sample EVAL Commands for Pointers, Variables, and Bit Fields

“Sample EVAL Commands for Pointers, Variables, and Bit Fields” on page 125 shows the use of the EVAL command with pointers, variables, and bit fields. The pointers, variables, and bit fields are based on the source in “Source for Sample EVAL Commands” on page 129.

Sample EVAL Commands for Pointers, Variables, and Bit Fields

```

Pointers
// Display a pointer
>eval pc1
pc1 = SPP:0000C0260900107C

// Assign a value to a pointer
>eval pc2=pc1
pc2=pc1 = SPP:0000C0260900107C

// Dereference a pointer
>eval *pc1
*pc1 = 'C'

// Take the address of a pointer
>eval &pc1
&pc1 = SPP:0000C02609001040

// Build an expression with normal C precedence
>eval *&pc1
*&pc1 = SPP:0000C0260900107C

// Casting a pointer
>eval *(short *)pc1
*(short *)pc1 = -15616

// Treat an unqualified array as a pointer
>eval arr1
arr1 = SPP:0000C02609001070

```

```

// Apply the array type through dereferencing
// (character in this example)
>eval *arr1
  *arr1 = 'A'

// Override the formatting of an expression that is an lvalue
>eval *arr1:s
  *arr1:s = "ABC"

// Set a pointer to null by assigning 0
>eval pc1=0
  pc1=0 = SYP:*NULL

// Evaluate a function pointer
>eval fncptr
  fncptr = PRP:0000A0CD0004F010

// Use the arrow operator
>eval *pY->x.p
  *pY->x.p = ' '

Simple Variables
// Perform logical operations
>eval i1==u1 || i1<u1
  i1==u1 || i1<u1 = 0// Unary operators occur in proper order
>eval i1++
  i1++ = 100

// i1 is incremented after being used
>eval i1
  i1 = 101

// i1 is incremented before being used
>eval ++i1
  ++i1 = 102

// Implicit conversion
>eval u1 = -10
  u1 = -10 = 4294967286

// Implicit conversion
>eval (int)u1
  (int)u1 = -10

Bit Fields
// Display an entire structure
>eval bits
  bits.b1 = 1
  bits.b4 = 2

// Work with a single member of a structure
>eval bits.b4 = bits.b1
  bits.b4 = bits.b1 = 1

// Bit fields are fully supported
>eval bits.b1 << 2
  bits.b1 << 2 = 4

// You can overflow bit fields, but no warning is generated
>eval bits.b1 = bits.b1 << 2
  bits.b1 = bits.b1 << 2 = 4
>eval bits.b1
  bits.b1 = 0

```

The examples below show the use of the EVAL command with structures, unions, and enumerations. The structures, unions, and enumerations are based on the source in [“Source for Sample EVAL Commands” on page 129](#).

Note: For C++, the structures are simple structures, not Classes.

Sample EVAL Commands for C Structures, Unions and Enumerations

```

Structures and Unions
// Cast with typedefs
>eval (struct z *)&zz
  (struct z *)&zz = SPP:0000C005AA0010D0

```



```

// Cast with tags
>eval *(c *)&zz
  (*(c *)&zz).a = 1
  (*(c *)&zz).b = SYP:*NULL Structures and Unions// Assign union members
>eval u.x = -10
  u.x = -10 = -10

// Display a union. The union is formatted for each definition
>eval u
  u.y = 4294967286
  u.x = -10
Enumerations
// Display both the enumeration and its value
>eval Color
  Color = blue (2)
>eval Number
  Number = three (2)

// Cast to a different enumeration
>eval (enum color)Number
  (enum color)Number = blue (2)

// Assign by number
>eval Number = 1
  Number = 1 = two (1)

// Assign by enumeration
>eval Number = three
  Number = three = three (2)

// Use enums in an expression
>eval arr1[one]
  arr1[one] = 'A'

```

EVAL Commands for System and Space Pointers

The example below shows the use of the EVAL command with system and space pointers. The system and space pointers are based on the source in [“Source for Sample EVAL Commands for Displaying System and Space Pointers”](#) on page 130.

```

System and Space Pointers
// System pointers are formatted
// :1934:QTEMP      :111111110
>eval pSYSptr
  pSYSptr =
    SYP:QTEUSERSPC
    0011100
// Space pointers return 8 bytes that can be used in
// System Service Tools
>eval pBuffer
  pBuffer = SPP:0000071ECD000200

```

Figure 72. Sample EVAL Commands for System and Space Pointers

You can use the EVAL command on C and C++ language features and constructs. The ILE source debugger can display a full class or structure but only with those fields defined in the derived class. You can display a base class in full by casting the derived class to the particular base class.

The example below shows the use of the EVAL command with C++ language constructs. The C++ language constructs are based on the source in [“Source for Sample EVAL Commands for Displaying C++ Constructs”](#) on page 131. Additional C++ examples are provided in the source debugger online help.

```

// Follow the class hierarchy (specifying class D is optional)
> EVAL *(class D *)this
  (*(class D *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(class D *)this).__vbp1D = SPP:C40F5E3D7F000440
  (*(class D *)this).d = 4

// Follow the class hierarchy (without specifying class D)
> EVAL *(D *)this
  (*(D *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(D *)this).__vbp1D = SPP:C40F5E3D7F000440
  (*(D *)this).d = 4

// Look at a local variable
> EVAL VAR
  VAR = 1

// Look at a global variable
> EVAL ::VAR
  ::VAR = 2

// Look at a class member (specifying this-> is optional)
> EVAL this->f
  this->f = 6

// Look at a class member (without specifying this->)
> EVAL f
  f = 6

// Disambiguate variable ac
> EVAL A::ac
  A::ac = 12

// Scope operator with template
> EVAL E<int>::ac
  E<int>::ac = 12

// Cast with template:
> EVAL *(E<int> *)this
  (*(E<int> *)this).__vbp1B = SPP:C40F5E3D7F000490
  (*(E<int> *)this).__vbp1EXTi_ = SPP:C40F5E3D7F000400
  (*(E<int> *)this).e = 5

// Assign a value to a variable
> EVAL f=23
  f=23 = 23

// See all local variables in a single EVAL statement
> EVAL %LOCALVARS
  local = 828
  this = SPP:C40F5E3D7F000400
  VAR = 1

```

Figure 73. Sample EVAL Commands for C++ Expressions

Displaying a Class Template and a Function Template

To display a class template or a function template, enter `EVAL template-name` on the debug command line. The variable *template-name* is the name of the class template or function template you want to display.

The example below shows the results of evaluating a class template. You must enter a template name that matches the demangled template name. Type definition names are not valid because the typedef information is removed when the template name is mangled.

```

> EVAL XX<int>::a
XX<int>::= '1 '
> EVAL XX<inttype>::a
Identifier not found
1  template < class A >           //Code evaluated at line 8
2  class XX {                     //where a breakpoint was set
3      static A a;
4      static B b;
5  };
6  XX<int> x;
7  typedef int inttype;
8  int XX<int>::a =1;              //mangled name a__2XXXTi_
9  int XX<inttype>::b = 2;        //mangled name b__2XXXTi_

```

Figure 74. Using EVAL with a Class Template

The example below shows the results of evaluating a function template.

```

> EVAL XX<int,12>::sxa
XX<int,12>::sxa = '1 '
> EVAL xxobj.xca[0]
xxobj.xca[0] = '2 '
1  template < class A, int B>     //Code evaluated at lines 8 and 9
2  class XX {                     //where breakpoints were set
3      static A sxa;
4      char xca[B];
5  public:
6      XX(void) { xca[0] = 2; }
7  };
8  XX<int,12> xxobj;
9  int XX<int,2*6>::sxa =1;
                                     //same as intXX<int,12>::sxa
                                     //mangled name sxa__2XXXTiSP12_

```

Figure 75. Using EVAL with a Function Template

Source for Sample EVAL Commands

The sample EVAL commands presented in “[Sample EVAL Commands for Pointers, Variables, and Bit Fields](#)” on page 125 and “[Sample EVAL Commands for C Structures, Unions and Enumerations](#)” on page 126 are based on the source shown in the following figure:

```

#include <iostream.h>
#include <pointer.h>

/** POINTERS **/
_SYSPTR pSys;           //System pointer
_SPCPTR pSpace;        //Space pointer
int (*fncptr)(void);   //Function pointer
char *pc1;             //Character pointer
char *pc2;             //Character pointer
int *pi1;              //Integer pointer
char arr1[] = "ABC";   //Array

/** SIMPLE VARIABLES **/
int i1;                //Integer
unsigned u1;           //Unsigned Integer
char c1;               //Character
float f1;              //Float

/** STRUCTURES **/
struct {               //Bit fields
    int b1 : 1;
    int b4 : 4;
}bits;
struct x{              // Tagged structure
    int x;
    char *p;
};
struct y{              // Structure with
    int y;              // structure member

```

```

    struct x x;
};
typedef struct z {           // Structure typedef
    int z;
    char *p;
} z;
z zz;                        // Structure using typedef
z *pZZ;                      // Same
typedef struct c {          // Structure typedef
    unsigned a;
    char *b;
} c;
c d;                          // Structure using typedef

/** UNIONS **/
union u{                     // Union
    int x;
    unsigned y;
};
union u u;                   // Variable using union
union u *pU;                 // Same

/** ENUMERATIONS **/
enum number {one, two, three};
enum color {red,yellow,blue};
enum number Number = one;
enum color Color = blue;

```

```

/** FUNCTION **/
int ret100(void) { return 100;}
int main()
{
    float dec1;
    struct y y, *pY;
    bits.b1 = 1;
    bits.b4 = 2;
    i1 = ret100();
    c1 = 'C';
    f1 = 100e2;
    dec1 = 12.3;
    pc1 = &c1;
    pi1 = &i1;
    d.a = 1;
    pZZ = &zz;
    pZZ->z=1;
    pY = &y;
    pY->x.p=(char*)&y;
    pU=&u;
    pU->x=255;
    Number=(number)Color;
    fncptr = &ret100;
    pY->x.x=1;                // Set breakpoint here
    return 0;
}

```

Source for Sample EVAL Commands for Displaying System and Space Pointers



The sample EVAL command for displaying system and space pointers presented in [Figure 72 on page 127](#) is based on the source shown in the following figure:

```

#include <stdio.h>
#include <mispace.h>
#include <pointer.h>
#include <miscobj.h>
#include <except.h>
#include <lecond.h>
#include <leenv.h>
#include <qtedbgs.h>          /* From qsysinc */
/* Link up the Create User Space API */
#pragma linkage(CreateUserSpace,OS)
#pragma map(CreateUserSpace,"QUSCRTUS")
void CreateUserSpace(char[20],
                    char[10],
                    long int,
                    char,
                    char[10],
                    char[50],
                    char[10],
                    _TE_ERROR_CODE_T *
                    );
/* Link up the Delete User Space API */
#pragma linkage>DeleteUserSpace,OS)
#pragma map>DeleteUserSpace,"QUSDLTUS")
void DeleteUserSpace(char[20],
                    _TE_ERROR_CODE_T *
                    );
/* Link up the Retrieve Pointer to User Space API */
#pragma linkage(RetrievePointerToUserSpace,OS)
#pragma map(RetrievePointerToUserSpace,"QUSPTRUS")
void RetrievePointerToUserSpace(char[20],
                                char **,
                                _TE_ERROR_CODE_T *
                                );

int main (int argc, char *argv[])
{
    char *pBuffer;
    _SYSPTR pSYSptr;
    _TE_ERROR_CODE_T errorCode;
    errorCode.BytesProvided = 0;
    CreateUserSpace("QTEUSERSPCQTEMP      ",
                  "QTESSPC      ",
                  10,
                  0,
                  "*ALL      ",
                  "",
                  "*YES      ",
                  &errorCode
                  );

    /*! call RetrievePointerToUserSpace - Retrieve Pointer to User Space */
    /*!! (pass: Name and library of user space, pointer variable */
    /*!! return: nothing (pointer variable is left pointing to start*/
    /*!! of user space) */
    RetrievePointerToUserSpace("QTEUSERSPCQTEMP      ",
                              &pBuffer,
                              &errorCode);

    /* convert the space pointer to a system pointer */
    pSYSptr = _SETSPFP(pBuffer);

    printf("Space pointer: %p\n",pBuffer);
    printf("System pointer: %p\n",pSYSptr);

    return 0;
}

```

Figure 76. Source for Sample EVAL Commands for Displaying System and Space Pointers

Source for Sample EVAL Commands for Displaying C++ Constructs



The sample EVAL command for displaying C++ constructs presented in [Figure 73 on page 128](#) is based on the source shown in the following figure:

```
// Program demonstrates the EVAL debug command
class A {
public:
    union {
        int a;
        int ua;
    };
    int ac;
    int amb;
    int not_amb;
};

class B {
public:
    int b;
};

class C {
public:
    int ac;
    static int c;
    int amb;
    int not_amb;
};

int C::c = 45;
template <class T> class E : public A, public virtual B {
public:
    T e;
};

class D : public C, public virtual B {
public:
    int d;
};

class outter {
public:
    static int static_i;
    class F : public E<int>, public D {
public:
        int f;
        int not_amb;
        void funct();
    } inobj;
};
```

```
int outter :: static_i = 45;

int VAR = 2;

void outter::F::funct()
{
    int local;
    a=1;                //EVAL VAR : Is VAR in global scope
    b=2;
    c=3;
    d=4;
    e=5;
    f=6;

    local = 828;
    int VAR;

    VAR=1;
    static_i=10;
    A::ac=12;
    C::ac=13;
    not_amb=32;

    not_amb=13;
    // Stop here and show:
    // EVAL VAR          : is VAR in local scope
    // EVAL ::VAR        : is VAR in global scope
    // EVAL %LOCALVARS   : see all local vars
```

```

// EVAL *this      : fields of derived class
// EVAL this->f    : show member f
// EVAL f          : in derived class
// EVAL a          : in base class
// EVAL b          : in Virtual Base class
// EVAL c          : static member
// EVAL static_i   : static var made visible
//                                     : by middle-end
// EVAL au         : anonymous union members
// EVAL a=49       :
// EVAL au         :
// EVAL ac         : show ambiguous var
// EVAL A::ac      : disambig with scope op
// EVAL B::ac      : Scope op
// EVAL E<int>::ac : Scope op
// EVAL this       : notice pointer values
// EVAL (E<int>*)this : change
// EVAL (class D *)this : class is optional
// EVAL *(E<int> *)this : show fields
// EVAL *(D *) this  : show fields
}

```

```

int main()
{
    outter obj;
    int outter::F::*mptr = &outter::F::b;
    int i;
    int& r = i;
    obj.inobj.funct();
    i = 777;

    obj.static_i = 2;
    // Stop here
    // EVAL obj.inobj.*mptr : member ptr
    // EVAL obj.inobj.b
    // EVAL i
    // EVAL r
    // EVAL r=1
    // EVAL i
    // EVAL (A &) (obj.inobj) : reference cast
    // EVAL
}

```

Changing Module Optimization and Observability

After a program is created, it might need to be changed to address problems or revised user requirements.

When a program is in production, it is optimized for performance and reduced to its minimum size.

When you debug a program, you need to be able to:

- Observe the behavior of the program as it processes data
- See variable values that might not be visible at higher levels of optimization

When you create a listing view, you add the data required to observe the behavior of the program. See [“Creating a Listing View for Debugging”](#) on page 99.

During a debug session, you can lower the optimization level of a module to display variables accurately as you debug the program, and then raise the level again afterwards to improve the program efficiency as you get the program ready for production.

After a debug session, you can remove module observability to reduce the size of the module.

This topic describes how to:

- [Change optimization levels during a debug session.](#)
- [Remove module observability.](#)

Changing Optimization Levels

Optimizing an object means looking at the compiled code, determining what can be done to make the runtime performance as fast as possible, and making the necessary changes. In general, the higher the optimizing request, the longer it takes to create an object. At runtime, the highly optimized program or service program should run faster than the corresponding non-optimized program or service program.

Example:

This example shows you how to change the optimization level of module T1520IC4 from *FULL to *NONE to allow variables to be displayed and changed when the program is in debug mode. Once debug is complete, you can change the optimization level back to *FULL for improved runtime performance.

1. Enter WRKMOD MODULE(T1520IC1). The Work with Modules display is shown.
2. Select option 5 (Display) to see the attribute values that need to be changed. The Display Module Information display is shown:

```
Display Module Information
Module . . . . . : T1520IC1
Library . . . . . : MYLIB
Detail . . . . . : *BASIC
Module attribute . . . . . : CLE
Module information:
  Module creation date/time . . . . . : 93/09/93 12:00:00
  Source file . . . . . : QACSRC
  Library . . . . . : MYLIB
  Source member . . . . . : T1520IC1
  Source file change date/time . . . . . : 93/08/18 13:31:40
  Owner . . . . . : SMITH
  Coded character set identifier . . . . . : 65535
  Text description . . . . . :
  Creation data . . . . . : *YES
  Intermediate language data . . . . . : *NO
More...
Press Enter to continue.
F3=Exit F12=Cancel
```

Note: In the display shown above, the Creation data value is *YES. This means that the module can be translated again once the optimization level value is changed. If the value is *NO, you must compile the module again in order to change the optimization level.

3. Press the Roll Down key to see more information for the module as shown:

```
Display Module Information
Module . . . . . : T1520IC4
Library . . . . . : MYLIB
Detail . . . . . : *BASIC
Module attribute . . . . . : CLE
Sort sequence table . . . . . : *HEX
Language identifier . . . . . : *JOB RUN
Optimization level . . . . . : *NONE
Maximum optimization level . . . . . : *FULL
Debug data . . . . . : *YES
Compressed . . . . . : *NO
Program entry procedure name . . . . . : _C_ pep
Number of parameters . . . . . : 0 255
Module state . . . . . : *USER
Module domain . . . . . : *SYSTEM
Number of exported defined symbols . . . . . : 1
Number of imported (unresolved) symbols . . . . . : 10
Press Enter to continue.
More...
F3=Exit F12=Cancel
```

4. Check the Maximum Optimization Level value. It may already be at the level you desire. If the module has the creation data, and you want to change the optimization level, press F12 (Cancel). The Work with Modules display is shown.
5. Select option 2 (Change) for the module whose optimization level you want to change. The CHGMOD command prompt is shown.

6. Type over the value specified for the field *Optimize Module*. Changing the module to a lower level of optimization allows you to display, and possibly change, the value of variables while debugging. The following command can be used to lower the optimization level to *NONE on the command prompt but it will NOT be put in the job log.

```
CHGMOD MODULE(MYLIB/T1520IC4) OPTIMIZE(*NONE)
```

7. Do steps “2” on page 134 through “6” on page 135 again for any additional modules you may want to change. Whether you are changing one module or several in the same ILE program, the program creation time is the same because all imports are resolved when the system encounters them.

Note: Imports can be left unresolved using the *UNRSLVREF parameter of the CRTPGM command.

8. Create the program again using the CRTPGM command.

Removing Module Observability

Before you can observe a module, two types of data must be stored with the module.

The two types of data are:

Create Data

Represented by the *CRTDTA value. This data is necessary to translate the code to machine instructions. The module must have this data before you can change the module optimization level.

Debug Data

Represented by the *DBGDTA value.

Both *CRTDTA and *DBGDTA are necessary for a module to be debugged.

You can change the module without re-compiling it only if these two data types are stored with it. After the module is re-compiled, only this data can be removed. After this data is removed, its observability is also removed, and you must recompile the module to replace the data.

Removing all observability reduces the module to its minimum size (with compression). It is not possible to change the module in any way unless you compile the module again. To compile it again, you must have authorization to access the source code.

You can use the CHGMOD command to remove either kind of data from the module.

Example:

Use the following procedure to remove observability from the T1520IC4 program:

1. Enter WRKMOD. The Work with Modules display is shown.
2. Select option 5 (Display) to see the attribute values that need to be changed. The Display Module Information display is shown.

Check the value of the field *Creation data*. If it is *YES, the Create Data exists, and can be removed. If this value is *NO, there is no Create Data to remove. The module cannot be translated again unless you re-create it.

3. Press the Roll Down key to see more information for the module. Check the value of the field *Debug Data*. If it is *YES, the module can be debugged. If it is *NO, the module cannot be debugged unless you compile it again, and include the debug data. Then press F3 to get back to Work with Modules display.
4. Select option 2 (Change) for the module whose observability you want to change. The CHGMOD command prompt is shown.
5. Type over the value specified for the *Remove Observable Info* prompt. The following command appears in the job log for the Change Module command after the Enter key is pressed.

```
CHGMOD MODULE(MYLIB/T1520IC4) RMVOBS(*ALL)
```

6. You can ensure that the module is created again by changing the value of the *Force Module Recreation* parameter to *YES.

Note: This parameter is not required simply because the optimization level is changed. A change in the optimization level typically results in module re-creation *unless the Create Data has been removed*. If you want the program to be translated again *after* removing the debug data, without changing the optimization level, you must use the *Force Module Recreation* parameter.

7. Do steps “2” on page 135 through “5” on page 135 again for any additional modules you want to change. Whether you are changing one module or several in the same ILE program, the program creation time is the same because all imports are resolved when the system encounters them.

Note: Imports can be left unresolved using the *UNRSLVREF parameter of the CRTPGM command. Program creation time is the same.

8. Create the ILE program again by using the CRTPGM command.

Performing I/O Operations

This topic describes how to:

- [Use ILE C/C++ stream and record I/O functions with files](#)
- [Use ILE C/C++ stream I/O functions with the integrated file system \(IFS\)](#)

Using ILE C/C++ Stream and Record I/O Functions with IBM i files

The ILE C/C++ compiler allows your program to process stream files as text stream files or as binary stream files. See [“File Control Structure of Text Streams and Binary Streams”](#) on page 142.

This topic describes:

- [ILE C Record I/O Functions](#)
- [IBM i files](#)
- [File control structure of text streams and binary streams](#)
- [I/O processes for text stream files](#)
- [I/O processes for binary stream files](#)
- [Open feedback area](#)
- [I/O feedback area](#)
- [How to use Session Manager](#)

ILE C Record I/O Functions



The ILE C library provides a set of extensions to the ISO C definition for I/O. This set of extensions, referred to as *record I/O*, allows your program to perform I/O operations one record at a time.

The ILE C record I/O functions work with all the file types that are supported on the IBM i platform.

Each file that is opened with `_Ropen()` has an associated structure of type `_RFILE`. The `<recio.h>` header file defines this structure.



Attention: Unpredictable results may occur if you attempt to change this structure.

Different open modes and keyword parameters apply to the different IBM i data management file types. For information about each file type and how to open a record file using `_Ropen()`, see:

- [“Using Externally Described Files in a Program”](#) on page 179
- [“Using Database Files and Distributed Files in a Program”](#) on page 197
- [“Using Device Files in a Program”](#) on page 211

Note: There is no equivalent function provided by the C++ runtime library.

Stream Buffering



Three buffering schemes are defined for ISO standard C streams. They are:

- *Unbuffered* - characters are intended to appear from the source or at the destination, as soon as possible. The ILE C compiler does not support unbuffered streams.
- *Fully buffered* - characters are transmitted to and from a file one block at time, after the buffer is full. The ILE C compiler treats a block as the size of the system file's record.

- *Line buffered* - characters are transmitted to and from a file, as a block, when a new-line control character (`\n`) is encountered.

The ILE C compiler supports fully-buffered and line-buffered streams in the same manner, because a block and a line are equal to the record length of the opened file.

Note: The `setbuf()` and `setvbuf()` functions do not allow you to control buffering and buffer size when using the system.

Dynamic Stream File Creation

Dynamic file creation for text stream files is the same as specifying:

```
CRTSRCPF FILE(filename) RCDLEN(recLn)
```

Dynamic file creation for binary stream files is the same as specifying:

```
CRTPF FILE(filename) RCDLEN(recLn)
```

The length that is specified on the `lrecL` parameter of `fopen()` is used for the record length of the file that is created, with the following exceptions:

- If you do not specify a record length when you open a text file, then a source physical file with a record length of 266 is created.
- If you do not specify a record length when you open a binary or record file, then a physical file with a record length of 80 is created.
- If you specify a record length of zero (`lrecL=0`) when you open a text file, then a source physical file with a record length of 266 is created.
- If you specify a record length of zero (`lrecL=0`) when you open a binary file, then a physical file with a record length of 80 is created.
- If the `lrecL` parameter is not specified for program-described files, then the record length that is specified on the `CRTPRTG`, or `CRTPRTF` is used. This length has a default value of 132, and if specified must be a minimum of 1.

Note: To use the source entry utility (SEU) to edit source files, specify an `lrecL` value of 240 characters or less on `fopen()`.

Open Modes for Dynamically Created Stream Files

If you specify the mode when opening a file, and if the file you specified does not already exist, the IBM i platform automatically creates the file.

- If you are using binary mode, a physical database file is created.
- If you are using text mode, a source physical file is created.

If the file exists, but the member does not, the IBM i platform adds the member to the file.

If you do not specify a library name when you open the file, the database file is dynamically created in library `QTEMP`. If you do not specify a member name, a member is created with the same name as the file.

Standard I/O Text Stream Files (<stdio.h>)

When a program that includes the `<stdio.h>` file starts, three text streams are defined:

- Standard input (`stdin`) reads input from the terminal.
- Standard output (`stdout`) writes output to the terminal.
- Standard error (`stderr`) writes diagnostic output to the terminal.

Streams `stdin`, `stdout`, and `stderr` are implicitly opened the first time they are used.

- Stream `stdin` is opened with `fopen("stdin", "r")`.

- Stream stdout is opened with `fopen("stdout", "w")`.
- Stream stderr is opened with `fopen("stderr", "w")`.

Note: These streams are not real IBM i files, but are simulated as files by the ILE C library functions. By default, they are directed to the terminal session.

Overriding Standard Output to the Terminal

The stdin, stdout, and stderr streams can be associated with other devices using the IBM i override commands on the files stdin, stdout, and stderr respectively. If stdin, stdout, and stderr are used, and a file override is present on any of these streams prior to opening the stream, then the override takes effect, and the I/O operation may not go to the terminal.

If stdout or stderr are used in a non-interactive job, and if there are no file overrides for the stream, then the ILE C compiler overrides the stream to the printer file QPRINT. Output prints or spools for printing instead of displaying at your workstation.

Allowing a Program to Re-Read an Input File with QINLINE Specified

If stdin is specified (or the default accepted) for an input file that is not part of an interactive job, then the QINLINE file is used. You cannot re-read a file with QINLINE specified, because the database reader will treat it as an unnamed file, and therefore it cannot be read twice. You can avoid this by issuing an override. If you are reading characters from stdin, pressing F4 triggers the runtime to end any pending input and to set the EOF indicator on. Pressing F3 is the same as calling `exit()` from your ILE C/C++ program.

If stdin is specified in batch and has no overrides associated with it, then QINLINE will be used. If stdin has overrides associated with it, then the override is used instead of QINLINE.

Note: You can also use `freopen()` to reopen text streams. The stdout and stderr streams can be reopened for printer and database files. The stdin stream can be overridden only with database files. *Using `freopen()` to redirect stdin/stdout/stderr from/to an IFS stream file is not supported.*

IBM i Files

An ILE C stream file or record file is the same as an IBM i file. System files are also called *file objects*. Each IBM i file or file object is differentiated and categorized by information that is stored within it. Each file has its own set of unique characteristics, which determine how the file can be used and what capabilities it provides. This information is called the *file description*.

The file description also contains the file's characteristics, details on how the data associated with the file is organized into records, and how the fields are organized within these records. Whenever a file is processed, the operating system uses the file description. Data is created and accessed on the system through file objects.

IBM i File Types

The IBM i files are listed:

- *Database files* store data on the IBM i platform.
- *Device files* provide access to externally attached devices such as: displays, printers, tapes, and diskettes.
- *Intersystem communications function (ICF) files* define the layout of the data sent and received between two application programs on different systems. This file links the configuration objects that are used to communicate with the remote system
- *Save files* save data in a format that is used for backup and recovery purposes.
- *Distributed (DDM) files* access data on remote systems.

Stream Files and ILE C I/O Operations

C The C International Standard defines a C language *stream file* as a sequence of data that is read and written one character at a time. All I/O operations in ISO C are stream operations.

On the IBM i platform:

- A stream is a continuous string of characters.
- All files are made up of records.
- All I/O operations at the operating system level are carried out a record at a time, using operations.

The ILE C/C++ runtime library allows your program to process stream files as text stream files or as binary stream files. Text stream files process one character at a time. Binary stream files process either one character at a time or one record at a time.

Because the IBM i platform carries out I/O operations one record at a time, the ILE C/C++ library simulates stream file processing with IBM i records. Although the ILE C/C++ library logically handles I/O one character at a time, the actual I/O that is performed by the operating system is done one record at a time.

Avoiding Positioning Problems in the File

Because the operating system carries out I/O operations one record at a time, using system commands such as OPNQRYP together with stream I/O operations on the same file may cause positioning problems in the file your program is processing.

Caution:

- Do not mix the use of ILE C/C++ extensions for record I/O and stream file functions on the same file as unpredictable results can occur.
- Avoid using system commands that logically work with records instead of characters in programs that contain stream I/O operations.

Using the fopen() Function

The format of fopen() is:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

The *mode* variable is a character string that consists of an open mode which may be followed by keyword parameters. The open mode and keyword parameters must be separated by a comma or one or more blank characters.

Note: For information about the recIn parameter, see [“Dynamic Stream File Creation”](#) on page 138.

Using the open() member Function

C++

Create an input, output, or input/output file stream and then link to a file. Use the open() member function of the file stream class to link a file stream with a file. The format of the open() member function is:

```
void ifstream::open(const char *filename, openmode mode=ios::in);
void ofstream::open(const char *filename, openmode mode=ios::out|ios::trunc);
void fstream::open(const char *filename, openmode mode);
```

IBM i File Naming Conventions

The _Ropen() and fopen() functions that refer to IBM i files require a *file name*. This file name must be a null-ended string.

The syntax of a filename is:



library-name

Enter the name of the library that contains the file. If you do not specify a library, the system searches the job's library list for the file.

filename

Enter the name of the file. This is a required parameter.

member-name

Enter the name of the file member. If you do not specify a member name, the first member (*FIRST) is used.

Note: If you specify *ALL for *member-name* when using `fopen()` and `_Ropen()`, multi-member processing occurs.

All characters specified for *library-name*, *filename*, or *member-name* are folded to uppercase unless you surround the string by the back slash and quotation mark (\") control sequence. This allows you to specify the IBM i quoted names. For example:

```
"\"tstlib\"/tstfile(tstnbr)"
Library is:  "tstlib"
File is:    TSTFILE
Member is:  TSTMBR
```

If you surround the filename, library name, or member name in double quotation marks and the name is a normal name, the double quotation marks are discarded by the ILE C/C++ compiler. A normal name is any file, library, or member name with the following characters:

- Uppercase characters
- Numeric values
- \$ (hexadecimal value 0x5B)
- @ (hexadecimal value 0x7C)
- # (hexadecimal value 0x7B)
- _ (hexadecimal value 0x6D)
- . (hexadecimal value 0x4B)

The following characters cannot appear anywhere in your filenames, library names, or member names:

Incorrect Character

Hexadecimal Representation

- (0x4D
- * 0x5C
-) 0x5D
- / 0x6I
- ? 0x6F
- ' 0x7D

"
 0x7F
 (blank)
 0x40

Note: "()/\" can be used in quoted filenames.

File Control Structure of Text Streams and Binary Streams

Both text streams and binary streams map to records in IBM i files. They can be processed one character at a time or one record at a time.

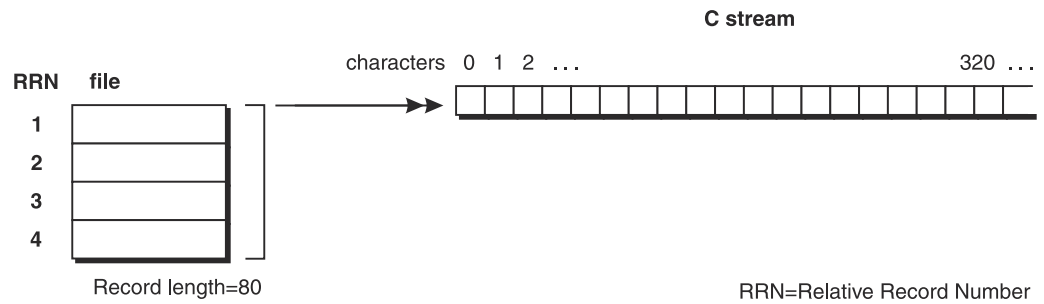


Figure 77. IBM i Records Mapping to an ILE C Stream File

Each text stream file and each binary stream file is represented by a file control structure of type FILE. This structure is defined in the <stdio.h> header file.



Attention: Unpredictable results may occur if you attempt to change the file control structure.

Table 12. Comparison of IBM i Text Streams and Binary Stream File Processing		
	Text Streams	Binary Streams
Definition	An ordered sequence of characters that are composed of lines. Each line consists of zero or more characters and ends with a new-line character.	A sequence of characters that has a one-to-one correspondence with the characters stored in the associated IBM i file. On the IBM i platform, the length of a binary stream file is a multiple of the record length.

Table 12. Comparison of IBM i Text Streams and Binary Stream File Processing (continued)

	Text Streams	Binary Streams
Impact of I/O Processing	<p>The IBM i may add, alter, or delete some special characters during input or output. Therefore, there may not be a one-to-one correspondence between the characters written to a text stream and characters read from the same text stream. Data read from a text stream is equal to data written to the text stream if all of the following are true:</p> <ul style="list-style-type: none"> • The data consists of printable characters, horizontal tab, vertical tab, new-line character, or form-feed control characters. • No new-line character is immediately preceded by a space (blank) character. • The last character in a stream is a new-line character. • The lines that are written to a file do not exceed the record length of the file. 	<p>Character translation is <i>not</i> performed on binary streams. When data is written to a binary stream, it is the same when it is read back later.</p> <p>Note: New-line characters have no special significance in a binary stream.</p>
End-of-File Processing	<p>When a file is closed, an implicit new-line character is appended to the end of the file unless a new-line character is already specified.</p>	<p>When a file is closed, the last record in the file is padded with nulls (hexadecimal value 0x00) to the end of the record.</p>

I/O Processes for Text Stream Files

This section describes how to:

- [open text stream files](#)
- [write text stream files](#)
- [read text stream files](#)
- [update text stream files](#)

Opening Text Stream Files

To open an IBM i file as a text stream file, use `fopen()` with one of the following modes:

- `r`
- `r+`
- `w`
- `w+`
- `a`
- `a+`

Note:

1. The number of files that can be simultaneously opened by `fopen()` depends on the amount of the system storage available.
2. The `fopen()` function open modes also apply to the `freopen()` function.

3. If the text stream file contains deleted records, the deleted records are skipped by the text stream I/O functions.

The valid keyword parameters are:

- `lrecl`
- `ccsid`
- `recfm` (F, FA, and FB only)

If you specify a mode or keyword parameter that is not valid on `fopen()`, `errno` is set to `EBADMODE`, and `NULL` is returned.

Example:

The following example illustrates how to open a text stream file. Library `MYLIB` must exist. The file `TEST` is created for you if it does not exist. The mode `w+` indicates that if `MBR` does not exist, it is created for update. If it does exist, it is cleared.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    /* Open a text stream file.                */
    /* Check to see if it opened successfully */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "w+" ) ) == NULL )
    {
        printf ( "Cannot open MYLIB/TEST(MBR)\n" );
        exit ( 1 );
    }

    printf ( "Opened the file successfully\n" );

    /* Perform some I/O operations.           */

    fclose ( fp );
    return 0;
}
```

Figure 78. ILE C Source to Open an ILE C Text Stream File

Note: You can read, write to, or update any text stream file that is open for processing.



To open an IBM i file as a text stream file, use the `open()` member function with the following modes:

- `ios::app`
- `ios::ate`
- `ios::in`
- `ios::out`
- `ios::trunc`

Writing Text Stream Files

During a write operation, a new-line character in the buffer causes the remainder of the record written to the text stream file to be padded with blank characters (hexadecimal value `0x40`). The new-line character itself is discarded.

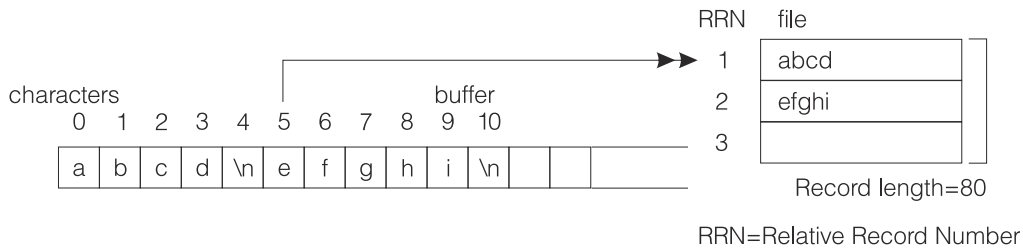


Figure 79. Writing to a Text Stream File

If the number of characters being written in the buffer exceeds the record length of the file, the data written to the file is truncated, and `errno` is set to `ETRUNC`.

Example:

The following example illustrates how to write to a text stream file.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buf[12] = "abcd\nefghi\n";
    FILE *fp;
    /* Open a text file for writing. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "w" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write characters to the file. */
    fputs ( buf, fp );
    /* Close the text file. */
    fclose ( fp );
    return 0;
}
```

Figure 80. ILE C Source to Write Characters to a Text Stream File

Reading Text Stream Files

During a read operation from a text stream file, all the trailing blank characters (hexadecimal value `0x40`) in the record that are read from the file into a buffer are ignored. A new-line character is inserted after the last non-blank.

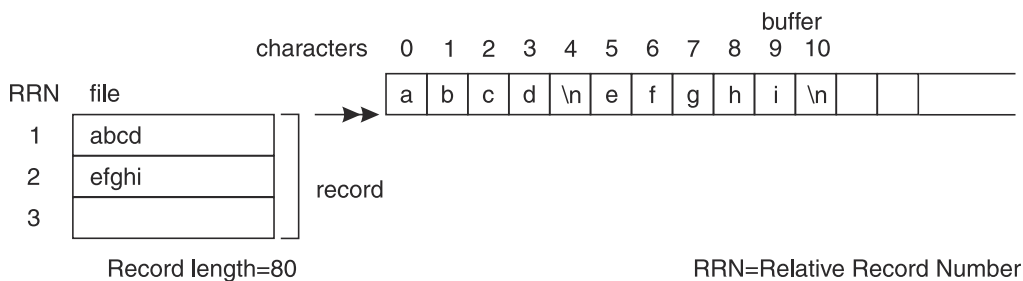


Figure 81. Reading from a Text Stream File

Example:

The following example illustrates how to read from a text stream file.

```

#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buf[12];
    char *result;
    FILE *fp;
    /* Open an existing text file for reading. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "r" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Read characters into the buffer. */

    result = fgets ( buf, sizeof(buf), fp );
    printf("%10s", result);
    result = fgets ( buf+5, sizeof(buf), fp );
    printf("%10s", result);

    fclose ( fp );
    return 0;
}

```

Figure 82. ILE C Source to Read Characters from a Text Stream File

Updating Text Stream Files

During an update operation to a text stream file, if the number of characters being written to the file exceeds the record length of the file, trailing characters in the record are truncated and `errno` is set to `ETRUNC`.

If the data being written to the text stream file is shorter than the record length being updated, and the last character of the data being written is a new-line character, then the record is updated and the remainder of the record is filled with blank characters. If the last character of the data being written is not a new-line character, the record is updated and the remainder of the record remains unchanged.

I/O Process for Binary Stream Files

This section describes how to:

- [Open binary stream files, one character at a time](#)
- [Write binary stream files, one character at a time](#)
- [Read binary stream files, one character at a time](#)
- [Update binary stream files, one character at a time](#)
- [Open binary stream files, one record at a time](#)
- [Write binary stream files, one record at a time](#)
- [Read binary stream files, one record at a time](#)

Opening Binary Stream Files (character at a time)

To open an IBM i file as a binary stream file for character-at-a-time processing, use `fopen()` with any of the following modes:

- `rb`
- `r+b`
- `rb+`
- `wb`
- `w+b`
- `wb+`
- `ab`

- a+b
- ab+

Note:

1. The number of files that can be simultaneously opened by `fopen()` depends on the size of the system storage available.
2. The `fopen()` function open modes also apply to the `freopen()` function.
3. If the binary stream file contains deleted records, the deleted records are skipped by the binary stream I/O functions.

The valid keyword parameters are:

- `blksize`
- `lrecl`
- `recfm`
- `type`
- `commit`
- `ccsid`
- `arrseq`
- `indicators`

If you specify the `type` parameter the value must be memory for binary stream character-at-a-time processing.

Note: The memory parameter identifies this file as a memory file that is accessible only from C programs. This parameter is the default and is ignored.

If you specify a mode or keyword parameter that is not valid on `fopen()`, `errno` is set to `EBADMODE`.

Example:

The following example illustrates how to open a binary stream file.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    /* Open an existing binary file.          */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "wb+" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    printf ("Opened the file successfully\n");

    /* Perform some I/O operations.          */

    fprintf (fp, "Hello, world");

    fclose ( fp );
    return 0;
}
```

Figure 83. ILE C Source to Open a Binary Stream File

Note: You can read, write to, or update any binary stream files that are open for character-at-a time processing.



To open an IBM i file as a binary stream file for character-at-a-time processing, use the `open()` member function with `ios::binary` as well as any of the following modes:

- ios::app
- ios::ate
- ios::in
- ios::out
- ios::trunc

Writing Binary Stream Files (character at a time)

If you write data to a binary stream processed one character at a time, and the size of the data is greater than the current record length, then the excess data is written to the current record up to its record size and the remaining data is written to the next record in the file.

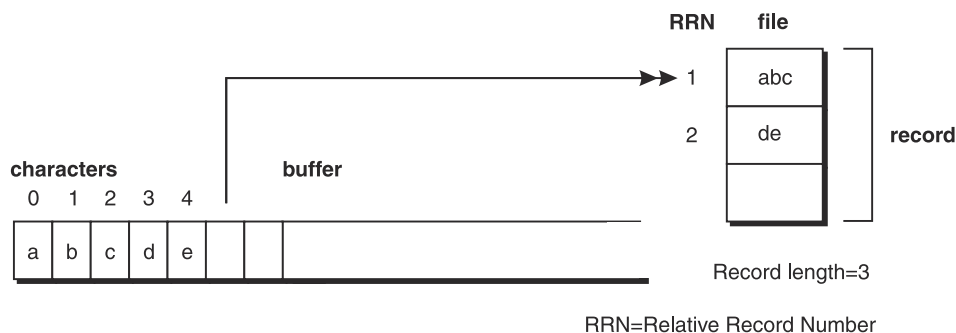


Figure 84. Writing to a Binary Stream File One Character at a Time

Example:

The following example illustrates how to write to a binary stream by character.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = {'a', 'b', 'c', 'd', 'e'};
    /* Open an existing binary file for writing. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "wb" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 5 characters from the buffer to the file. */
    fwrite ( buf, 1, sizeof(buf), fp );

    fclose ( fp );
    return 0;
}
```

Figure 85. ILE C Source to Write Characters to a Binary Stream File

Reading Binary Stream Files (character at a time)

During a read operation from a binary stream that is processed a character at a time, if the length of the data being read is greater than the record length of the file, then data is read from the next record in the file.

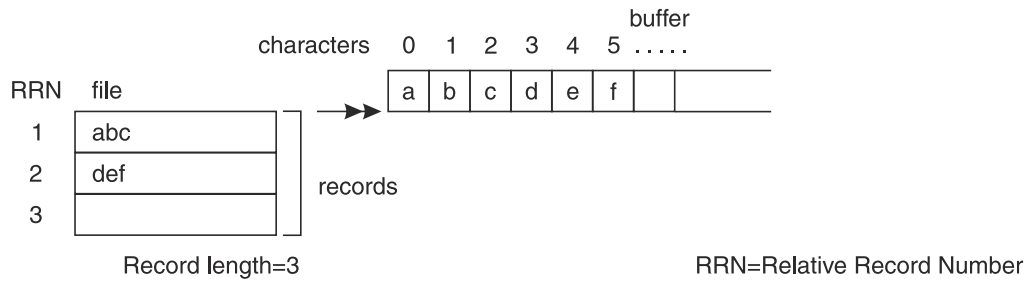


Figure 86. Reading from a Binary Stream File One Character at a Time

Example:

The following illustrates how to read from a binary stream file by character.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[6];
    /* Open an existing binary file for reading. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "rb" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Read characters from the file to the buffer. */

    fread ( buf, 1, sizeof(buf), fp );
    printf ( "%6s\n", buf );

    fclose ( fp );
    return 0;
}
```

Figure 87. ILE C Source to Read Characters from a Binary Stream File

Updating Binary Stream Files (character at a time)

If the amount of data being updated exceeds the current record length, then the excess data updates the next record. If the current record is the last record in the file, a new record is created.

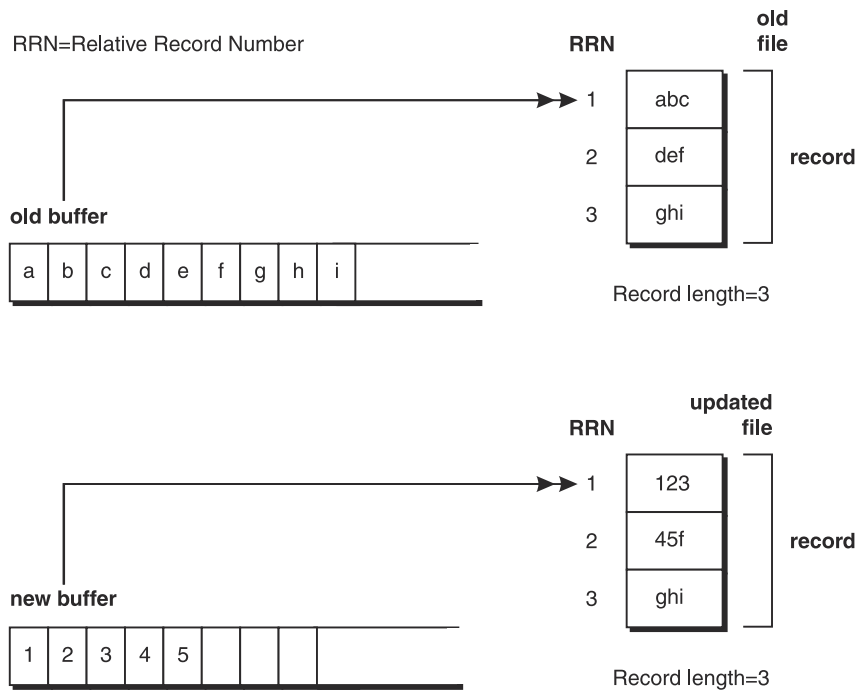


Figure 88. Updating a Binary Stream File with Data Longer than Record Length

Example:

The following example illustrates updating a binary stream file with data that is longer than the record length.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = "12345";
    /* Open an existing binary file for updating. */
    if (( fp = fopen ( "QTEMP/TEST(MBR)", "rb+" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 5 characters from the buffer to the file. */
    fwrite ( buf, 1, sizeof(buf), fp );

    fclose ( fp );
    return 0;
}
```

Figure 89. ILE C Source to Update a Binary Stream File with Data Longer than the Record Length

If the amount of data being updated is shorter than the current record length, then the record is partially updated and the remainder is unchanged.

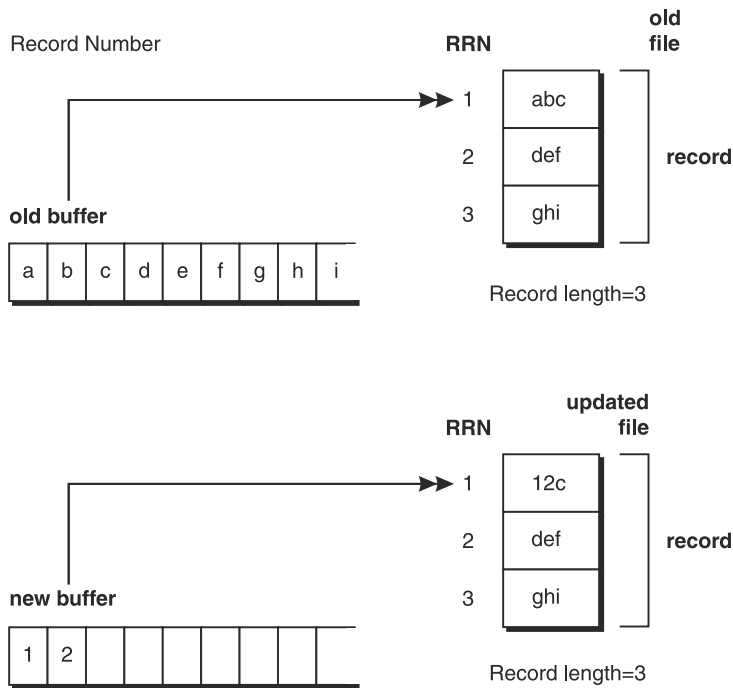


Figure 90. Updating a Binary Stream File with Data Shorter than Record Length

Example:

The following example illustrates updating a binary stream file with data that is shorter than the record length.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[2] = "12";
    /* Open an existing binary file for updating. */
    if (( fp = fopen ( "QTEMP/TEST(MBR)", "rb+" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 2 characters from the buffer to the file. */
    fwrite ( buf, 1, sizeof(buf), fp );
    fclose ( fp );
}
```

Figure 91. ILE C Source to Update a Binary Stream File with Data Shorter than the Record Length

Opening Binary Stream Files (record at a time)

To open an IBM i file as a binary stream file for record-at-a-time processing, use `fopen()` with any of the following modes:

- `rb`
- `r+b`
- `rb+`
- `wb`
- `w+b`
- `wb+`
- `ab`

- a+b
- ab+

Note:

1. The number of files that can be simultaneously opened by `fopen()` depends on the size of the system storage available.
2. The `fopen()` open modes also apply to `freopen()`.
3. If the binary stream file contains deleted records, the deleted records are skipped by the binary stream I/O functions.
4. The file must be opened with the type set to record.

The valid keyword parameters are:

- `blksize`
- `recfm`
- `commit`
- `arrseq`
- `lrecl`
- `type`
- `ccsid`
- `indicator`

If you specify a mode or keyword parameter that is not valid on `fopen()` function, `errno` is set to `EBADMODE`.

Only `fread()` and `fwrite()` can be used for binary stream files opened for record-at-a-time processing.

C++

To open an IBM i file as a binary stream file for record-at-a-time processing, use the `open()` function with `ios::binary` as well as any of the following modes:

- `ios::app`
- `ios::ate`
- `ios::in`
- `ios::out`
- `ios::trunc`

Writing Binary Stream Files (record at a time)

If you write data to a binary stream processed one record at a time, and the product of `size` and `count` (parameters of `fwrite()`) is greater than the record length, then only the data that fits in the current record is written and `errno` is set to `ETRUNC`.

If the product of `size` and `count` is less than the actual record length, the current record is padded with blank characters and `errno` is set to `EPAD`.

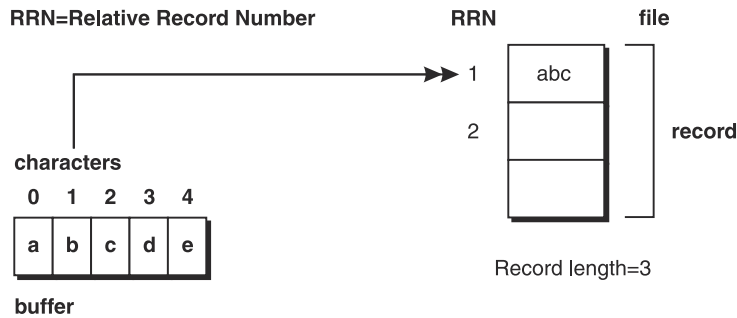


Figure 92. Writing to a Binary Stream File One Record at a Time

Only `fwrite()` is valid for writing to binary stream files opened for record-at-a-time processing. All other output and positioning functions fail, and `errno` is set to `ERECIO`.

Example:

The following example illustrates how to write to a binary stream file by record.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[5] = {'a', 'b', 'c', 'd', 'e'};
    /* Open an existing binary file for writing. */
    if ((fp = fopen ( "MYLIB/TEST(MBR)", "wb,type=record,lrcl=3" ))==NULL)
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Write 3 characters from the buffer to the file. */

    fwrite ( buf, 1, sizeof(buf), fp );

    fclose ( fp );
    return 0;
}
```

Figure 93. ILE C Source to Write to a Binary Stream File by Record

Reading Binary Stream Files (record at a time)

If you read data from a binary stream processed one record at a time, and the product of size and count (parameters of `fread()`) is greater than the record length, then only the data in the current record is read into the buffer. The `fread()` function returns a value indicating that there is less data in the buffer than was specified.

If the product of size and count is less than the actual record length, `errno` is set to `ETRUNC` to indicate that there is data in the record that was not copied into the buffer.

This figure illustrates how only the current record is read into the buffer, when the product of size and count is greater than the record length.

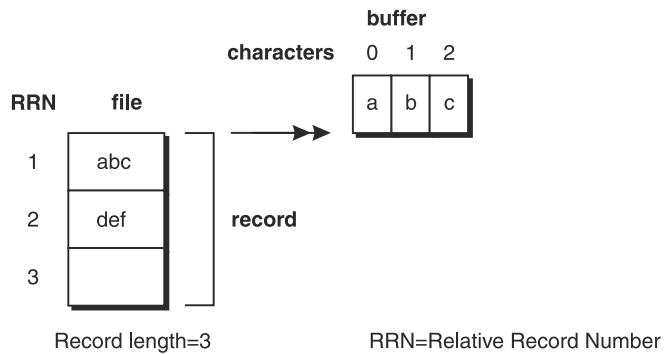


Figure 94. Reading from a Binary Stream File a Record at a Time

Only `fread()` function is valid for reading binary stream files opened for record-at-a-time processing. All other input and positioning functions fail, and `errno` is set to `ERECIO`.

Example:

The following example illustrates how to read a binary stream a record at a time.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    char buf[6];
    /* Open an existing binary file for reading a record at a time. */
    if (( fp = fopen ( "MYLIB/TEST(MBR)", "rb, type=record" ) ) == NULL )
    {
        printf ( "Cannot open file\n" );
        exit ( 1 );
    }
    /* Read characters from the file to the buffer. */
    fread ( buf, sizeof(buf), 1, fp );
    printf ( "%6s\n", buf );

    fclose ( fp );
    return 0;
}
```

Figure 95. ILE C Source to Read from a Binary Stream File by Record

Open Feedback Area

The open feedback area is part of the open data path that contains information about the open file that is associated with that open data path. You can assign a pointer to this information by using the `_Ropnfbk()` function. The structure that maps to the open feedback area can be that is found in the `<xxfdbk.h>` header file.

I/O Feedback Area

The I/O feedback area is a part of the open data path for the file that is updated after each successful non-blocked I/O operation. If record blocking is taking place, the I/O feedback is updated after each block of records is transferred between your program and the system.

The I/O feedback consists of two parts: one part that is common to all file types, and one part that is specific to the type of file.

To assign a pointer to the common part of the I/O feedback area, use the `_Riofbk()` function. To assign a pointer to the part of the I/O feedback area that is specific to the type of file, add the offset contained in the `file_dep_fb_offset` field of the common part to a pointer to the common part.

Note: The offset is in bytes, so you need to cast the pointer (`char *`) to the common part to a pointer to character when performing the pointer arithmetic. The structures that map to the I/O feedback areas are the structures contained in the `<xxfdbk.h>` header file.

Using Session Manager

ILE C stream I/O functions that output information to the display are defined through the Dynamic Screen Manager (DSM) session manager APIs.

Obtaining the Session Handle

You can obtain the session handle for the C/C++ session and then use the DSM APIs to manipulate that session. The session handle is supplied through `_C_Get_Ssn_Handle()` in `<stdio.h>`.

You can write a simple C program to clear the C session using the DSM `QsnClrScl` API, as shown in the following example:

```
#include <stdio.h>
#include "qsnapi.h"
void main (void)
{
    QsnClrScl(_C_Get_Ssn_Handle(), '0', NULL);
}
```

Figure 96. Simple C Program to Clear a C Session

Using Session Manager APIs

You can use the DSM APIs to perform any operation that is valid with a session handle, which includes the window services APIs and many of the low-level services.

For example:

- You can display the session using a combination of the `QsnStrWin`, `QsnDspSsnBot`, and `QsnReadSsnDta` APIs, but it is simpler in this case to simply write a program that contains a `getc()`.
- You can use the `QsnRtvWinD` and `QsnChgWin` APIs to change the C/C++ session from the default full-screen window to a smaller window.

Example: Using an ILE Bindable API to Display a DSM Session

The following example shows you how to call a Dynamic Screen Manager (DSM) ILE bindable API to display a DSM session. This DSM session echoes back data that you enter during the DSM session.

Instructions

1. To create module T1520API using the source shown in [“Code Samples”](#) on page 156, enter:

```
CRTCMOD MODULE(MYLIB/T1520API) SRCFILE(QCPPL/ QACSRC) OUTPUT(*PRINT)
```

2. To create program T1520API, enter:

```
CRTPGM PGM(MYLIB/T1520API) MODULE(MYLIB/T1520API) BNDDIR(QSNAPI)
```

The CRTPGM command creates the program T1520API in library MYLIB.

3. To run the program T1520API, enter:

```
CALL PGM(MYLIB/T1520API)
```

The output is as follows:


```

Q_Bin4      botline_len = sizeof(BOTLINE) - 1;
Q_Bin4      sess_desc_length = sizeof(Qsn_Ssn_Desc_T) +
              botline_len;
Q_Bin4      bytes_read;

/* Initialize Session Descriptor API. */

QsnInzSsnD( sess_desc, sess_desc_length, NULL);

/* Initialize Window Descriptor API. */

QsnInzWinD( &win_desc, win_desc_length, NULL);

sess_desc->cmd_key_desc_line_1_offset = sizeof(Qsn_Ssn_Desc_T);
sess_desc->cmd_key_desc_line_1_len = botline_len;
memcpy( storage.buffer, botline, botline_len );

sess_desc->cmd_key_desc_line_2_offset = sizeof(Qsn_Ssn_Desc_T) +
              botline_len;

/* Set up the session type. */

sess_desc->EBCDIC_dsp_cc = '1';
sess_desc->scl_line_dsp = '1';
sess_desc->num_input_line_rows = 1;
sess_desc->wrap = '1';

/* Set up the window size. */

win_desc.top_row      = 3;
win_desc.left_col     = 3;
win_desc.num_rows    = 13;
win_desc.num_cols    = 45;

/* Create a window session. */

sess_desc->cmd_key_action[2] = F3Exit;
session1 = QsnCrtSsn( sess_desc, sess_desc_length,
                    NULL, 0,
                    '1',
                    &win_desc, win_desc_length,
                    NULL, 0,
                    NULL, NULL);

if(input_buffer == 0)
{
    input_buffer = QsnCrtInpBuf(100, 50, 0, NULL, NULL);
}
for (;;)
{

/* Echo lines until F3 is pressed. */

    QsnReadSsnDta(session1, input_buffer, NULL, NULL);
    if (QsnRtvReadAID(input_buffer, NULL, NULL) == QSN_F3)
    {
        break;
    }
}
}

```

Note:

1. The prototypes for the DSM APIs are in the <qsnssess.h> header file.
2. The #pragma argument (API, OS, nowiden) directive is specified for each API. This ensures that any value argument is passed by value indirectly.

Using ILE C/C++ Stream Functions with the IBM i Integrated File System

Read this section to learn how to open, write, read, and update text and binary stream files through the IBM i integrated file system (IFS).

IFS provides a common interface to store and operate on information in stream files. Examples of stream files are PC files systems, files systems in UNIX, LAN server files systems, and folders.

Note: The ILE C/C++ IFS-enabled stream I/O functions are defined through the integrated file system. You need to be familiar with the integrated file system to use the ILE C/C++ stream I/O function. Seven file systems comprise the integrated file system. Depending on your application and environment, you may use several of the file systems. If you have existing applications that use IBM i files, you need to understand the limitations of the QSYS.LIB file system. If you have new applications, you can use the other file systems which do not have the QSYS.LIB file handling restrictions. See [“The Integrated File System \(IFS\)”](#) on page 158 section for information on each file system.

This topic describes:

- [The integrated file system \(IFS\) components](#)
- [How to enable IFS stream I/O](#)
- [Stream files, text streams, and binary streams](#)
- [How to open text stream and binary stream files](#)
- [How to store data as a text stream or binary stream](#)
- [Useful information for using IFS files](#)

The Integrated File System (IFS)

A *file system* provides the support that allows applications to access specific segments of storage that are organized as logical units. These logical units are files, directories, libraries, and objects.

There are seven file systems in the integrated file system:

- root (/)
- Open Systems (QOpenSys)
- Library (QSYS.LIB)
- Document Library Services (QDLS)
- LAN Server/400 (QLANSrv)
- Optical Support (QOPT)
- File Server (QFileSvr.400)

[Figure 97 on page 159](#) illustrates these file systems.

Users and application programs can interact with any of the file systems through a common *integrated file system (IFS) interface*. This interface is optimized for input/output of stream data, in contrast to the record input/output that is provided through the interfaces. The common integrated file system interface includes a set of user interfaces (commands, menus, and displays) and application program interfaces (APIs).

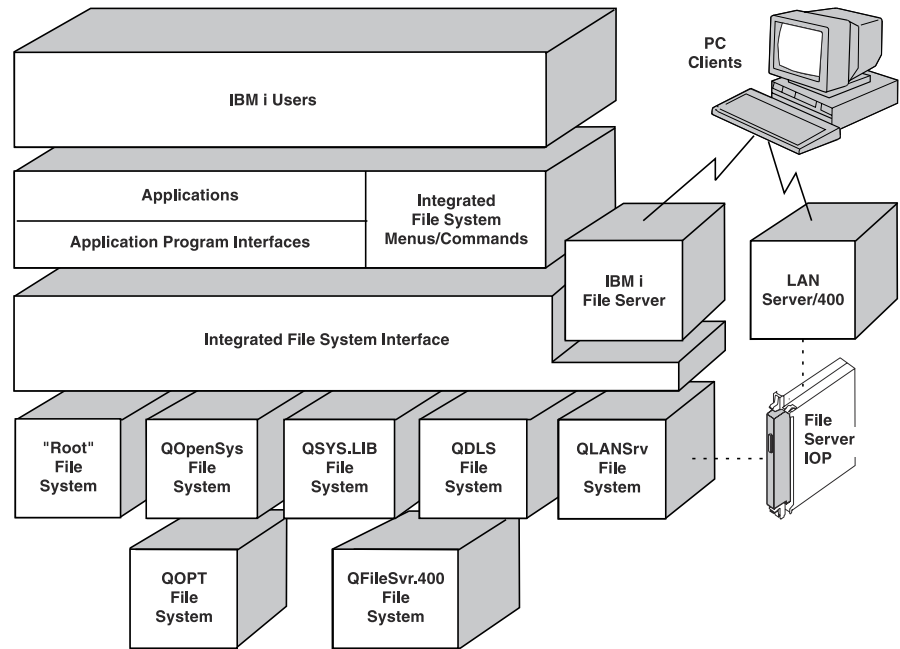


Figure 97. The Integrated File System Interface

root(/) File System

The root (/) file system is designed to take full advantage of the stream file support and hierarchical directory structure of the integrated file system. It has the characteristics of the DOS and OS/2 file systems.

User Access

The root (/) file system can be accessed only through the integrated file system interface. You work with the root (/) file system using integrated file system commands, user displays, or ISO stream I/O functions and system APIs.

Path Names

This file system preserves the same uppercase and lowercase form in which object names are entered, but no distinction is made between uppercase and lowercase when the system searches for names.

- Path names have the following form:

```
Directory/Directory/ . . . /Object
```

- Each component of the path name can be up to 255 characters long. The path can be up to 16 megabytes.
- There is no limit on the depth of the directory hierarchy other than program and space limits.
- The characters in names are converted to Universal Character Set 2 (UCS2) Level 1 form when the names are stored.

Open Systems (QOpenSys) File System

The Open Systems (QOpenSys) file system is designed to be compatible with UNIX-based open system standards, such as POSIX and XPG. Like the root (/) file system, it takes advantage of the stream file and directory support provided by the integrated file system. In addition, it supports case-sensitive object names.

User Access

QOpenSys can be accessed only through the integrated file system interface. You work with QOpenSys using integrated file system commands, user displays, or ISO stream I/O functions and system APIs.

Path Names

Unlike the QSYS.LIB, QDLS, QLANSrv, and root (/) file systems, the QOpenSys file system distinguishes between uppercase and lowercase characters when searching object names.

The path names, link support, commands, displays and ISO stream I/O functions and system APIs are the same as defined under the root (/) file system.

Library (QSYS.LIB) File System

The library (QSYS.LIB) file system supports the IBM i library structure. It provides access to database files and all of the other IBM i object types that are managed by the library support.

The QSYS.LIB file system maps to the IBM i file system. For example, the path /qsys.lib/qsysinc.lib/h.file/stdio.mbr refers to the file member STDIO, in the file H, in library QSYSINC, within the root library QSYS.

File Handling Restrictions

There are some limitations in using the integrated file system facilities:

- Logical files are not supported.
- The only types of physical files that are supported are program-described files that contain a single field, and source physical files that contain a single text field.
- Byte-range locking is not supported.
- If any job has a database file member open, only one job is given write access to that file at any time; other jobs are allowed only read access.

Path Names

In general, the QSYS.LIB file system does not distinguish between uppercase and lowercase names of objects. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

However, if the name is enclosed in quotation marks, the case of each character in the name is preserved. The search is sensitive to the case of characters in quoted names.

Each component of the path name must contain the object name followed by the object type. For example:

```
/QSYS.LIB/QGPL.LIB/PRT1.OUTQ  
/QSYS.LIB/PAYROLL.LIB/PAY.FILE/TAX.MBR
```

The object name and object type are separated by a period (.). Objects in a library can have the same name if they are different object types, so the object type must be specified to uniquely identify the object.

The object name in each component can be up to 10 characters long, and the object type can be up to 6 characters long.

The directory hierarchy within QSYS.LIB can be either two or three levels deep (two or three components in the path name), depending on the type of the object being accessed. If the object is a database file, the hierarchy can contain three levels (library, file, member); otherwise, there can be only two levels (library, object). The combination of the length of each component name and the number of directory levels determines the maximum length of the path name.

If root (/) and QSYS.LIB are included as the first two levels, the directory hierarchy for QSYS.LIB can be four or five levels deep.

The characters in names are converted to code page 37 when the names are stored. Quoted names are stored using the code page of the job.

Document Library Services (QDLS) File System

The Document Library Services (QDLS) file system supports the folder objects. It provides access to documents and folders.

User Access

To work with the QDLS file system through the integrated file system interface, use the integrated file system commands, user displays, or ISO stream I/O functions and system APIs.

All users working with objects in QDLS must be enrolled in the system distribution directory.

Path Names

QDLS does not distinguish between uppercase and lowercase in the names containing only the alphabetic characters a to z. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

Other characters are case sensitive and are used as is.

Each component of the path name can consist of just a name, such as:

```
/QDLS/FLR1/DOC1
```

or a name plus an extension, such as:

```
/QDLS/FLR1/DOC1.TXT
```

The name in each component can be up to 8 characters long, and the extension can be up to 3 characters long. The maximum length of the path name is 82 characters.

The directory hierarchy below /QDLS/ can be 32 levels deep.

The characters in names are converted to code page 500 when the names are stored. A name may be rejected if it cannot be converted to code page 500.

LAN Server/400 (QLANSrv) File System

The LAN Server/400 (QLANSrv) file system provides access to the same directories and files that are accessed through the LAN Server/400 licensed program. It allows users of the IBM i file server and IBM i applications to use the same data as LAN Server/400 clients.

Files and directories in the QLANSrv file system are stored and managed by a LAN server that is based on the OS/2 LAN server. This LAN server does not support the concept of a file or directory owner or owning group. File ownership cannot be changed using a command or an ISO stream I/O function and system API. Access is controlled through *access control lists*. You can change these lists by using the WRKAUT and CHGAUT commands.

User Access

To work with the QLANSrv file system through the integrated file system interface, use the integrated file system commands, user displays, or ISO stream I/O functions and system APIs.

Path Names

The file system preserves the same uppercase and lowercase form in which object names are entered. No distinction is made between uppercase and lowercase when the system searches for names. A search for object names achieves the same result regardless of whether characters in the names are uppercase or lowercase.

- Path names have the following form:

```
Directory/Directory/ . . . /Object
```

- Each component of the path name can be up to 255 characters long.
- The directory hierarchy within QLANSrv can be 127 levels deep. If all components of a path are included as hierarchy levels, the directory hierarchy can be 132 levels deep.
- Names are stored in the code page that is defined for the File Server.

Optical Support (QOPT) File System

The Optical Support (QOPT) file system can be accessed through the integrated file system interface. This is done using either the IBM i file server or the integrated file system commands, user displays, and ISO stream I/O functions, and system APIs.

Path Names

QOPT converts the lowercase English alphabetic characters a to z to uppercase when used in object names. Therefore, a search for object names that uses only those characters is not case-sensitive.

- The path name must begin with a slash (/) and contain no more than 294 characters. The path is made up of the file system name, the volume name, the directory and subdirectory names, and the file name. For example:

```
/QOPT/VOLUMENAME/DIRECTORYNAME/SUBDIRECTORYNAME/FILENAME
```

- The file system name, QOPT, is required.
- The volume name is required and can be up to 32 characters long.
- One or more directories or subdirectories can be included in the path name, but none are required. The total number of characters in all directory and subdirectory names, including the leading slash, cannot exceed 63 characters. Directory and file names allow any character except 0x00 through 0x3F, 0xFF, 0x80, lowercase-alphabetic characters, and the following characters:
 - Asterisk (*)
 - Hyphen (-)
 - Question mark (?)
 - Quotation mark (")
 - Greater than (>)
 - Less than (<)
- The file name is the last element in the path name. The file name length is limited by the directory name length in the path. The directory name and file name that are combined cannot exceed 256 characters, including the leading slash.
- The characters in names are converted to code page 500 within the QOPT file system. A name may be rejected if it cannot be converted to code page 500. Names are written to the optical media in the code page that is specified when the volume was initialized.

File Server (QFileSvr.400) File System

The File Server (QFileSvr.400) file system can be accessed through the integrated file system (IFS) interface. This is done by using either the IBM i file server or the integrated file system commands, user displays, and ISO stream I/O functions and system APIs.

Note: The characteristics of the QFileSvr.400 file system are determined by the characteristics of the file system that are being accessed on the target system.

Path Names

For a first-level directory, which actually represents the root (/) directory of the target system, the QFileSvr.400 file system preserves the same uppercase and lowercase form in which object names are

entered. However, no distinction is made between uppercase and lowercase when QFileSvr.400 searches for names.

For all other directories, case-sensitivity is dependent on the specific file system being accessed. QFileSvr.400 preserves the same uppercase and lowercase form in which object names are entered when file requests are sent to the IBM i file server.

- Path names have the following form:

```
/QFileSvr.400/RemoteLocationName/Directory/Directory . . . /Object
```

The first-level directory (that is, RemoteLocationName in the example shown above) represents both of the following:

- The name of the target system that will be used to establish a communications connection. The target system name can be either of the following:
 - A TCP/IP host name (for example, beowulf.newyork.corp.com)
 - An SNA LU 6.2 name (for example, appn.newyork)
- The root (/) directory of the target system

Therefore, when a first-level directory is created using an integrated file system interface, any specified attributes are ignored.

Note: First-level directories are not persistent across initial program loads (IPLs). That is, the first-level directories must be created again after each IPL.

- Each component of the path name can be up to 255 characters long. The absolute path name can be up to 16 megabytes long.

Note: The file system in which the object resides may restrict the component length and path name length to less than the maximum allowed by QFileSvr.400.

- There is no limit to the depth of the directory hierarchy, other than program and system limits, and any limits that are imposed by the file system being accessed.
- The characters in names are converted to UCS2 Level 1 form when the names are stored.

Enabling Integrated File System Stream I/O

You can enable ILE C/C++ stream I/O for files up to two gigabytes in size by specifying the *IFSIO option on the system interface keyword (SYSIFCOPT) on the Create Module or Create Bound Program command prompt. For example:

```
CRTCMOD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFSIO)
CRTBNDC PGM(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFSIO)
```

Using Stream I/O with Large Files

The 64-bit version of the integrated file system interface lets you use ILE C/C++ Stream I/O with files greater than two gigabytes in size. Use any of the methods listed below to enable this interface.

- Specify the *IFS64IO option with the SYSIFCOPT keyword on the Create Module or Create Bound Program command prompt. When this keyword is specified, the compiler defines the __IFS64_IO__ macro, which in turn causes the _LARGE_FILES and _LARGE_FILE_API macros to be defined in the IBM-supplied header files. For example:

```
CRTCPMOD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFS64IO)
```

- Define the _LARGE_FILES macro in the program source. Alternately, specify DEFINE('_LARGE_FILES') on a Create Module or Create Bound Program command line. Integrated file system APIs and relevant data types are automatically mapped or redefined to their 64-bit integrated file system counterparts. For example:

```
CRCTPPMOD MODULE(QTEMP/IFSIO) SRCFILE(QCPPL/QACSRC)
          SYSIFCOPT(*IFSIO) DEFINE('_LARGE_FILES')
```

- Define the `_LARGE_FILE_API` macro in the program source. Alternately, specify `DEFINE('_LARGE_FILE_API')` on a Create Module or Create Bound Program command line. This makes 64-bit integrated file system APIs and corresponding data types visible, but applications must explicitly specify which integrated file system APIs (regular or 64-bit) to use. For example:

```
CRCTMOD MODULE(QTEMP/IFSIO) SRCFILE(QCPPL/QACSRC)
          SYSIFCOPT(*IFSIO) DEFINE('_LARGE_FILE_API')
```

Note: The `__IFS64_IO__`, `_LARGE_FILES`, and `_LARGE_FILE_API` macros are not mutually exclusive. For example, you might specify `SYSIFCOPT(*IFS64IO)` on the command line, and define either or both of the `_LARGE_FILES` and `_LARGE_FILE_API` macros in your program source.

Stream Files

The ILE C/C++ compiler allows your program to process stream files as true text or binary stream files (using the integrated file system enabled stream I/O) or as simulated text and binary stream files (using the default stream I/O).

When writing an application that uses stream files, for better performance, it is recommended that the integrated file system be used instead of the default C stream I/O which is mapped on top of the record I/O.

Stream Files Versus Database Files

To better understand stream files, it is useful to compare them with IBM i database files.

On the integrated file system, a stream is simply a continuous string of characters. A database file is record arranged; It has predefined subdivisions consisting of one or more fields that have specific characteristics, such as length and data type.

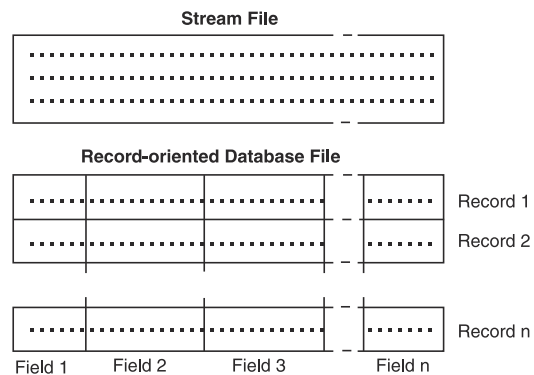


Figure 98. Comparison of a Stream File and a Record-Oriented File

Default C/C++ stream I/O on the IBM i is simulated on top of an IBM i database file. [Figure 99 on page 165](#) illustrates how an IBM i record is mapped to a C/C++ stream. This is simulated stream file processing with IBM i records.

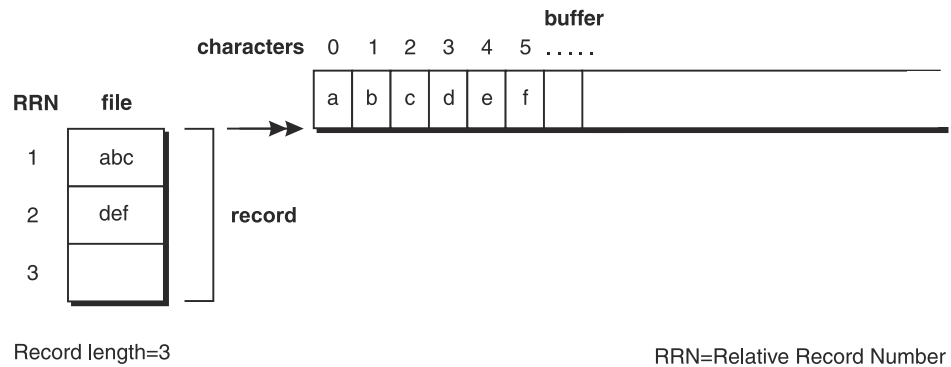


Figure 99. IBM i Records Mapping to a C/C++ Stream File

The differences in structure of stream files and record-oriented files affect how an application is written to interact with them and which type of file is best suited to an application.

- A *record-arranged file* is well suited for storing customer information, such as name, address, and account balance. These fields can be individually accessed and manipulated using the extensive programming functions of the IBM i.
- A *stream file* is better suited for storing information such as a customer's picture, which is composed of a continuous string of bits representing variations in color. Stream files are particularly well suited for storing strings of data such as the text of a document, images, audio, and video.

Text Streams

Text streams contain printable characters and control characters that are organized into lines. Each line consists of zero or more characters and ends with a new-line character (`\n`). A new-line character is not automatically appended to the end of file.

The ILE C/C++ runtime environment may add, alter, or ignore some special characters during input or output so as to conform to the conventions for representing text in the IBM i environment. Thus, there may not be a one-to-one correspondence between characters written to a file and characters read back from the same file.

Data read from an integrated file system text stream is equal to the data which was written if the data consists only of printable characters and the horizontal tab, new-line, vertical tab, and form-feed control characters.

For most integrated file system stream files, a line consists of zero or more characters, and ends with the following character combination: a carriage return plus a new-line character. However, the integrated file system can have logical links to files on different systems that may use a single line-feed as a line terminator. A good example of this are the files on most UNIX systems.

On input, the default in text mode is to strip all carriage-returns from new-line carriage-return character combination line delimiters. On output, each line-feed character is translated to a carriage-return character that is followed by a line-feed character. The line terminator character sequence can be changed with the `CRLN` option on `fopen()`.

Note: The `*IFSIO` option also changes the value for the `'\n'` escape character value to the `0x25` line feed character. If `*NOIFSIO` is specified, the `'\n'` escape character has a value of `0x15`.

When a file is opened in text mode, there may be code-page conversions on data that is processed to and from that file. When the data is read from the file, it is converted from the code page of the file to the code page of the application, job, or system receiving the data.

When data is written to an IBM i file, it is converted from the code page of the application, job, or system to the code page of the file. For true stream files, any line-formatting characters (such as carriage return, tab, and end-of-file) are converted from one code page to another.

When reading from QSYS.LIB files end-of-line characters (carriage return and line feed) are appended to the end of the data that is returned in the buffer.

The code-page conversion that is done when a text file is processed can be changed by specifying the code-page or CCSID option on `fopen()`. The default is to convert all data read from a file to the job's CCSID or code page.

Binary Streams

A *binary stream* is a sequence of characters that has a one-to-one correspondence with the characters stored in the associated IBM i file. The data is not altered on input or output, so the data that is read from a binary stream is equal to the data that was written. New-line characters have no special significance in a binary stream. The application is responsible for knowing how to handle the data. The `fgets()` function handles new-line characters. Binary files are always created with the CCSID of the job.

Opening Text Stream and Binary Stream Files

Each text stream file and each binary stream file is represented by a file control structure of type `file`. This structure is initialized depending on the mode in which the file was opened. Unpredictable results might occur if you attempt to change the file control structure.

The format of `fopen()` is:

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

The *mode* variable is a character string that consists of an open *mode* which may be followed by keyword parameters. The open mode and keyword parameters must be separated by a comma or one or more blank characters.

To open a text stream file, use `fopen()` with one of the following modes:

- `r` or `r+`
- `w` or `w+`
- `a` or `a+`

To open a binary stream file, use `fopen()` with one of the following modes:

- `rb`,
- `rb+` , or `r+b`
- `wb`,
- `wb+` or `w+b`
- `ab`,
- `awb+` or `a+b`

 C++

To open a binary stream file, use the `open()` member function with `ios::binary`, or any of the following modes:

- `ios::app`
- `ios::ate`
- `ios::in`
- `ios::out`
- `ios::trunc`

Storing Data as a Text Stream or as a Binary Stream

If two streams are opened, one as a binary stream and the other as text stream, and the same information is written to both, the contents of the stream may differ. The following illustrates two streams of different types. The hexadecimal values of the resulting files (which show how the data is actually stored) are not the same.


```

/* Use CRTBNDC SYSIFCOPT(*IFSIO)                                */
#include <stdio.h>
int main(void)
{
    FILE *fp1, *fp2;
    char lineBin[15], lineTxt[15];
    int x;
    fp1 = fopen("script.bin","wb");
    fprintf(fp1,"hello world\n");
    fp2 = fopen("script.txt","w");
    fprintf(fp2,"hello world\n");
    fclose(fp1);
    fclose(fp2);
    fp1 = fopen("script.bin","rb");
    /* opening the text file as binary to suppress
    the conversion of internal data */
    fp2 = fopen("script.txt","rb");
    fgets(lineBin, 15, fp1);
    fgets(lineTxt, 15, fp2);
    printf("Hex value of binary file = ");
    for (x=0; lineBin[x]; x++)
        printf("%.2x", (int)(lineBin[x]));
    printf("\nHex value of text file = ");
    for (x=0; lineTxt[x]; x++)
        printf("%.2x", (int)(lineTxt[x]));
    printf("\n");
    fclose(fp1);
    fclose(fp2);

    /* The expected output is:                                */
    /*                                                            */
    /* Hex value of binary file = 888593939640a69699938425    */
    /* Hex value of text file   = 888593939640a6969993840d25    */
}

```

Figure 100. Comparison of Text Stream and Binary Stream Contents

As the hexadecimal values of the file contents shows in the binary stream (script.bin), the new-line character is converted to a line-feed hexadecimal value (0x25). While in the text stream (script.txt), the new-line is converted to a carriage-return line-feed hexadecimal value (0x0d25).

Using the Integrated File System (IFS)

ILE C/C++ primarily supports the IBM i root file system. The root file system is one of the many file systems accessible through the integrated file system interface. It uses notation similar to that used to access files and directories on UNIX systems, allowing you to access information across multiple platforms in a uniform way.

Take care when transferring files to and from various platforms. Use of a download and upload utility like FTP allows you to specify the correct mapping of characters so your streamed source remains valid on the IBM i platform, even if it has been stored temporarily on other platforms. See [“Pitfalls to Avoid” on page 175](#) for more tips.

Copying Source Files into the IFS

You can copy your main source physical file to an integrated file system (IFS) file. Assuming that you used a standard name for your source physical file, use the following command:

```
CPYTOSTMF FROMMBR('/QSYS.LIB/MYLIB.LIB/QCSRC.FILE/QCSRC.MBR') TOSTMF('/home/qcsrc.c')
```

Editing Stream Files

You can edit stream files directly with the Edit File (EDTF) command. There are also three ways that you can edit files to be used with stream files:

- Use Client Access to map the integrated file system directory as a PC network drive and then use a PC-based editor to edit files in that path as if they were local PC files.

- Edit with SEU and then use the Copy to Stream File (CPYTOSTMF) command to move that file from the traditional QSYS file system to a root file system path.
- Place the source in a Source Physical File (SRCPF) with integrated file system links. (The actual source resides in a QSYS member, but there is a root file system link that points to the member.) Use the Add Link (ADDLNK) command to create the link, and thereafter edit the member with SEU, but use the root file system pathname link when you compile.

The SRCSTMF Parameter

The SRCSTMF parameter identifies a source stream file as a path name. Specify the path name of the stream file that contains the ILE C source code that you want to compile. The path name can be either absolutely qualified or relatively qualified.

For file systems that are case sensitive, the path name may be case sensitive.

An absolutely qualified name starts with / or \. A / or \ character at the beginning of a path name means that the path begins at the topmost directory, the root (/) directory. For example:

```
/Dir1/Dir2/Dir3/UsrFile
```

If the path name does not begin with a / or \ character, the path is assumed to begin at your current directory. For example:

```
MyDir/MyFile
```

is equivalent to

```
/CurrentDir/MyDir/MyFile
```

where MyDir is a subdirectory of your current directory.

There is no support for the tilde (~) character or wildcards (* or ?).

SRCSTMF is mutually exclusive with SRCMBR and SRCFILE. Also, if you specify SRCSTMF, then the compiler ignores TEXT(*SRCMBRTXT). Other values for TEXT are valid.

Header File Search

The compiler uses different search techniques when entering your source file using the source stream file parameters. The compiler no longer uses the library list search method.

Include File Links

ILE C/C++ headers, along with system headers, are located in QSYSINC/H. The links are in the directory /QIBM/include.

For example, the links are as follows for `assert.h`:

- Display Symbolic Link Object link: `/QIBM/include/assert.h`
- Content of Link: `/qsys.lib/qsysinc.lib/h.file/assert.mbr`

ILE C++ standard library headers are located in QSYSINC/STD. The links are in the directory /QIBM/include/std.

For example, the links are as follows for `<algorithm>`:

- Display Symbolic Link Object link: `/QIBM/include/std/algorithm`
- Content of Link: `/QSYS.LIB/QSYSINC.LIB/STD.FILE/ALGORITHM.MBR`

Note:

Because files (and members) within the QSYS.LIB file system are limited to 10 characters in length, the C/C++ header names in QSYSINC have to be truncated if their standard names are longer than 10 characters. And intentionally, the compiler truncates include file names to 10 characters when the

QSYS.LIB file system is used (that is, when the SRCFILE/SRCMBR options are used). For example, `#include <type_traits>` is truncated to `#include <type_trait>` (from 11 characters to 10 characters). This allows source code to compile cleanly without change. And it works well as long as the include files are unique within the first 10 characters.

Beginning with C++ TR1 (Draft Technical Report on C++ Library Extensions, ISO/IEC DTR 19768), some new C++ standard library header file names are not unique within their first 10 characters. Specifically, 2 new header files named `<unordered_set>` and `<unordered_map>` were introduced and the truncation of both files maps to "unordered_".

To avoid the conflicts within QSYSINC/STD, a file member UNORDERED_ was created for allowing include with the standard name `<unordered_map>` or `<unordered_set>`. The `<UNORDERED_>` acts as a wrapper header file and further includes the corresponding STD header files in short names for the QSYS.LIB file system, which are `<unord_map>` and `<unord_set>`. That said, `<unord_map>` and `<unord_set>` are the real headers in QSYSINC/STD. If the standard name `<unordered_map>` or `<unordered_set>` is included, both `<unord_map>` and `<unord_set>` are included.

Note that the header files and links in the directory `/QIBM/include` do not have file name length limitation. Therefore, all standard headers are shipped with their standard names within `/QIBM/include`.

Include Directive Syntax

The `#include` directive syntax depends on the file system specified for the root source file: integrated file system (IFS) or (DM) file system.

Integrated File System (IFS) compilations

IFS is a hierarchical file system similar to that found on AIX®.

When an IFS file specification is used for the root source file (that is, when the SRCSTMF option is used), all `#include` directives within that compilation are similarly resolved to the IFS file system. The syntactical variations are:

#include specification	enclosed in <>	enclosed in " "
filename (e.g., <code><stdio></code>)	resolves to <code>[syssearchpath]/filename</code>	resolves to <code>[usrsearchpath]/filename</code>
dir/filename (e.g., <code><sys/limits.h></code>)	resolves to <code>[syssearchpath]/dir/filename</code>	resolves to <code>[usrsearchpath]/dir/filename</code>
<code>/dir/filename</code> (e.g., <code>"/home/header.h"</code>)	resolves to <code>/dir/filename</code>	resolves to <code>/dir/filename</code>

File System (DM) compilations

DM is the traditional IBM i monolithic (fixed-depth) file system. It is composed of a number of libraries, which contain objects. There are a fixed set of object types - source files are found within *FILE object types, in sub-objects called members. All native IBM i processes have an ordered library list (*LIBL) and, in general, IBM i objects are resolved by searching through this library list. The library list has three components, ordered as follows:

- The *System Library List* (*SYSLIBL): A set of libraries which comprises the operating system.
- The *Product Library List* (*PRDLIBL): Officially licensed programs typically add themselves to the product library list when run. For example, the C and C++ compilers add their product library QCPPL to the library list when run.
- The *User Library List* (*URSLIBL): Libraries that you can configure or order.

When a DM file specification is used for the root source file (that is, when the SRCFILE/SRCMBR options are used), all #include directives within that compilation are similarly resolved to the DM filesystem. The syntactical variations are:

#include specification	library	file	member
mbr	default search ¹	default file ²	mbr
mbr.file	default search ¹	file	mbr
file/mbr	default search ¹	file	mbr
file(mbr)	default search ¹	file	mbr
file/mbr.ext ³	default search ¹	file	mbr or mbr.ext ³
lib/file/mbr	lib	file	mbr
lib/file(mbr)	lib	file	mbr



Note:

1. For default library search paths:

Note: When the *SYSINCPATH option is specified, the compiler treats user includes (" ") the same as system includes (< >).

- When the library is not specified, and:
 - the #include specification is enclosed in < >: Search the *LIBL.
 - the #include specification is enclosed in " ": Check the library containing the root source file; if not found there, then search the *USRLIBL; if still not found, search the *LIBL.
- When the library is specified, and:
 - the #include specification is enclosed in < >: Search the lib/file/mbr only.
 - the #include specification is enclosed in " ": Search for the member in the library/file specified.

2. For the Default file:

- When includes have the form #include <mbr>:
 -  the default file is H.
 -  the default file is STD.
- When includes have the form #include "mbr": the default file is the root source file.

3. For mbr.ext:

- When the #include specification is enclosed in < >, and:
 - the member name has the h extension: <file/mbr.h> format resolves to member *mbr* in file *file*. This rule is for POSIX support (for example, to be able to include specifications like <sys/limits.h>). The only member names which activate POSIX support are extensions of h or H.
 - Otherwise, <file/mbr.ext> resolves to file *file*, and member *mbr.ext*
- When the #include specification is enclosed in " ": "*file/mbr.ext*" resolves to file *file*, and member *mbr.ext*

Include Search Path Rules

This section lists rules governing search paths for the following:

- INCDIR (Include Directory) command parameter
- INCLUDE environment variable

- *STDINC/*NOSTDINC command options
- *SYSINCPATH/*NOSYSINCPATH command options

INCDIR (Include Directory) Command Parameter

The Include Directory parameter (INCDIR) works with the Create Module and Create Bound Program compiler commands, allowing you to redefine the path used to locate include header files (with the #include directive) when compiling a source stream file only. The parameter is ignored if the source file's location is not defined as an IFS path via the Source Stream File (SRCSTMF) parameter, or if the full (absolute) path name is specified on the #include directive.

The parameter accepts a list of IFS directories. These directories are inserted into the include search path in the order they are entered.

The include files search path adheres to the following directory search order to locate the file:

#include type	Directory Search Order
#include <file_name>	<ol style="list-style-type: none"> 1. If you specify a directory in the INCDIR parameter, the compiler searches for <i>file_name</i> in that directory first. 2. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. 3. Searches the directory /QIBM/include.
#include "file_name"	<ol style="list-style-type: none"> 1. Searches the directory where your current source file resides. The current source file is the file that contains the #include "file_name" directive. 2. If you specify a directory in the INCDIR parameter, the compiler searches for file_name in that directory. 3. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. 4. Searches the directory /QIBM/include.

For example, if you enter the following value for the INCDIR parameter:

```
Include directory . . . . . INCDIR      '/tmp/dir1'
                  + for more values    './dir2'
```

and with your source stream file you include the following header files:

```
#include "foo.h"
#include <stdio.h>
```

The compiler first searches for a file "foo.h" in the directory where the root source file resides. If the file is found, it is included and the search ends. Otherwise, the compiler searches the directories entered INCDIR, starting with "/tmp/dir1". If the file is found, this file is included. If the directory does not exist, or if the file does not exist within that directory, the compiler continues to search in the subdirectory "dir2" within the current working directory (symbolized by "."). Again, if the file is found, this file is included, otherwise, because the directories in INCDIR path have now been exhausted, the default user include path (/QIBM/include) is used to find the header.

As for <stdio.h>, the same logic is followed in the same order, except the initial search in the root source directory is bypassed.

INCLUDE Environment Variable

The INCLUDE environment variable value:

- Contains a path of directories delimited by colons (:)
- Does not override the order
- Has higher priority in the search order than the default include path
- Has a lower priority in the search order than INCDIR and the root source's directory (for a user-defined include search)

If the include search contains a defined INCLUDE environment variable for both C and C++ compilers, the resulting include search order including is as shown in the following table:

#include type	Directory Search Order
#include <file_name>	<ol style="list-style-type: none">1. If you specify a directory in the INCDIR parameter, the compiler searches for <i>file_name</i> in that directory first.2. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line.3. If the INCLUDE environment variable is defined, the compiler searches the directories in the order they appear in the INCLUDE path.4. Searches the directory /QIBM/include.
#include "file_name"	<ol style="list-style-type: none">1. Searches the directory where your current source file resides. The current source file is the one that contains the directive #include "file_name".2. If you specify a directory in the INCDIR parameter, the compiler searches for <i>file_name</i> in that directory.3. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line.4. If the INCLUDE environment variable is defined, the compiler searches the directories in the order they appear in the INCLUDE environment variable.5. Searches the directory /QIBM/include.

Note: This feature is only available for source stream file compiles.

*STDINC/*NOSTDINC Command Options

The *STDINC/*NOSTDINC command options have been added to the OPTION parameter of the CRTCMOD/CRTCPPMOD and CRTBNDC/CRTBNDCPP commands.

The *STDINC and *NOSTDINC command options work on the CRTCMOD/CRTCPPMOD and CRTBNDC/CRTBNDCPP commands.

The *NOSTDINC option allows you to remove the default include path (/QIBM/include for IFS source stream files; QSYSINC for source file members) from the search order, while the *STDINC option retains the default include path at the end of the order. *STDINC is the default.

The *STDINC option works as did the former SYSINC parameter for source file members. The options relate to the old parameter values as follows:

<i>Table 17. Parameter Values</i>	
SYSINC values	Equivalent New Command Option
*YES	*STDINC
*NO	*NOSTDINC

***INCDIRFIRST/*NOINCDIRFIRST Command Options:**

The *INCDIRFIRST option allows you to process the directories listed via the INCDIR parameter first in the search order (that is, before the root source file directory) in a user include search, while the *NOINCDIRFIRST option retains INCDIR directories to their default position in the user include search order as described above.

Note: These options are valid only for source stream file compiles.

If *INCDIRFIRST is selected, the following changes occur to the user include search order:

<i>Table 18. INCDIRFIRST Command Options</i>	
#include type	Directory Search Order
#include "file_name"	<ol style="list-style-type: none"> 1. If you specify a directory in the INCDIR parameter, the compiler searches for file_name in that directory. 2. If more than one directory is specified, the compiler searches the directories in the order that they appear on the command line. 3. Searches the directory where your current root source file resides. 4. If the INCLUDE environment variable is defined, the compiler searches the directories in the order they appear in the INCLUDE path. 5. If the *NOSTDINC compiler option is not chosen, search the default include directory /QIBM/include.

***SYSINCPATH/*NOSYSINCPATH Command Options**

The *SYSINCPATH/*NOSYSINCPATH command options work on the Create Module and Create Bound Program commands.

The *SYSINCPATH option changes the search path of user includes to the system include search path. It is equivalent to changing the double-quotes in the user #include directive (#include "file_name") to angle brackets (#include <file_name>). *NOSYSINCPATH is the default value.

Considerations for Specifying Source Stream Files

When you specify the SRCSTMF parameter during program or module creation, the *MODULE object contains no source file attribute information.

If the source is specified via the SRCFILE/SRCMBR parameters, the INCDIR parameter and the INCLUDE environment variable are ignored. When the source resides in the file system, the library list is used to search for include files. The library list (*LIBL) has no concept of the directory file structure.

If the source file is not specified via the SRCSTMF parameter, the job's *LIBL is used to find the include files.

Restrictions on the Absolute Include Path Name

If you specify an absolute (full) path name on the #include directive, the INCDIR parameter and the INCLUDE environment variable have no effect.

Example:

Assume the following:

- There is a statement like `#include "myinc.h"` in a C/C++ source file
- You are compiling a source member from the QSYS file system through the SRCSTMF parameter in the following command:

```
CRTCMOD MODULE(MYSOURCE) SRCSTMF('/qsys.lib/goodness.lib/qcppsrc.file/mysource.mbr')
```

ILE C/C++ tries to find something called `/qsys.lib/goodness.lib/qcppsrc.file/myinc.h`, which is an invalid integrated file system filename because `.h` is not a valid object type in the QSYS file system.

If you want to use a header file that is in the QSYS file system, you must do either of the following:

- Specify the path in the source code, as shown below:

```
#include "/qsys.lib/goodness.lib/h.file/myinc.mbr"
```

- Set the search path appropriately, as shown in [“Examples of Using Integrated File System Source” on page 175](#). Then you can leave the path out of the include statement, as shown below:

```
#include "myinc.mbr"
```

Recommendation for Source and Header Files

If you are porting from other platforms which have hierarchical file systems (such as the Microsoft Windows, UNIX, or OS/2 operating systems), consider that the integrated file system (IFS) is more compatible with those file systems. To avoid changing your current C/C++ source code, put source and header files into IFS.

Preprocessor Output

If you specify SRCSTMF with `OPTION(*PPONLY)`, then the preprocessor writes a stream file to the current directory with the new extension `.i`. For example, if you specify `SRCSTMF('/home/MOE/mainprogram.c')` with `OPTION(*PPONLY)`, then the preprocessor writes output to the current directory as a stream file called `mainprogram.i`. For this to happen, you need `*RW` authority to the current directory.

Listing Output

The compiler can send the listing output to a user-specified IFS file, as well as to a spool file. The prolog identifies the source file from a path name:

```
Module . . . . . : TEST
Library . . . . . : MOE
Source stream file . . . . . : /home/MOE/src/mainprogram.c
```

Note: The source stream file is not included in the prolog when `SRCFILE()` and `SRCSTMF()` are specified.

The listing also identifies the include files from their path names:

```
**** FILE TABLE SECTION ****
0 = hello.cpp
1 = /QIBM/include/iostream.h
2 = /QIBM/include/string.h
```

The listing includes:

- The files specified in any `#include` directive
- The file specified or implied in the `SRCSTMF()` or `SRCFILE()/SRCMBR()` options

Note: This happens for either a database file or a stream file source.

The format of the OUTPUT option is: OUTPUT(*NONE | *PRINT | filename), where *PRINT causes the compiler to send it to a spool file, and filename causes the compiler to send it to a user-specified IFS file.

Code Pages and CCSIDs

For source physical files, the compiler respects the CCSID of ILE C source. A similar scheme exists for stream file compilation. Stream files have a code page attribute.

The compiler converts source files, translating code pages to the root source.

The source stream file may have been entered through a mounted file on an ASCII system. In such a case, the compiler translates from the ASCII codepage that is associated with the stream file (for example, 437) to EBCDIC (for example, 37).

Support for Unicode wide-character literals can be enabled when building your program by specifying LOCALETYPE(*LOCALEUCS2) on the compile command. See [“International Locale Support”](#) on page 413 for more information..

You can configure most file transfer utilities to automatically do the conversion to enable ASCII-based file systems to work for producing stream file source.

Pitfalls to Avoid

Any source file created on the workstation with an ASCII editor that deposits an EOF marker at the end of a text file will generate an invalid character warning message when it is compiled with the ILE C/C++ compiler. This includes your main source file. The problem arises when the source file is copied to, or saved in, the root(/) file system on the IBM i. This is because of the translation between ASCII and EBCDIC codepoints.

If you receive an invalid character message referring to the last character of a file, it is likely that you have an EOF marker in the file. One way to avoid this problem is to use an editor which does not add the EOF marker.

Alternatively you can use a File Transfer Protocol (FTP) utility. FTP will result in a root(/) file system file with either codepage 819 or 37. Any of these FTP commands issued to the target IBM i prior to the put command will result in a file of codepage 819:

- `ascii`
- `quote type a`

If you issue the following command to the target IBM i prior to the put command, put results in a file with codepage 37 (EBCDIC): `quote type e`. When the file is transferred using FTP to the Root file system, the file is created with either codepage 819 or codepage 37 (depending on the previous commands as outlined above) whether the file exists prior to the transfer or not.

Files transferred to an integrated file system with codepage 37, fail to compile.

Examples of Using Integrated File System Source

The most basic entry of an integrated file system name does not specify any path information.

Create C++ Module (CRTCPPMOD)

Type choices, press Enter.

```
Module . . . . . > TEST           Name
Library . . . . . *CURLIB        Name, *CURLIB
Source file . . . . . QCPPSRC     Name
Library . . . . . *CURLIB        Name, *CURLIB
Source member . . . . . *MODULE   Name, *MODULE
Source stream file . . . . . > test.cpp
Text 'description' . . . . . *BLANK
```

```
F3=Exit   F4=Prompt   F5=Refresh   F10=Additional parameters   F12=Cancel   Bottom
F13=How to use this display   F24=More keys
```

Without a pathname, the system assumes that your source is located in the current directory. The default current directory is the base (/) directory of the root file system, but your individual user profile may change this default to a different directory. You can change the current directory with the Change Current Directory (CHGCURDIR) command.

Note: The current directory and the current library are separate and distinct entities. Although you can set the current library and the current directory to be the same name, a change in one will not affect the other.

The header files specified in any `#include` statements in your source will be searched for in the source directory first and then the specified INCDIR directory. For example, if you compile the following source in file `/goodness/mysource.cpp`:

```
#include "special/mystuff.h"

class test : public base
{
  .
  .
  .
}
```

with the INCDIR value set to `/mydir`, your included header file is first searched for as `/goodness/special/mystuff.h` and then `/mydir/special/mystuff.h`.

Using Stream I/O

The following sections describe stream I/O requirements for using:

- Large files
- Open mode
- Line-end characters

Large Files

Within the C or C++ runtime environment, stream I/O for files up to two GB in size is enabled by specifying the `*IFSIO` option on the system interface keyword (SYSIFCOPT) on the Create Module or Create Bound Program command prompt.

When using the SYSIFCOPT keyword with either command, follow this format:

```
CRTCPPMOD MODULE(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFSIO)
CRTBNDCPP PGM(QTEMP/IFSIO) SRCFILE(QCPPLE/QACSRC) SYSIFCOPT(*IFSIO)
```

When the `*IFSIO` option is specified, the compiler defines the `__IFS_IO__` macro. When `__IFS_IO__` is defined, prototypes associated with stream I/O in `<stdio.h>` are no longer defined. The header file

`<ifs.h>` is included by `<stdio.h>`, which declares all structure and prototypes associated with the integrated file system enabled C stream I/O.

C The 64-bit version of the integrated file system interface lets you use ILE C stream I/O with files up to and greater than two gigabytes in size. (C++ stream I/O for files greater than two gigabytes is not supported.) To enable the 64-bit interface, specify the `*IFS64IO` option with the `SYSIFCOPT` keyword on the `CRTCPMOD` or `CRTBNDCPP` command prompt. When this option is specified, the compiler defines the `__IFS64_IO__` macro which, for example, remaps the `open()` function to an `open64()` function to allow 64-bit indexing.

Open Mode

C++ The `fstream()`, `ifstream()`, and `ofstream()` classes have a new open mode `ios::text`, which opens the file in text mode.

By default, I/O streams are opened in binary mode.

Line-End Characters

- If the input or output is unformatted (for example, via the `read()` or `write()` method), newline (`\n`) characters are *not* expanded to `\r\n` on output and `\r` characters are *not* stripped out on input.
- **C++** If the input or output is formatted (via the `>>` or `<<` operator), newline (`\n`) characters are *not* expanded to `\r\n` on output but any `\r` characters *are* stripped out on input

If you want to add carriage return (`\r`) characters, use the `fopen()` function with `cr1n=Y` (the default).

Working with File Systems and Devices

This topic describes how to:

- Retrieve external file descriptions
- Work in disconnected mode
- Include externally described physical and logical database files
- Use physical and logical database files and distributed data
- Use commitment control
- Use display files, printer files, ICF files, tape files, diskette files, and save files
- Use the device attributes feedback area

Using Externally Described Files in a Program

This topic describes how to:


- Create externally described files
- Use level checking to verify that the descriptions with which the program was compiled are still functional
- Avoid field alignment problems in C/C++ structures
- Including external field definitions in a program
- Define and use indicators
- Include physical and logical database files in a program
- Include device files in a program
- Include multiple record formats in a program
- Include packed decimal data in a program

Creating Externally Described Database Files

Externally described files are files that have their field descriptions stored as part of the file. The description includes information about the type of file (such as data or device), record formats, and a description of each field and its attributes.

You can create an externally described database file using any of the following:

- SQL/400 database
- Interactive Data Definition Utility (IDDU) using DDS for externally described files
- Data Description Specifications (DDS)

 The ILE C preprocessor automatically creates C structure type definitions from external file descriptions when you use the `#pragma mapinc` directive with the `#include` directive.

Note: You cannot use the `#pragma mapinc` directive if you are compiling IFS files. For more information about including IFS files in a program, see [“The GENCSRC Utility and the #pragma mapinc Directive” on page 423](#).

The `#pragma mapinc` directive identifies only those file formats and fields to the compiler; it does not include the file description in the ILE C program. To include a file description, the `#include` directive must be coded in the ILE C program.

You refer to the include-name parameter of the `#pragma mapinc` directive on the `#include`. The `#include` directive must be coded after the `#pragma mapinc` directive in your source program.

For example, to include a type definition of the input fields for the record format FMT from the file EXAMPLE/TEST, the following statements must appear in your program in the order shown below:

```
#pragma mapinc("tempname", "EXAMPLE/TEST(FMT)", "input", "d", ",")
#include "tempname"
```

Creating Type Definitions

To create the type definition structure to be included in your ILE C program, use the *options* parameter. A header description is also created. This header description contains information about the external file.

C++ C++ users must use the GENCSRC utility for creating type definitions.

C C users can use either the GENCSRC utility or the `#pragma mapinc` directive for creating type definitions.

Note: For more information on the differences between the GENCSRC utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

To see the type definitions in your compiler listing, specify `OPTION(*SHOWUSR)`.

Note: `OPTION(*SHOWINC)` expands any include file created by GENCSRC or `#pragma mapinc`, but it also expands the system includes.

Creating Header Descriptions

The header description for each format contains the following information:

- File and library name of the external file
- File type (physical, logical, device)
- Date the file was created
- Record format name
- Record format level ID (level checking information)

C For example, the following directives are used to create the header shown below:

```
#pragma mapinc("payroll", "example/test(fmt1)", "input", "")
#include "payroll"

/* ----- */
/* PHYSICAL FILE: EXAMPLE/TEST */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
```

Figure 101. Header Description

The following is an example of a type definition of type structure:

```
typedef struct {
    .
    .
    .
} LIBRARY_FILE_FORMAT_tag_t;
```

C Parameters of the `#pragma mapinc` directive are used to create the name of the created type. LIBRARY, FILE, and FORMAT are the library-name, file-name, and format-name specified on the `#pragma mapinc` directive. These names are folded to uppercase unless quoted names are used. The library and file names can be replaced with your own prefix-name as specified on the `#pragma mapinc` directive.

C Any characters that are not recognized as valid by the C language that appear in library and file names are translated to the underscore (`_`) character.

Note: Do not use the special characters #, @, or \$ in library and file names. If these characters are used in library and file names, they are also translated to the underscore (_) character.

The tag on the structure name indicates the type of fields that are included in the structure definition. The possible values for tag are:

Table 19. Field Type and Tag Values	
Field Type	Tag
input	i
output	o
both	both
key	key
indicators	indic
nullflds	nmap/nkmap

Unlike the naming convention used for other listed field types, if field type `lvlchk` is specified, the name of the array of structure type created is `_LVLCHK_T`.

C To include external file descriptions for more than one format, specify more than one format name (`format1 format2`) or `(*ALL)` on the `#pragma mapinc` directive. A header description and type definitions are created for each format.

When the `lname` option is specified and the filename in the `#pragma mapinc` directive is greater than 10 characters in length, a system-generated 10-character name will be used in the type definitions generated by the compiler.

Specifying the Record Format Name

C A *record format* describes all the fields and the arrangement of these fields within a record. You can include a record format from an externally described file in your ILE program by providing its name on the `#pragma mapinc` directive. You can provide more than one format name, or you can specify the special value `*ALL` to include all record formats from the file.

If the file you are working with contains more than one record format, set the format for subsequent I/O operations with the `_Rformat()` function.

Record format functions are useful when working with display, ICF, and printer files. Logical files can also contain more than one record format.

The record format name for a device file defaults to blank unless you explicitly set it to a name with `_Rformat()`. You can reset the format name to blank by passing a blank name to `_Rformat()`.

C If the record format does not contain fields that match the option specified (input, output, both, key, indicators or nullflds) on the `#pragma mapinc` directive, the following comment appears after the header description:

```
/* FORMAT HAS NO FIELDS OF REQUIRED TYPE */
```

Note: Do not use #, @, or \$ in record format names. These characters are not allowed in ILE identifiers and cannot appear in a type definition name. If you have record format names that contain #, @ or \$, these characters are translated to the lowercase characters p, a, and d, respectively.

Specifying Record Field Names

When you specify record field names, consider the following:

- All DDS keywords are supported by the ILE library and compiler. The actual comment that is associated with the TEXT keyword is translated to uppercase in the type definition that is generated. The ALIAS keyword is supported and brings the alias field name into the type definition that is generated.
- Some of the special characters that are supported in DDS variable names are not supported by the ILE compiler and library. If you use the special characters (@, #, or \$) in a field name, those characters are changed to lowercase a, p and d in the type definition that is generated.

Note: If the format names contain C characters that are not valid, they are translated to the underscore (_) character.

Including Database Files in the Type Definition

Input and output buffers for database files have the same format. When you specify input, the fields that are defined as either INPUT or BOTH in the externally described database file are included in the type definition. When you specify both, the fields that are defined as either INPUT, OUTPUT, or BOTH are included in the type definition.

If all the fields are defined as BOTH or INPUT, only one type definition structure is generated in the type definition.

Defining the Structure Type (KEY Field)

To include a separate structure type definition for the KEY fields in a format, specify the KEY option on the `#pragma mapinc` directive. Comments are listed beside the fields in the structure definition to indicate how the key fields are defined in the externally described file.

C++ C++ users must use the GENCSRC utility for structure type definition.

C C users can use either the GENCSRC utility or the `#pragma mapinc` directive for structure type definition.

Note: For more information on the differences between the GENCSRC utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

Example:

The following ILE C program contains the `#pragma mapinc` directive to include the externally described database file CUSMSTL:

```
#pragma mapinc("custmf","example/cusmstl(cusrec)","both key","d")
#include "custmf"
```

The following example contains the DDS for the file T1520DD8 in the library MYLIB.

```
A* CUSTOMER MASTER FILE -- T1520DD8
  A          R CUSREC          TEXT('Customer master record')
  A          CUST             5  TEXT('Customer number')
  A          NAME            20  TEXT('Customer name')
  A          ADDR            20  TEXT('Customer address')
  A          CITY            20  TEXT('Customer city')
  A          STATE           2   TEXT('State abbreviation')
  A          ZIP              5 0  TEXT('Zip code')
  A          ARBAL           10 2  TEXT('Accounts receivable balance')
  A          K CUST
A*
A*
```

Figure 102. T1520DD8 – DDS Source for Customer Records

Program T1520EDF uses the `#pragma mapinc` directive to generate the file field structure that is defined in T1520DD8.


```

/* This program contains the #pragma mapinc directive to          */
/* include the externally described database file T1520DD8.      */
/* This program reads customer information from a terminal and issues */
/* a warning message if the customer's balance is less than $1000. */

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>

#pragma mapinc("custmf","QGPL/T1520DD8(cusrec)","both key","_P")
#include "custmf"

int main(void)
{

    /* Declare x of data structure type QGPL_T1520DD8_CUSREC_both_t. */
    /* The data structure type was defined from the DDS specified.    */

    QGPL_T1520DD8_CUSREC_both_t x;

    /* Get information from entry.                                    */

    printf("Please type in the customer name (max 20 char).\n");
    gets(x.NAME);
    printf("Please type in the customer balance.\n");
    scanf("%D(10,2)",&x.ARBAL);

    /* Prints out warning message if x.ARBAL<1000.                */

    if (x.ARBAL<1000)
    {
        printf("%s has a balance less than $1000!\n", x.NAME);
    }
}

```

Figure 103. T1520EDF – ILE C Source to Include an Externally Described Database File

The type definitions are created in your ILE C source listing that is based on the #pragma directive that is specified in the ILE C source program.

The output is as follows:

```

Please type in the customer name (max 20 char).
> James Smith
Please type in the customer balance.
> 250.58
James Smith has a balance less than $1000!
Press ENTER to end terminal session.

```

The DDS part of the program listing is as follows:

```

/* -----*/
/* PHYSICAL FILE: QGPL/T1520DD8 */
/* FILE CREATION DATE: 93/08/14 */
/* RECORD FORMAT: CUSREC */
/* FORMAT LEVEL IDENTIFIER: 4E9D9ACA60E00 */
/* -----*/
typedef _Packed struct {
    char CUST[5]; /* Customer number */
    char NAME[20]; /* Customer name */
    char ADDR[20]; /* Customer address */
    char CITY[20]; /* Customer city */
    char STATE[2]; /* State abbreviation*/
    decimal(5,0) ZIP; /* Zip code */
    decimal(10,2) ARBAL; /* PACKED SPECIFIED IN DDS */
    /* Accounts receivable balance*/
    /* PACKED SPECIFIED IN DDS */
}QGPL_T1520DD8_CUSREC_both_t;
typedef _Packed struct {
    char CUST[5];
    /* DDS - ASCENDING*/
    /* STRING KEY FIELD*/
}QGPL_T1520DD8_CUSREC_key_t;

```

Figure 104. Output Listing from Program T1520EDF – Customer Master Record

Using Long Names for Files

The `#pragma mapinc` directive supports file names up to 128 characters long and record field names up to 30 characters long.

The `LNAME` option was added to `#pragma mapinc` to support SQL long name formats. SQL long names map to a 10-character short file name, which consists of the first 5 characters of the name followed by a 5-digit unique number. For example, the system short name for `LONGSQLTABLENAME` is `LONGS00001`.

Long record field names are not mapped to a 10-character short name. When the `LNAME` option is specified it is assumed that the long name format for the file name is being used. If the file name has more than 10 characters, this name is converted to the associated short name internally. This short name is used to extract the external file definition. When a regular short name of 10 characters or less is specified, no conversion occurs.

C The `#pragma mapinc` directive uses the 30 character record field names in the type definitions that are generated, with or without the `LNAME` option that is specified. For the filenames that are specified using a long name format, the type definitions that are generated use the associated regular 10-character short filename.

C++ C++ users must use the `GENCSRC` utility to create type definitions from an external file.

C C users can use either the `GENCSRC` utility or the `#pragma mapinc` directive to create type definitions from an external file.

Note: For more information on the differences between the `GENCSRC` utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

Level Checking to Verify Descriptions

When an ILE C/C++ program that uses externally described files is compiled, the compiler extracts the record-level and field-level descriptions for the files referred to in the program and makes those descriptions part of the compiled program. When you run the program, you can verify that the descriptions with which the program was compiled are the current descriptions. This process is referred to as *level checking*.

When it creates the associated header file, the server assigns a unique level identifier for each record format. The following information determines the level identifier:

- Record format name
- Field name
- Total length of the record format

- Number of fields in the record format
- Field attributes (for example, length and decimal positions)
- Order of the field in the record format

Note: It is possible for files with large record formats (many fields) to have the same format level identifiers even though their formats may be slightly different. Problems can occur when copying these files if the record format names of the from-file and the to-file are the same.


If you change any of the data description specification (DDS) items in the preceding list, the level identifier changes.

When you create or change files, and you specify that you want level checking:

- The system checks the level identifier to determine whether the description of the record format you are using was changed since the program was compiled.
- If that information has changed so much that your program cannot process the file, the system notifies your program of this condition.

If the changes affect a field that your program uses, you must compile the program again for it to run properly.

 C++ users must use the GENCSRC utility for level checking.

 C users can use either the GENCSRC utility or the `#pragma mapinc` directive for level checking.

Note: For more information on the differences between the GENCSRC utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

If you make changes that do not affect the fields that your program uses, you can run the program without compiling again by entering an override command for the file and specifying `LVLCHK(*NO)`. For example, suppose that you add a field to the end of a record format in a database file, but the program does not use the new field. You can use the Override with Database File (OVRDBF) command with `LVLCHK(*NO)` to enable the program to run without compiling again.

Note: The Override with Database File (OVRDBF) command can be used with DDM to override a local database file named in the program with a DDM file. The DDM file causes the associated remote file to be used by the program instead of the local database file.

The use of level checking ensures file integrity. It alerts you to the possibility of unpredictable results.

An alternative to level checking is to display and analyze the file description to determine if the changes affect your program. You can use the Display File Field Description (DSPFFD) command to display the description or, if you have the source entry utility (SEU), you can display the source file containing the DDS for the file. To display the format level identifier defined in the file, use the Display File Description (DSPFD) command.

Note: When you are displaying the level identifier, remember that the record format identifier is compared, rather than the file identifier.

Using the GENCSRC Utility for Level Checking

Use the GENCSRC utility to retrieve externally described file information for use in a C/C++ program. The utility:

- Creates a C/C++ header file which contains the type definition structure for the include file.
- Supports creation of C/C++ include files.

Use the `SLTFLD` keyword to turn on level checking.

Note: For a list of options for the `SLTFLD` keyword, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

If you specify the keyword `SLTFLD` with value `*LVLCHK` on the GENCSRC command, in addition to generating the type `_LVLCHK_T` (array of structures), a variable of type `_LVLCHK_T` is also generated. The name of this variable of type `_LVLCHK_T` is based on some or all of the following:

- File name specified for the OBJ keyword (see [Figure 105 on page 186](#))
- Member name (see [Figure 106 on page 186](#))
- Value of the TYPEDEFPFX keyword (see [Figure 107 on page 187](#))
- Include name

Note: In each of the following figures, the include name is actually the include file (that is, the SRCFILE/SRCMBR or SRCSTMF keywords).

In the case when SLTFLD(*LVLCHK) is specified with the default TYPEDEFPFX(*OBJ), the name of the level check structure is based on the file name as specified in the OBJ keyword and the include name (see [Table 44 on page 423](#)). The GENCSRC command generates the level check structure named *mylib_myfile_mybr_lvlchk*, as shown in the following examples:

```
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE') SRCFILE(INCLIB/H)
SRCMBR(MYMBR) SLTFLD(*LVLCHK) TYPEDEFPFX(*OBJ)
```

or

```
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE')
SRCSTMF('includir/mybr.h) SLTFLD(*LVLCHK) TYPEDEFPFX(*OBJ)
```

```
/* ----- */
// PHYSICAL FILE : MYLIB/MYFILE
// FILE LAST CHANGE DATE : 2001/09/13
// RECORD FORMAT : FORMAT1
// FORMAT LEVEL IDENTIFIER : 38A624C5F3B51
/* ----- */

_LVLCHK_T mylib_myfile_mybr_lvlchk = {
    .
};
```

Note: The level check name depends on your source location (library, file, member).

*Figure 105. Example of SLTFLD(*LVLCHK) with the Default TYPEDEFPFX(*OBJ)*

In the case when SLTFLD(*LVLCHK) is specified with TYPEDEFPFX(*NONE), the name of the level check structure is based on the member name, and the commands in [Figure 106 on page 186](#) generate a level check structure named *mybr_lvlchk*.

```
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE') SRCFILE(INCLIB/H)
SRCMBR(MYMBR) SLTFLD(*LVLCHK) TYPEDEFPFX(*NONE)
```

or

```
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE')
SRCSTMF('includir/mybr.h) SLTFLD(*LVLCHK) TYPEDEFPFX(*NONE)
```

```
/* ----- */
// PHYSICAL FILE : MYLIB/MYFILE
// FILE LAST CHANGE DATE : 2001/09/13
// RECORD FORMAT : FORMAT1
// FORMAT LEVEL IDENTIFIER : 38A624C5F3B51
/* ----- */

_LVLCHK_T mybr_lvlchk = {
    .
};
```

Note: The level check name depends on your source location (library, file, member).

*Figure 106. Example of SLTFLD(*LVLCHK) with the Default TYPEDEFPFX(*NONE)*

In the case when SLTFLD(*LVLCHK) is specified with TYPEDEFPFX(*prefix_name*), the name of the level check structure is the *prefix_name* followed by the file name based on the OBJ keyword and the SRCFILE/SRCMBR or SRCSTMF keywords (the include file). The commands in [Figure 107 on page 187](#) give the level check structure named MYPREFIX_mylib_myfile_mybr_lvlchk.

```
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE') SRCFILE(INCLIB/H)
SRCMBR(MYMBR) SLTFLD(*LVLCHK) TYPEDEFPFX(MYPREFIX)
```

or

```
GENCSRC OBJ('/QSYS.LIB/MYLIB.LIB/MYFILE.FILE')
SRCSTMF('incdir/mybr.h) SLTFLD(*LVLCHK) TYPEDEFPFX(MYPREFIX)
```

```
/* ----- */
// PHYSICAL FILE : MYLIB/MYFILE
// FILE LAST CHANGE DATE : 2001/09/13
// RECORD FORMAT : FORMAT1
// FORMAT LEVEL IDENTIFIER : 38A624C5F3B5
/* ----- */

_LVLCHK_T MYPREFIX_mylib_myfile_mybr_lvlchk = {
    .
};
```

Note: The level check name depends on your source location (library, file, member).

*Figure 107. Example of SLTFLD(*LVLCHK) with the Default TYPEDEFPFX value *MYPREFIX*

Using the #pragma mapinc Directive for Level Checking



The #pragma mapinc directive provides the opportunity to convert DDS files to include files directly.

If you specify the LVLCHK option on the #pragma mapinc directive, the following are generated:

- An array of structures of type _LVLCHK_T
- A variable of type _LVLCHK_T

The array is initialized so that each array element contains the level check information for the corresponding formats specified on the #pragma mapinc directive. The last two array elements are always empty strings, one for each field of the structure.

The name of the variable is LIBRARY_FILE_INCLUDE_lvlchk, where LIBRARY, FILE, and INCLUDE are the library_name, file_name and include_name, respectively.

If you specify the lvlchk keyword on the _Ropen varparm parameter and the composition of the file is changed, the file pointer on the _Ropen returns NULL and the CPF4131 message is issued.

Note: For more information about using the LVLCHK option of the #pragma mapinc directive, see the *ILE C/C++ Compiler Reference*.

The following figure shows the #pragma mapinc directive and the LVLCHK option to perform a level check on a file when it is opened.

```

/* This program illustrates how to use level check information.      */
/* This example uses ILE C record I/O. See the ILE C              */
/* Programmer's Reference for descriptions of the record I/O      */
/* functions.                                                     */
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#pragma mapinc("DD3FILE", "MYLIB/T1520DD3(purchase)", "key lvlchk", "_P")
#include "DD3FILE"
int main(void)
{
    _RFILE    *in;
    char      new_purchase[21] = "PEAR    1002022244";
/* Open the file for processing in keyed sequence. File is created */
/* with the default access path.                                  */

    if ( (in = _Ropen("MYLIB/T1520DD3", "rr+ varparm = (lvlchk)",
        &MYLIB_T1520DD3_DD3FILE_lvlchk)) == NULL)
    {
        printf("Open failed\n");
        exit(1);
    }

/* Update the first record in the keyed sequence. The function    */
/* _Rlocate locks the record.                                     */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);
/* Force the end of data.                                         */
    _Rfeod(in);
    _Rclose(in);
}

```

Figure 108. ILE C Source Using the #pragma mapinc lvlchk Option

The following example contains the DDS in the file T1520DD3 in the library MYLIB.

```

A          R PURCHASE
A          ITEMNAME      10
A          SERIALNUM     10
A          K SERIALNUM

```

Figure 109. T1520DD3 – DDS Source for Program

The DDS part of the program listing is as follows:

```

/* -----*/
/* PHYSICAL FILE: MYLIB/T1520DD3                                */
/* FILE CREATION DATE: 93/09/02                                */
/* RECORD FORMAT: PURCHASE                                     */
/* FORMAT LEVEL IDENTIFIER: 322C4B361172D                      */
/* -----*/
typedef _Packed struct {
    char SERIALNUM[10];
/* DDS - ASCENDING*/
/* STRING KEY FIELD*/
}MYLIB_T1520DD3_PURCHASE_key_t;
typedef _Packed struct {
    unsigned char format_name[10];
    unsigned char sequence_no[13];
} _LVLCHK_T[];
_LVLCHK_T MYLIB_T1520DD3_DD3FILE_lvlchk = {
    "PURCHASE ", "322C4B361172D",
    "", "" };

```

Figure 110. Output Listing from the Program

Avoiding Field Alignment Problems in C/C++ Structures

All fields defined in ILE C/C++ structures are aligned on their natural boundaries. For example, `int` fields are four bytes long and are stored on four-byte boundaries. If you create a file that is externally described, the system does not enforce boundary alignment of the externally described data. The structure may need to be packed because packed structures match the alignment of the externally described data.

If the fields defined in the DDS are aligned (for example, all are character fields), you can use the type definition that is generated without packing the structure.


To avoid an alignment problem, specify the `_P` option to generate a packed structure. For example, to include a packed type definition structure of `input` and key fields for the record format `custrec` from the file `EXAMPLE/CUSTOMSTL`, the following statements must appear in your program in the order shown below:

```
#pragma mapinc("custmf","EXAMPLE/CUSTOMSTL(custrec)","input key","_P")
...
#include "custmf"
```

Including External Field Definitions in a Program

Response indicators are included when the DDS keyword `INDARA` is not specified. When this is the case, use the `INPUT`, `OUTPUT`, or `BOTH` option.

 C++ users must use the `GENCSRC` utility to create external file definitions.

 C users can use either the `GENCSRC` utility or the `#pragma mapinc` directive to create external file definitions.

Note: For more information on the differences between the `GENCSRC` utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

When the DDS shown in [Figure 111 on page 189](#) is included in your ILE C program, the structure definition shown in [Figure 112 on page 189](#) is generated.

```
#pragma mapinc("test","example/phonelist(phone)","input","")
#include "test"
A          R PHONE
A
A          CF03(03 'EXIT')
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          7 28'Name:'
A          NAME          11A I 7 34
A          9 25'Address:'
A          ADDRESS      20A I 9 34
A          11 25'Phone #:'
A          PHONE_NUM    8A I 11 34
A          23 34'F3 - EXIT'
A          DSPATR(HI)
```

Figure 111. DDS Source for a Display File

```
/* -----*/
/* DEVICE FILE: EXAMPLE/PHONELIST */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: PHONE */
/* FORMAT LEVEL IDENTIFIER: 10D2D0DB2BEE8 */
/* -----*/
typedef struct {
char IN03; /* EXIT */
char NAME[11];
char ADDRESS[20];
char PHONE_NUM[8];
}EXAMPLE_PHONELIST_PHONE_i_t;
```

Figure 112. Structure Definition for a Display File

The INPUT Option

Specify the `INPUT` option when you want to include the fields that are defined as `INPUT` or `BOTH` in the externally described device file.

Note: Option and response indicators are included in the type definition structures only if the DDS keyword `INDARA` is not specified in the external file description.

The OUTPUT Option

Specify the OUTPUT option when you want to include fields that are defined as OUTPUT or BOTH in the externally described device file.

Note: Option and response indicators are included in the type definition structures only if the DDS keyword INDARA is not specified in the external file description.

The BOTH Option

When you specify BOTH, two type definition structures are generated:

- One type definition contains all fields defined as INPUT or BOTH; the other contains all fields defined as OUTPUT, or BOTH.
- One type definition structure is generated for each format that is specified when all fields are defined as BOTH, and a separate indicator area is not specified.

Note: Option and response indicators are included in the type definition structures only if the DDS keyword INDARA is not specified in the external file description.

If you are including the external file description for only one record format, a type definition union is automatically created containing the two type definitions. The name of this type definition union is LIBRARY_FILE_FMT_both_t. If you specify a union-type-name on the #pragma mapinc directive, the name that is generated is union-type-name_t.

C++ C++ users must use the GENCSRC utility for type definitions.

C C users can use either the GENCSRC utility or the #pragma mapinc directive for type definitions.

Note: For more information on the differences between the GENCSRC utility and the #pragma mapinc directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

```
A
A          R FMT
A
A          CF01(50)
A          CF02(51)
A          CF03(99 'EXIT')
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          7 28'Name: '
A          NAME          11A I 7 34
A          ADDRESS      20A 0 9 34
A          PHONE_NUM    8A 0 11 34
A          23 34'F3 - EXIT'
```

Figure 113. DDS Source for a Device File

C When the DDS shown above is included in your ILE C program, the following structure is generated:

```
#pragma mapinc("example/screen1", "example/test(fmt)", "both", "d")
#include "example/screen1"
```



```

/* -----
/* DEVICE FILE: EXAMPLE/TEST
/* FILE CREATION DATE: 93/09/01
/* RECORD FORMAT: FMT
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7
/* -----
/* INDICATORS FOR FORMAT FMT
/* INDICATOR 50
/* INDICATOR 51
/* INDICATOR 99
/* -----
typedef struct {
    char NAME[11];
}EXAMPLE_TEST_FMT_i_t;
typedef struct {
    char ADDRESS[20];
    char PHONE_NUM[8];
}EXAMPLE_TEST_FMT_o_t;
typedef union {
    EXAMPLE_TEST_FMT_i_t EXAMPLE_TEST_FMT_i;
    EXAMPLE_TEST_FMT_o_t EXAMPLE_TEST_FMT_o;
}EXAMPLE_TEST_FMT_both_t;

```


Figure 114. Structure Definitions for a Device File

This shows the structure definitions that are created when the format FMT in the device file EXAMPLE/TEST is included in your program. The external file description contains three indicators IN50, IN51, and IN99, and the DDS keyword INDARA. The indicators will appear as comments and will not be included in the structure because the option INDICATOR was not specified in the `#pragma mapinc` directive.

Defining and Using Indicators

Indicators for a record format are allowed only for device files, and can be defined as a separate indicator structure or as a member in the record format structure.

 C++ users must use the GENCSRC utility for structure definitions.

 C users can use either the GENCSRC utility or the `#pragma mapinc` directive for structure definitions.

Note: For more information on the differences between the GENCSRC utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

Creation of Indicators in the File Buffer

If you do not specify the keyword INDARA in DDS, the indicators are created as part of the file buffer being read or written. Only those indicators that are used are declared in the type definition of the structure. They are declared as `char`, and created when the INPUT, OUTPUT, or BOTH option is specified on the `#pragma mapinc` directive.

Creating a Separate Indicator Area

To use indicators as a separate structure you must specify:

- the DDS keyword INDARA in the external description of the file
- the INDICATORS option on the `#pragma mapinc` directive or the GENCSRC command.
- use `indicators=y` when opening the file

You must also set the address of the separate indicator area by using the `_Rindara()` function for record files, before performing I/O operations.

Note: If you specify indicators on the `#pragma mapinc` directive and do not use the DDS keyword INDARA in your external file description, you will receive a warning message at compile time.

If indicators are requested, and exist in the format, a 99-byte structure definition is created. The structure definition contains a field declaration for each indicator defined in the DDS. The name of each field is

INXX, where XX is the DDS indicator number. The sequence of bytes between indicators is defined as INXX_INYY, where XX is the first undefined byte and YY is the last undefined byte in a sequence.

```

A          INDARA
A          R FMT
A          CF01(50)
A          CF02(51)
A          CF03(99 'EXIT')
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          7 28'Name:'
A          NAME          11A I 7 34
A          9 25'Address:'
A          ADDRESS      20A 0 9 34
A          11 25'Phone #:'
A          PHONE_NUM    8A 0 11 34
A          23 34'F3 - EXIT'

```

Figure 115. DDS Source for Indicators

```

#pragma mapinc("example/temp","exindic/test(fmt)","indicators","")
#include "example/temp"

```

When this DDS is included in your ILE C/C++ program, the following structure is generated:

```

/* ----- */
/* DEVICE FILE: EXINDIC/TEST */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
/* INDICATORS FOR FORMAT FMT */
/* INDICATOR 50 */
/* INDICATOR 51 */
/* INDICATOR 99 */
/* ----- */
typedef struct {
char IN01_IN49[49]; /* UNUSED INDICATOR(S) */
char IN50;
char IN51;
char IN52_IN98[47]; /* UNUSED INDICATOR(S) */
char IN99;
}EXINDIC_TEST_FMT_indic_t;

```

Figure 116. Structure Definition for Indicators

This shows a type definition of a structure for the indicators in the format FMT of the file EXINDIC/TEST. The external file description contains three indicators: IN50, IN51, and IN99. The DDS keyword INDARA is also specified in the DDS for the file.

If indicators are defined for a record format and the INDICATOR option is not specified on the #pragma mapinc directive or GENCSRC command, a list of the indicators in the DDS is included as a comment in the header description.

C The following shows the header description created when the file description for the file EXINDIC/TEST is included in your program and the indicators option is not specified on the #pragma mapinc directive.

```

/* ----- */
/* DEVICE FILE: EXINDIC/TEST */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
/* INDICATORS FOR RECORD FORMAT FMT */
/* INDICATOR 50 */
/* INDICATOR 51 */
/* INDICATOR 99 */
/* ----- */

```

Figure 117. Header Description Showing Comments for Indicators

Including Physical and Logical Database Files in a Program

To include external database file descriptions, use the INPUT, BOTH, KEY, NULLFLDS, or LVLCHK option on the `#pragma mapinc` directive. If you specify either the OUTPUT or INDICATOR option, an error message is generated.

You can include external file descriptions for Distributed (DDM) files using the same method described for database files if you specify either the INPUT, KEY, or BOTH option. If you specify OUTPUT or INDICATOR, an error message is issued.

C++ C++ users must use the GENCSRC utility for level checking and including external file descriptions.

C C users can use either the GENCSRC utility or the `#pragma mapinc` directive for level checking and external file descriptions.

Note: For more information on the differences between the GENCSRC utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

Including Device Files in a Program

To include external device file descriptions, use the INPUT, OUTPUT, BOTH, and/or INDICATOR *options* on the `#pragma mapinc` directive. Device files do not contain KEY fields. Therefore, you cannot specify the KEY option with device files.

C C users may use the GENCSRC utility to create type definitions from an external file.

C++ C++ users must use the GENCSRC utility to create type definitions from an external file.

Note: For more information on the differences between the GENCSRC utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

Including Externally Described Multiple Record Formats in a Logical File

To include multiple formats in a logical file, specify more than one record format name or (*ALL) on the `#pragma mapinc` directive. If you specify multiple formats, a header description and type definition is created for each format. If you specify a union- type-name, a union type definition is created.

C++ C++ users must use the GENCSRC utility to create type definitions from an external file.

C C users can use either the GENCSRC utility or the `#pragma mapinc` directive to create type definitions from an external file.

Note: For more information on the differences between the GENCSRC utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

The typedef union contains structure definitions created for each format. Structure definitions that are created for key fields when the key option is specified are not included in the union definition. The name of the union definition is union-type-name_t. The name you provide for the union-type-name is not folded to uppercase.

The following shows the type definitions created for a logical file with two record formats with the BOTH and KEY *options* specified. A typedef union with the tag `buffer_t` is also generated. **C**

```
#pragma mapinc("pay", "lib1/pay(fmt1 fmt2)", "both key", "", "buffer", "Pay")
#include "pay"
```

```

/* -----*/
/* LOGICAL FILE: PAY */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* -----*/
typedef struct {
    .
    .
}Pay_FMT1_both_t;
typedef struct {
    .
    .
}Pay_FMT1_key_t;

/* -----*/
/* LOGICAL FILE: PAY */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT2 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* -----*/
typedef struct {
    .
    .
}Pay_FMT2_both_t;
typedef struct {
    .
    .
}Pay_FMT2_key_t;


typedef union {
    Pay_FMT1_both_t;          Pay_FMT1_both;
    Pay_FMT2_both_t;          Pay_FMT2_both;
}buffer_t;

```

Figure 118. Structure Definition for Multiple Formats

Note: A typedef union is not created for the key fields.

If you specify *ALL, or more than one record format on the format-name parameter, structure definitions for multiple formats are created.

If you specify multiple formats, and the input, or output option, one structure is created for each format. The following shows the structure definitions that are created when you include the following statements in your program. The device file TESTLIB/FILE contains two record formats, FMT1, and FMT2. Each record format has fields defined as OUTPUT in its file description. 

```

#pragma mapinc("example","testlib/file(fmt1 fmt2)","output","z","unionex")
#include "example"

```

```


/* ----- */
/* DEVICE FILE: TESTLIB/FILE */
/* FILE CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
typedef struct {
    .
    .
}TESTLIB_FILE_FMT1_o_t;

/* ----- */
/* DEVICE FILE: TESTLIB/FILE */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT2 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA8 */
/* ----- */
typedef struct {
    .
    .
}TESTLIB_FILE_FMT2_o_t;

typedef union {
    TESTLIB_FILE_FMT1_o_t    TESTLIB_FILE_FMT1_o;
    TESTLIB_FILE_FMT2_o_t    TESTLIB_FILE_FMT2_o;
}unionex_t;

```

Figure 119. Structure Definitions for a Device File

When both are specified as an option, two structure definitions are created for each format. The following shows the structure definitions created when you include two formats, FMT1 and FMT2, for the device file EXAMPLE/TEST and specify the both option: 

```

#pragma mapinc("test", "example/test(fmt1 fmt2)", "both", "z", "unionex")
#include "test"

```

If all the fields are defined as BOTH and there are to be no indicators in the typedef struct, only one typedef struct is generated for each format specified. The following shows a separate typedef structure for input and output fields.

```

/* ----- */
/* DEVICE FILE: EXAMPLE/TEST */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT1 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA7 */
/* ----- */
typedef struct {
    .
    .
    .
}EXAMPLE_TEST_FMT1_i_t;
typedef struct {
    .
    .
    .
}EXAMPLE_TEST_FMT1_o_t;
/* ----- */
/* DEVICE FILE: EXAMPLE/TEST */
/* CREATION DATE: 93/09/01 */
/* RECORD FORMAT: FMT2 */
/* FORMAT LEVEL IDENTIFIER: 371E00A681EA8 */
/* ----- */
typedef struct {
    .
    .
    .
}EXAMPLE_TEST_FMT2_i_t;
typedef struct {
    .
    .
    .
}EXAMPLE_TEST_FMT2_o_t;
typedef union{
    EXAMPLE_TEST_FMT1_i_t    EXAMPLE_TEST_FMT1_i;
    EXAMPLE_TEST_FMT1_o_t    EXAMPLE_TEST_FMT1_o;
    EXAMPLE_TEST_FMT2_i_t    EXAMPLE_TEST_FMT2_i;
    EXAMPLE_TEST_FMT2_o_t    EXAMPLE_TEST_FMT2_o;
}unionex_t;

```

Figure 120. Structure Definitions for BOTH Option

Using Externally Described Packed Decimal Data in a Program

The ILE C/C++ compiler and library supports packed decimal data.

The `d` option of the `#pragma mapinc` directive causes the compiler to generate packed decimal variables for any packed decimal fields defined in the DDS file. If you use the `p` option of the `#pragma mapinc` directive or the `PKDDECFLD` parameter of the `GENCSRC` command, character arrays are generated to store the packed decimal values.

C++ C++ users must use the `GENCSRC` utility to create type definitions from an external file.

C C users can use either the `GENCSRC` utility or the `#pragma mapinc` directive to create type definitions from an external file.

Note: For more information on the differences between the `GENCSRC` utility and the `#pragma mapinc` directive, see [“The GENCSRC Utility and the #pragma mapinc Directive”](#) on page 423.

These character arrays then need to be converted to an ILE C/C++ numeric data type to be used in the ILE C/C++ program. If neither the `d` or `p` option is specified, the `d` option is the default. See [“Using Packed Decimal Data in a C Program”](#) on page 354 or [“Using Packed Decimal Data in a C++ Program”](#) on page 365 for examples in using packed decimal data types.

Note: If you have a DDS file with packed decimal fields defined in your program, your source code must:

- **C** Include the `<decimal.h>` header file
- **C++** Include the `<bcd.h>` header file

Note: If `<decimal.h>` is included in a source that is compiled with the C++ compiler, `<bcd.h>` is included anyway.

Use either the `#pragma mapinc d` option or the `GENCSRC PKDDECFLD` parameter in your ILE C/C++ source code.

Note: The ILE C/C++ compiler and library do not support zoned decimal data. If there are zoned fields, `#pragma mapinc` and `GENCSRC` convert them into `*CHAR` type.

To convert packed decimal data that is stored in character arrays to integer or floating point (double) and vice versa, the functions `QXXDTOP()`, `QXXITOP()`, `QXXPTOD()`, `QXXPTOI()` can be used.

To convert zoned decimal data that is stored in character arrays to integer or floating point (double) and vice versa, the functions `QXXDZOZ()`, `QXXITZOZ()`, `QXXZTOD()`, `QXXZTOI()` can be used.

The `MI cpyrv()` function can also be used to convert packed or zoned decimal data to an ILE C/C++ numeric data type. It can be used to convert an ILE C/C++ numeric data type to packed or zoned decimal data.

The conversion functions are included with the ILE C/C++ compiler so that EPM C code that uses these functions can be maintained.

If you are doing database I/O operations, you can use a logical file with integer or floating point fields to redefine packed and zoned fields in your physical file. When you perform an input, or output operation through the logical file, the IBM i converts the data for you automatically.

Using Database Files and Distributed Files in a Program

This topic describes how to:

- Copy data from one file to another using an arrival sequence access path
- Update data in a record file by using a keyed sequence access path
- Read and print records from a data file
- Specify commitment control conditions

Database Files and Distributed Files

A *database file* contains data that is stored permanently on the system. The object type is `*FILE`.

Database files can be created and used as either physical files or logical files. Database files can contain either data or source statements.

ILE C/C++ programs access files on remote systems through *distributed* (DDM). DDM allows application programs on one system to use files that are stored on a remote system as database files. No special statements are required in ILE C/C++ programs to support DDM files.

A DDM file is created by a user or program on a local (source) system. This file (with object type `*FILE`) identifies a file that is kept on a remote (target) system. The DDM file provides the information that is needed for a local IBM i to locate a remote IBM i and to access the data in the target file.

Physical Files and Logical Files

Physical files contain the actual data that is stored on an IBM i, and a description of how data is to be presented to or received from a program. They contain only one record format, and one or more members.

A physical file can have a keyed sequence access path. This means that data is presented to an ILE C/C++ program in a sequence that is based on one or more key fields in the file.

Logical files do not contain data. They contain a description of records that are found in one or more physical files. A logical file is a view or representation of one or more physical files. Logical files that contain more than one format are referred to as *multi-format* logical files.

If your program processes a logical file which contains more than one record format, you can use the `_Rformat()` function to set the format you wish to use.

Some operations cannot be performed on logical files. If you open a logical file for stream file processing with open modes `w`, `w+`, `wb`, or `wb+`, the file is opened but not cleared. If you open a logical file for record file processing with open modes `wr` or `wr+`, the file is opened but not cleared.

Describing Records in Database Files

Records in database files can be described using either a field level description or record level description.

A *field level description* includes a description of all fields and their arrangement in this record. because the description of the fields and their arrangement is kept within a database file and not in your ILE C/C++ program, database files created with a field level description are referred to as *externally described files*. See [“Using Externally Described Files in a Program”](#) on page 179.

A *record level description* describes only the length of the record, and not the contents of the record. Database files that are created with record level descriptions are referred to as *program-described files*. This means that your ILE C/C++ program must describe the fields in the record.

Defining Externally Described Files

An ILE C/C++ program can use either externally described or program-described files. If it uses an externally described file, the ILE C/C++ compiler can extract information from the externally described file, and automatically include field information in your program. Your program does not need to define the field information. For further information see [“Using Externally Described Files in a Program”](#) on page 179.

To define externally described database files, use one of the following:

- DB2® for IBM i
- Interactive Data Definition Utility (IDDU)
- Data descriptive specifications (DDS) source

A *data description specification* is a description of a database file that is entered into the system in a fixed form, and is used to create files. This description is composed of one or more record formats that define the fields that make up the record. It can also include access path information that determines the order in which records are retrieved from the file.

Data Files and Source Files

A *data file* contains actual data.

Records in data files are grouped into members. All the records in a file can be in one member, or they can be grouped into different members. Most database commands and operations by default assume that database files which contain data have only one member. This means that when your ILE C/C++ program works with database files containing data you do not need to specify the member name for the file unless your file contains more than one member.

Usually, database files that contain source programs are made up of more than one member. Organizing source programs into members within database files allows you to better manage your programs. These *source members* contain source statements that the IBM i uses to create IBM i objects. For example, a source member which contains C++ statements is used to create a program object.

Access Paths

Access paths describe the logical order of records in a file. There are two types of access paths: arrival sequence and keyed sequence.

Records that are retrieved using an *arrival sequence access path* will be retrieved in the same order in which they were added to the file. This is similar to processing sequential files. New records are physically stored at the end of the file. An arrival sequence access path is valid for both physical and logical files.

Records that are retrieved using a *keyed sequence access path* are retrieved based on the contents of one or more key fields in the record. This is similar to processing indexed or keyed files on other systems. A keyed sequence access path is updated whenever records are added, deleted, or updated, or when the contents of the key field are changed. This access path is valid for both physical and logical files.

If a file defines more than one record format, each record format may have different key fields. The default key for the file (for example, if no format is specified) will be the key fields that all record formats have in common. If there is no default key (for example, no common key fields), the first record in the file will always be returned on an input operation that does not specify the format.

Note: When your ILE C/C++ program opens a file, the default is to process the file with the access path that is used to create the file. If you specify `arrseq=N` (the default), the file is processed the way it was created. This means that if the file was created using a keyed sequence access path, your ILE C/C++ program processes the file by using a keyed sequence access path. If you specify `arrseq=Y`, the file is processed using arrival sequence. This means that even though the file was created using a keyed sequence access path, your ILE C/C++ program processes the file by using an arrival sequence access path.

Arranging Key Fields

Keyed sequence access paths can be ordered in ascending or descending sequence. When you describe a key field, the default is ascending sequence. If you are using Data Description Specifications (DDS) to create a keyed sequence file, the `DESCEND` DDS keyword can be used to specify that the key fields are to be arranged in descending sequence.

Duplicate Key Values

When a record has key fields whose contents are the same as another record's key fields in the same file, the file has records with duplicate key values. For example, if the record has two key fields *num* and *date*, duplicate key values occur when the contents of both *num* and *date* are the same in two or more records.

If you want an indication that your program is processing a record that contains a duplicate key value, specify `dupkey=y` on the call to `_Ropen()` that opens the file. If an I/O operation on a record is successful and a duplicate key value is found in that record, the `dup_key` flag in the `_RIOFB_T` structure is set. (The `_Rread()` function does not update this flag.)

Note: Using the `dupkey=y` option on the call to the `_Ropen()` function may cause your I/O operations to be slower.

You can avoid duplicate key values by specifying the keyword `UNIQUE` in the DDS file.

Deleted Records

When a database record is deleted, the physical record is marked as deleted but remains in the file. Deleted records can be overwritten by using the `_Rwrite()` function. Deleted records can be removed from a file by using the `RGZPFM` (Reorganize Physical File Member) command. They can also be reused on write operations by specifying the `REUSEDLT(*YES)` parameter on the `CRTPF` command.

Deleted records can occur in a file if the file has been initialized with deleted records using the Initialize Physical File Member (`INZPFM`) command. Once a record is deleted, it cannot be read.

Locking

The IBM i database has built-in record integrity. The system determines the lock conditions that are based on how your ILE C/C++ program opens the file. This table shows the valid open modes and the lock states that are associated with them:

<i>Table 20. Lock States for Open Modes</i>	
Open Mode	Lock State
<code>r</code> or <code>rb</code>	shared for read (*SHRRD)

Table 20. Lock States for Open Modes (continued)

Open Mode	Lock State
a, w, ab, wb, a+, r+, w+, ab+, rb+, wb+	shared for update (*SHRUPD)

You can change the lock state for a file by using the Override Database File (OVRDBF) command or the Allocate Object (ALCOBJ) command before you open the file. For example, your ILE C program can use the `system()` function to call the ALCOBJ command:

```
system("ALCOBJ OBJ((FILEA *FILE *EXCLRD))");
```

If a file is opened for update, the database locks any record read or positioned to provided the `__NO_LOCK` option is not specified. This means that the locked record cannot be locked to any other open data path, whether that open data path is opened by another program or even by the same program through another file pointer.

Successfully reading and locking another record releases the lock on a previously locked record. If the `__NO_LOCK` option is specified on any read then the lock on the previously locked record is not released. You can also release a lock on a record by using the `_Rr1s1ck()` function.

Sharing

If your application consists only of C and C++ modules, the preferred way to share a file is by opening the file in one program and passing the file pointer to the other programs. This eliminates the need to open the file more than once.

Sharing a file in the same job allows programs in that job to share the file's status, record position, and buffer. To allow an Open Data Path (ODP) to be shared between two or more programs running in the same job, use the `SHARE(*YES)` parameter on commands that create, change, or override files. An *open data path* is the path through which all I/O operations for a file are performed.

You can share open data paths for streams processed a record at a time. You can also share open data paths for record files. You should not share the open data path for streams processed a character at a time, because unpredictable results can occur when you perform I/O operations.

If you want to share a file between your C/C++ programs and programs that are written in other languages, you can do this by sharing an open data path.

The first open of a shared file determines the open mode for the file (for example, whether it is open for INPUT, OUTPUT, UPDATE, and DELETE). If a subsequent open specifies an open mode that was not specified by the first open, the file will be opened the second time but the open mode will be ignored. For example, if the first open specifies an open mode of IO and the second open specifies IOUD, the file will be opened the second time with a mode of IO.

Null-Capable Fields

The ILE C compiler allows you to process files with records that may contain fields that are considered to be null. You must specify `nullcap=Y` on the `_Ropen()` function. If a null-capable field is set to null, any data that is written into that field is not valid.

If a file is opened with `nullcap=Y`, the database provides input and output null maps and, if the file is keyed, a key null map. The input and output null maps consist of one byte for each field in the current record format of the file. These null field maps are used to communicate between the database and your program to indicate which specific fields should be considered null.

The `_RFILE` structure defined in the `<recio.h>` file contains pointers to the input, output and key null field maps, and the lengths of these maps (`null_map_len` and `null_key_map_len`).

When you write to a database file, you specify which fields are null with a character '1'. If a field is not null you specify the character '0'. This is specified in the null field map pointed to by the `out_null_map` pointer. If the file does not contain any null-capable fields, but has been opened with `nullcap=Y`, your program

must set each field in the null field map to the character '0'. This must be done prior to writing any data to the file.

When you read from a database file, the corresponding byte in the null field map is indicated with a character '1' if the field is considered null. This is specified in the null field map pointed to by the `in_null_map` pointer.

The null key field map consists of one byte for each field in the key for the current record format. If you are reading a database file by key which has null fields, you must first indicate in the null key map pointed to by `null_key_map` which fields contain null. Specify character '1' for any field to be considered null, and character '0' for the other fields.

When the `_Rupdate()` function is called to update a file which has been opened to allow null field processing, the system input buffer is used. As a result, the database requires that an input null field map be provided through the `in_null_map` pointer. Prior to calling `_Rupdate()`, the user must clear and then set the input null field map (using the `in_null_map` pointer) according to the data which will be used to update the record.

You can use the `#pragma mapinc` directive to generate typedefs that correspond to the null field maps. You can cast the null field map pointers in the `_RFILE` structures to these types to manipulate these maps. Null field macros have also been provided in the `<recio.h>` file to assist users in clearing and setting the null field maps in their programs.

Opening Database and DDM Files as Record Files

To open an IBM i database file as a record file, use the `_Ropen()` function with one of the following modes:

- `rr` or `rr+`
- `wr` or `wr+`
- `ar` or `ar+`

The valid keyword parameters for database and DDM files are:

- `arrseq`
- `blkrcd`
- `commit`
- `ccsid`
- `dupkey`
- `riofb`
- `secure`
- `varparm`
- `vlr`
- `rtncode`

Record Functions for Database and DDM Files

Use the following record functions to process database and DDM files:

- `_Rclose()`
- `_Rcommit()`
- `_Rdelete()`
- `_Rfeod()`
- `_Rformat()` (multi-format logical files)
- `_Riofbk()`
- `_Rlocate()`

- `_Ropen()`
- `_Ropnfbk()`
- `_Rreadd()`
- `_Rreadf()`
- `_Rreadk()`
- `_Rreadl()`
- `_Rreadn()`
- `_Rreadp()`
- `_Rrlslck()`
- `_Rrollbck()`
- `_Rupfbk()`
- `_Rupdate()`
- `_Rwrite()`
- `_Rwrited()`

I/O Considerations for DDM Files

DDM files may be accessed as program-described files (specify the remote file name on the `RMTFILE` parameter of the `CRTDDMF` command), or as externally described files (specify the remote DDS file name on the `RMTFILE` parameter of the `CRTDDMF` command).

Opening Database and DDM Files as Binary Stream Files

To open an IBM i database file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb`
- `wb`
- `ab`

The valid keyword parameters for database and DDM files are:

- `blksize`
- `lrecl`
- `type`
- `commit`
- `arrseq`
- `ccsid`

If you specify a database or a DDM file, the parameter `type` must be `record`.

Note: The physical database files that are created when the database file does not exist (that is, where the open mode is `wb` or `ab`) are equivalent to specifying the following CL command:

```
CRTPF FILE(filename) RCDLEN(lrecl)
```

Records in this file are created with a record length that is based on the keyword parameter `lrecl`.

The only way to create a DDM file is to use the Create DDM File (`CRTDDMF`) command. If you use the `fopen()` function with a mode of `wb` or `ab`, and the DDM file exists on the source system but the database file does not exist on the remote system, a physical database file is created on the remote system. If the DDM file does not exist on the source system, a physical database file is created on the source system.

I/O Considerations for Binary Stream Database and DDM Files

If the database file contains deleted records, the deleted records are skipped by all binary stream I/O functions.

Binary stream record-at-a-time files cannot be processed by key. As well, they can only be opened with the `rb`, `wb`, and `ab` modes.

Binary Stream Functions for Database and DDM Files

Use one of the following binary stream functions to process database files and DDM files one record at time:

- `fclose()`
- `fopen()`
- `fread()`
- `freopen()`
- `fwrite()`

Processing a Database Record File in Arrival Sequence

You can copy data from one file to another file by using an arrival sequence access path. The records are accessed in the file in the same order in which they are added to the file.

Instructions

The following example copies data from the input file `T1520ASI` to the output file `T1520ASO` by using the same order in which they are added to the file `T1520ASI`. The `_Rreadn()` and `_Rwrite()` functions are used.

1. To create an input file, enter:

```
CRTPF FILE(MYLIB/T1520ASI) RCDLEN(300)
```

This creates a physical file `T1520ASI`.

2. Type the following sample data into `T1520ASI`:

```
joe 5  
fred 6  
wilma 7
```

3. To create an output file, enter:

```
CRTPF FILE(MYLIB/T1520ASO) RCDLEN(300)
```

This creates a physical file `T1520ASO`.

4. To create the program, enter:

```
CRTBND C PGM(MYLIB/T1520ASP) SCRFILE(QCPPLE/QACSRC)
```

This creates the program `T1520ASP` that uses the source code in [Figure 121 on page 204](#).

5. To run the program, enter:

```
CALL PGM(MYLIB/T1520ASP)
```

The physical file `T1520ASO` contains the following data:

```
joe 5  
fred 6  
wilma 7
```

Source Code Sample

```
/* This program illustrates how to copy records from one file to */
/* another file, using the _Rreadn(), and _Rwrite() functions. */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define _RCDLEN 300

int main(void)
{
    _RFILE    *in;
    _RFILE    *out;
    _RIOFB_T  *fb;
    char      record[_RCDLEN];

    /* Open the input file for processing in arrival sequence.          */
    if ( (in = _Ropen("LIBL/T1520ASI", "rr, arrseq=Y")) == NULL )
    {
        printf("Open failed for input file\n");
        exit(1);
    };

    /* Open the output file.                                           */
    if ( (out = _Ropen("LIBL/T1520ASO", "wr")) == NULL )
    {
        printf("Open failed for output file\n");
        exit(2);
    };

    /* Copy the file until the end-of-file condition occurs.          */

    fb = _Rreadn(in, record, _RCDLEN, __DFT);
    while ( fb->num_bytes != EOF )
    {
        _Rwrite(out, record, _RCDLEN);
        fb = _Rreadn(in, record, _RCDLEN, __DFT);
    };

    _Rclose(in);
    _Rclose(out);
}
```

Figure 121. T1520ASP – ILE C Source to Process a Database Record File in Arrival Sequence

Note:

1. This program uses the `_Ropen()` function to open the input file T1520ASI to access the records in the same order that they are added.
2. The `_Ropen()` function also opens the output file T1520ASO.
3. The `_Rread()` function reads the records in the file T1520ASI.
4. The `_Rwrite()` function writes them to the file T1520ASO.

Processing a Database Record File in Keyed Sequence

You can update a record file by using a keyed sequence access path. The records are arranged based on the contents of one or more key fields in the record.

Example:

The following example updates data in the record file T1520DD3 by using the key field SERIALNUM. The `_Rupdate()` function is used.

1. On the command line, enter:

```
CRTPF FILE(MYLIB/T1520DD3) SRCFILE(QCPPLE/QADSSRC)
```

To create the physical file T1520DD3 that uses the following DDS source:

```

A      R PURCHASE
A      ITEMNAME    10
A      SERIALNUM   10
A      K SERIALNUM

```

Figure 122. T1520DD3 – DDS Source for Database Records

2. Enter the following sample data into T1520DD3:

```

orange 1000222200
grape  1000222010
apple  1000222030
cherry 1000222020

```

Although you enter the data as shown, the file T1520DD3 is accessed by the program T1520KSP in keyed sequence. Therefore the program T1520KSP reads the file T1520DD3 in the following sequence:

```

grape 1000222010
cherry 1000222020
apple 1000222030
orange 1000222200

```

3. On the command line, enter:

```
CRTBNDC PGM(MYLIB/T1520KSP) SRCFILE(QCPPL/ QACSRC)
```

This creates the program T1520KSP, using the following source:

```

/* This program illustrates how to update a record in a file using */
/* the _Rupdate() function. */

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE    *in;
    char      new_purchase[21] = "PEAR    1002022244";

    /* Open the file for processing in keyed sequence. File is created */
    /* with the default access path. */

    if ( (in = _Ropen("*/LIBL/T1520DD3", "rr+")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    };

    /* Update the first record in the keyed sequence. The function */
    /* _Rlocate locks the record. */

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);
    /* Force the end of data. */

    _Rfeed(in);

    _Rclose(in);
}

```

Figure 123. T1520KSP – ILE C Source to Process a Database Record File in Keyed Sequence

This program uses the `_Ropen()` function to open the record file T1520DD3. The default access path which is the keyed sequence access path is used to create the file T1520DD3. The `_Rlocate()` function locks the first record in the keyed sequence. The `_Rupdate()` function updates the record that is locked by `_Rlocate()` to PEAR 1002022244. (The first record becomes the second record in the keyed sequence access path because the key has changed.)

4. To run the program T1520KSP, enter:

```
CALL PGM(MYLIB/T1520KSP)
```

because grape is the first record in the keyed sequence, it is updated, and the data file T1520DD3 is as follows:

```
orange 1000222200  
PEAR 1002022244  
apple 1000222030  
cherry 1000222020
```

Processing a Database Record File Using Record Input and Output Functions

You can read and print records from a data file.

Example:

The following example uses the `_Ropen()`, `_Rreadl()`, `_Rreadp()`, `_Rreads()`, `_Rreadd()`, `_Rreadf()`, `_Rrslck()`, `_Rdelete()`, `_Ropnfbk()`, and `_Rclose()` record IO functions. The program T1520REC reads and prints records from the data file T1520DD4.

1. On the command line, enter:

```
CRTPF FILE(MYLIB/T1520DD4) SRCFILE(QCPPL/QADSSRC)
```

This creates the physical file T1520DD4 that uses the following DDS:

```
      A          R PURCHASE  
      A          ITEMNAME      10  
      A          SERIALNUM     10  
      A          K SERIALNUM
```

Figure 124. T1520DD4 – DDS Source for Database Records

2. Enter the following sample data into T1520DD4:

```
orange 1000222200  
grape 1000222010  
apple 1000222030  
cherry 1000222020
```

3. On the command line, enter:

```
CRTBND C PGM(MYLIB/T1520REC) SRCFILE(QCPPL/QACSRC).
```

This creates the program T1520REC that uses the following source:


```

/* This program illustrates how to use the _Rreadp(), _Rreads(),
/* _Rreadd(), _Rreadf(), _Rreadn(),
/* _Ropnfbk(), _Rdelete, and _Rrlslck() functions.
*/

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    char        buf[21];
    _RFILE      *fp;
    _XXOPFB_T   *opfb;
/* Open the file for processing in arrival sequence.
*/

    if ( ( fp = _Ropen ( "LIBL/T1520DD4", "rr+", arrseq=Y" ) ) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    };
/* Get the library and file names of the file opened.
*/
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:    %10.10s\n",
            opfb->library_name,
            opfb->file_name);
/* Get the last record.
*/
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Fourth record: %10.10s\n", *(fp->in_buf) );
/* Get the third record.
*/
    _Rreadp ( fp, NULL, 20, __DFT );
    printf ( "Third record: %10.10s\n", *(fp->in_buf) );
/* Release lock on the record so another function can access it.
*/
    _Rrlslck ( fp );
/* Read the same record.
*/
    _Rreads ( fp, NULL, 20, __DFT );
    printf ( "Same record: %10.10s\n", *(fp->in_buf) );
/* Get the second record without locking it.
*/
    _Rreadd ( fp, NULL, 20, __NO_LOCK, 2 );
    printf ( "Second record: %10.10s\n", *(fp->in_buf) );
/* Get the first record.
*/
    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record: %10.10s\n", *(fp->in_buf) );
/* Delete the second record.
*/
    _Rreadn ( fp, NULL, 20, __DFT );
    _Rdelete ( fp );
/* Read all records after deletion.
*/
    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record after deletion: %10.10s\n", *(fp->in_buf));
    _Rreadn ( fp, NULL, 20, __DFT );
    printf ( "Second record after deletion: %10.10s\n", *(fp->in_buf));
    _Rreadn ( fp, NULL, 20, __DFT );
    printf ( "Third record after deletion: %10.10s\n", *(fp->in_buf));

    _Rclose ( fp );
}

```

Figure 125. T1520REC – ILE C Source to Process a Database File Using Record I/O Functions

The `_Ropen()` function opens the file T1520DD4. The `_Ropnfbk()` function gets the library name MYLIB and file name T1520DD4. The `_Rreadl()` function reads the fourth record "cherry 1000222020". The `_Rreadp()` function reads the third record " apple 1000222030". The `_Rrlslck()` function releases the lock on this record so that `_Rreads()` can read it again. The `_Rreadd()` function reads the second record "grape 1000222010" without locking it. The `_Rreadf()` function reads the first record "orange 1000222200". The `_Rdelete()` function deletes the second record. All records are then read and printed.

4. Run the program T1520REC. On the command line, enter:

```
CALL PGM(MYLIB/T1520REC)
```

The screen output is as follows:

```

Library: MYLIB
File: T1520DD4
Fourth record: cherry
Third record: apple
Same record: apple
Second record: grape
First record: orange
First record after deletion: orange
Second record after deletion: apple
Third record after deletion: cherry
Press ENTER to end terminal session.

```

The physical file T1520DD4 contains the following data:

```

ORANGE 1000222200
APPLE 1000222030
CHERRY 1000222020

```

Synchronizing Database File Changes in a Single Job

Commitment control is a means of grouping file operations as a single unit so that you can synchronize changes to database files in a single job.

Before you can start commitment control, you must ensure that all the database files you want processed as one unit are in a single commitment control environment. All the files within this environment must be journaled to the same journal. Use the CL commands Create Journal Receiver (CRTJRNRCV), Create Journal (CRTJRN) and Start Journal Physical File (STRJRNPF) to prepare for the journaling environment.

Once the journaling environment is established, you can use the following commands:

- Start Commitment Control (STRCMTCTL)
- CALL *program-name*
- End Commitment Control (ENDCMTCTL)

You can use commitment control to define and process several changes to database files as a single transaction.

Example:

The following example uses commitment control. Purchase orders are entered and logged in two files, T1520DD5 for daily transactions, and T1520DD6 for monthly transactions. Journal entries that reflect the changes that are made to T1520DD5 and T1520DD6 are kept in the journal JRN.

1. Prepare the journaling environment:

- a. On the command line, enter:

```
CRTPF FILE(QTEMP/T1520DD5) SRCFILE(QCPPL/QADDSSRC)
```

This creates the physical file T1520DD5 using the DDS source shown below:

```

A          R PURCHASE
A          ITEMNAME    30
A          SERIALNUM   10

```

Figure 126. T1520DD5 – DDS Source for Daily Transactions

- b. On the command line enter:

```
CRTPF FILE(QTEMP/T1520DD6) SRCFILE(QCPPL/QADDSSRC)
```

This creates the physical file T1520DD6 using the DDS source shown below:

```

A          R PURCHASE
A          ITEMNAME    30
A          SERIALNUM   10

```

Figure 127. T1520DD6 – DDS Source for Monthly Transactions

c. On the command line, enter:

```
CRTPF FILE(MYLIB/NFTOBJ) RCDLEN(19)
```

This creates the physical file NFTOBJ for notification text.

Note: Notification text is sent to the file NFTOBJ when the ILE C program T1520COM that uses commitment control is run.

d. On the command line, enter:

```
CRTDSPF FILE(QTEMP/T1520DD7) SRCFILE(QCPPLE/QADSSRC)
```

This creates the display file T1520DD7 using the DDS source shown below:

```

A          DSPSIZ(24 80 *DS3)
A          REF(QTEMP/T1520DD5)
A          INDARA
A          CF03(03 'EXIT ORDER ENTRY')
A          R PURCHASE
A          3 32'PURCHASE ORDER FORM'
A          DSPATR(UL)
A          DSPATR(HI)
A          10 20'ITEM NAME      :'
A          DSPATR(HI)
A          12 20'SERIAL NUMBER :'
A          DSPATR(HI)
A          ITEMNAME R      I 10 37
A          SERIALNUM R    I 12 37
A          23 34'F3 - Exit'
A          DSPATR(HI)
A          R ERROR
A          6 28'ERROR: Write failed'
A          DSPATR(BL)
A          DSPATR(UL)
A          DSPATR(HI)
A          10 26'Purchase order entry ended'
```

Figure 128. T1520DD7 – DDS Source for a Purchase Order Display

e. On the command line, enter:

```
CRTJRNRCV JRNRCV(MYLIB/JRNRCV)
```

This creates the journal receiver JRNRCV.

Note: Journal entries are placed in JRNRCV when the application is run.

f. On the command line, enter:

```
CRTJRN JRN(MYLIB/JRN) JRNRCV(MYLIB/JRNRCV)
```

This creates the journal JRN and attaches the journal receiver JRNRCV to it.

g. On the command line, enter:

```
STRJRNPF FILE(QTEMP/T1520DD5 QTEMP/T1520DD6) JRN(MYLIB/JRN)
IMAGES(*BOTH)
```

This starts journaling the changes that are made to T1520DD5 and T1520DD6 in the journal JRN.

h. On the command line, enter:

```
CRTBNDC PGM(MYLIB/T1520COM) SRCFILE(QCPPLE/QACSRC)
```

This creates the program T1520COM using the program source shown below:

```
/* This program illustrates how to use commitment control using the */
/* _Rcommit() function and to rollback a transaction using the */
/* _Rollbck() function. */

#include <stdio.h>
#include <recio.h>
```

```

#include <stdlib.h>
#include <string.h>

#define PF03 2
#define IND_OFF '0'
#define IND_ON '1'
int main(void)
{
    char      buf[40];
    int       rc = 1;
    _SYSindara ind_area;
    _RFILE    *purf;
    _RFILE    *dailyf;
    _RFILE    *monthlyf;

    /* Open purchase display file, daily transaction file and monthly
    /* transaction file.
    if ( ( purf = _Ropen ( "*LIBL/T1520DD7", "ar+,indicators=y" ) ) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "*LIBL/T1520DD5", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    if ( ( monthlyf = _Ropen ( "*LIBL/T1520DD6", "ar,commit=y" ) ) == NULL )
    {
        printf ( "Monthly transaction file did not open.\n" );
        exit ( 3 );
    }

    /* The associate separate indicator area with the purchase file.
    _Rindara ( purf, ind_area );

    /* Select the purchase record format.
    _Rformat ( purf, "PURCHASE" );

    /* Invite the user to enter a purchase transaction.
    /* The _Rwrite function writes the purchase display.
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );
    /* While the user is entering transactions, update daily and
    /* monthly transaction files.
    while ( rc && ind_area[PF03] == IND_OFF )
    {
        rc = ( ( _Rwrite ( dailyf, buf, sizeof(buf) ) )->num_bytes );
        rc = rc && ( _Rwrite ( monthlyf, buf, sizeof(buf) ) )->num_bytes;

        /* If the databases were updated, then commit transaction.
        /* Otherwise, rollback the transaction and indicate to the
        /* user that an error has occurred and end the application.

        if ( rc )
        {
            _Rcommit ( "Transaction complete" );
        }
        else
        {
            _Rrollbck ( );
            _Rformat ( purf, "ERROR" );
        }
        _Rwrite ( purf, "", 0 );
        _Rreadn ( purf, buf, sizeof(buf), __DFT );
    }
}

```

The `_Ropen()` function opens the purchase display file, the daily transaction file, and the monthly transaction file. The `_Rindara()` function identifies a separate indicator area for the purchase file. The `_Rformat()` function selects the purchase record format defined in T1520DD7. The `_Rwrite()` function writes the purchase order display. Data that is entered updates the daily and monthly transaction files T1520DD5 and T1520DD6. The transactions are committed to these database files that use the `_Rcommit()` function.

2. Run program T1520COM under commitment control. On the command line, enter:

```
STRCMTCTL LCKLVL(*CHG) NFYOBJ(MYLIB/NFTOBJ (*FILE)) CMTSCOPE(*JOB)
```

```
CALL PGM(MYLIB/T1520COM)
```

The display appears as follows:

```
                PURCHASE ORDER FORM

                ITEM NAME      :
                SERIAL NUMBER  :

                F3 - Exit
```

3. Fill out the online Purchase Order Form, using the following sample data:

```
TABLE          12345
BENCH          43623
CHAIR          62513
```

After an item and serial number are entered, T1520DD5 and T1520DD6 files are updated. The daily transaction file T1520DD5 file contains the sample data after all three purchase order items are entered.

4. End commitment control. On the command line, enter:

```
ENDCMTCTL
```

The journal JRN contains entries that correspond to changes that are made to T1520DD5 and T1520DD6.

Blocking Records

You can use record blocking to improve the performance of I/O operations on files that are opened for input or output only. Specify the `blksize=value` parameter on a call to the `fopen()` function or the `blkrcd=y` on a call to the `_Ropen()` function to turn on record blocking. In some situations, the operating system will return only one record in the block when processing a file. In these cases there is no performance gain.

You can turn off record blocking without changing your program by specifying `SEQONLY(*YES)` on the `OVRDBF` command.

Note: When record blocking is in effect, the I/O feedback structure is updated only when a block of records is transferred between your program and the system.

Using Device Files in a Program

This topic describes how to:

- [Use IBM i feedback areas for all device files.](#)
- [Use indicators to transfer information between a program and the system.](#)
- [Establish a default program device.](#)

- [Change a default program device.](#)
- [Obtain feedback information.](#)
- [Use display files and subfiles.](#)
- [Use Intersystem Communication Files.](#)
- [Use printer files.](#)
- [Write source statements to a tape file.](#)
- [Write source statements to a diskette file.](#)
- [Use save files.](#)

Using IBM i Feedback Areas for all Device Files

To access the device attributes feedback area, use the `_Rdevatr()` function. To use stream files (`type=record`) with record I/O functions, you must cast the FILE pointer to an `_RFILE` pointer.

Using Indicators to Transfer Information

Indicators allow information to be passed from a program to the system or from the system to a program. Display, ICF, and printer files can make use of indicators. Indicators are boolean data items that can contain a character value of either 1 or 0.

This section describes:

- [Types of indicators](#)
- [Separate indicator areas](#)
- [Major and minor return codes](#)
- [Returning indicators to the file buffer](#)

For records that are either read or written by a program, you can specify indicators:

- [As part of the file buffer](#)
- [In a separate indicator area](#)

Types of Indicators

There are two types of indicators:

Option indicators pass information from a program to the system. For example, they can control which fields in a record can be displayed.

Response indicators pass information from the system to an application when an input request finishes. For example, they can be used to inform the program which function keys were pressed by the workstation user.

To use indicators, the display, ICF, and printer files must be defined as an externally described file. The data description specification (DDS) for the externally described display file must contain a one-character INDICATOR field for each indicator. Indicators are either in the records read or written by the program (the indicators are in the file buffer) or in a separate indicator area.

Separate Indicator Areas

An *indicator area* is a 99-element character array with indices from 0-98.

If you specify the INDARA keyword (`indicators=y`) in the DDS, the indicators for the display, ICF, and printer files are returned in a separate indicator area. Use the `_Rindara()` function to identify the separate indicator buffer associated with the file.

If you do not specify the INDARA keyword in the DDS, the indicators for the display, ICF, or printer file will be specified in the record buffer. The number and order of the indicators that are defined in the DDS determine the number and order of the indicators in the record buffer. Indicators are always positioned

first in the record buffer. The `in_buf` and `out_buf` pointers in the `_RFILE` structure point to the input and output record buffers for a file.

Major and Minor Return Codes

Major and minor return codes are used to report certain status information for display, ICF, and printer files. If the major return code is 00, the operation completed successfully. If an error occurs with a display, ICF, or printer file your program should handle it as it occurs.

After a read (`_Rreadindv()` or `_Rreadn()`) or write (`_Rwrite()`) operation, the `sysparm` field in the `_RIOFB_T` structure points to the major/minor return code for the display, ICF or printer files. The header file `<recio.h>` declares the `_RIOFB_T` structure.

Your program should test the return code after each I/O operation and define any error handling operations that are based on the major/minor return code.

The *Application Display Programming* manual describes major and minor return codes and their meanings for display files. The *Printer Device Programming* manual describes major and minor return codes and their meanings for printer files.

Example: Returning Indicators to a Separate Indicator Area

You can specify indicators in records to be read or written by a program in a separate indicator area using the `INDARA` keyword in DDS.

The following example illustrates how indicators are returned in a separate indicator area. The `INDARA` keyword that is specified in the DDS means that the indicator for the display is returned to a separate indicator area.

Instructions

1. To create the display file `T1520DD0` using the DDS source shown below, enter:

```
CRTDSPF FILE(MYLIB/T1520DD0) SRCFILE(QCPPL/QADSSRC)
```

[Figure 129 on page 214](#) shows the DDS source.

2. To create the program `T1520ID2`, using the source shown in [Figure 130 on page 214](#), enter:

```
CRTBNDC PGM(MYLIB/T1520ID2) SRCFILE(QCPPL/QACSRC)
```

3. To run the program `T1520ID2`, enter:

```
CALL PGM(MYLIB/T1520ID2)
```

The output is as follows:

```
                PHONE BOOK
      Name:
      Address:
      Phone #:
```

```
F3 - EXIT
```

Source Code Samples

```
A          INDARA
A          R PHONE
A          CF03(03 'EXIT')
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          7 28'Name:'
A          NAME          11A I 7 34
A          9 25'Address:'
A          ADDRESS      20A I 9 34
A          11 25'Phone #:'
A          PHONE_NUM    8A I 11 34
A          23 34'F3 - EXIT'
A          DSPATR(HI)
```

Figure 129. T1520DD0 – DDS Source for a Phone Book Display

```
/* This program uses response indicators to inform the program that */
/* F3 was pressed by a user to indicate that an input request      */
/* finished. The response indicators are returned in a separate    */
/* indicator area.                                                */
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
typedef struct{
    char name[11];
    char address[20];
    char phone_num[8];
}info;
#define IND_ON '1'
#define F3      2

int main(void)
{
    _RFILE      *fp;
    _RIOFB_T    *rfb;
    info        phone_list;
    _SYSindara  indicator_area;
    if (( fp = _Ropen ( "*LIBL/T1520DD0", "ar+ indicators=y" )) == NULL )
    {
        printf ( "display file open failed\n" );
        exit ( 1 );
    }
    _Rindara ( fp, indicator_area );
    _Rformat ( fp, "PHONE" );
    rfb = _Rwrite ( fp, "", 0 );
    rfb = _Rreadn ( fp, &phone_list, sizeof(phone_list), __DFT );
    if ( indicator_area[F3] == IND_ON )
    {
        printf ( "user pressed F3\n" );
    }
    _Rclose ( fp );
}
```

Figure 130. T1520ID2 – ILE C Source to Specify Indicators in a Separate Indicator Area

Note:

This program uses response indicators IND_ON '1' and F3 2 to inform the ILE C program T1520ID2 that a user pressed F3. The `_Rindara()` function accesses the separate indicator buffer `indicator_area` associated with the externally described file T1520DD0. The display file T1520DD0 is opened with the keyword `indicators=yes` to return the indicator to a separate area.

Example: Returning Indicators to the File Buffer

The following example shows how to specify an indicator in a record that is read by program T1520ID1. The indicator is placed in the file buffer of an externally described file. The DDS for the externally described file contains one character indicator field.

Instructions

1. To create the display file T1520DD9, enter:


```
CRTDSPF FILE(MYLIB/T1520DD9) SRCFILE(QCPPL/QADSSRC)
```

Figure 131 on page 215 shows the DDS source.

2. To create the program T1520ID1 using the program source shown in Figure 132 on page 216, enter:

```
CRTBNDC PGM(MYLIB/T1520ID1) SRCFILE(QCPPL/QACSRC)
```

3. To run the program T1520ID1, enter:

```
CALL PGM(MYLIB/T1520ID1)
```

The output is as follows:

```
                PHONE BOOK
      Name:
      Address:
      Phone #:
                F3 - EXIT
```

Code Samples

```
A          R PHONE
A
A          CF03(03 'EXIT')
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          7 28'Name:'
A          NAME          11A I 7 34
A          9 25'Address:'
A          ADDRESS      20A I 9 34
A          11 25'Phone #:'
A          PHONE_NUM    8A I 11 34
A          23 34'F3 - EXIT'
A          DSPATR(HI)
```

Figure 131. T1520DD9 – DDS Source for a Phone Book Display

```

/* This program uses a response indicator to inform the program */
/* that F3 was pressed by a user. The response indicator is returned */
/* in part of the file buffer. */
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

typedef struct{
    char in03;
    char name[11];
    char address[20];
    char phone_num[8];
}info;

#define IND_ON '1'

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *rfb;
    info phone_list;

    if (( fp = _Ropen ( "*LIBL/T1520DD9", "ar+" )) == NULL )
    {
        printf ( "display file open failed\n" );
        exit ( 1 );
    }

    _Rformat ( fp, "PHONE" );
    rfb = _Rwrite ( fp, "", 0);
    rfb = _Rreadn ( fp, &phone_list, sizeof(phone_list), __DFT );
    if ( phone_list.in03 == IND_ON )
    {
        printf ( "user pressed F3\n" );
    }
    _Rclose ( fp );
}

```

Figure 132. T1520ID1 – ILE C Source to Specify Indicators as Part of the File Buffer

Note: This program uses a response indicator IND_ON '1' to inform the ILE C program T1520ID1 that a user pressed F3.

Establishing the Default Program Device

You can establish the default device for display and ICF files.

Example:

The following example illustrates how to explicitly establish a default program device for a display file using the `_Racquire()` function.

Note: To run this example you must use a display device that is defined on your system in place of DEVICE2.

1. To create the display file T1520DDD using the DDS shown below, enter:

```
CRTDSPF FILE(MYLIB/T1520DDD) SRCFILE(QCPPL/QADDSSRC) MAXDEV(2)
```

```

A                                     DSPSIZ(24 80 *DS3)
A          R EXAMPLE
A          OUTPUT          5A  0  5  20
A          INPUT          20A  I  7  20
A                                     5 10' OUTPUT:'
A                                     7 10' INPUT:'

```

Figure 133. T1520DDD – DDS Source for an I/O Display

2. To override the file STDOUT with the printer file QPRINT, enter:

```
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
```

3. To create the program T1520DEV using the source shown below, enter:

```
CRTBNDC PGM(MYLIB/T1520DEV) SRCFILE(QCPPLE/QACSRC)

/* This program establishes a default device using the _Racquire      */
/* function.                                                         */

#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <stdlib.h>

void handler ( int );

int main(void)
{
    _RFILE    *fp;
    _RIOFB_T  *rfb;
    char      buf[21];

    signal (SIGALL, handler );

    if ( ( fp = _Ropen ( "*LIBL/T1520DDD","ar+" ) ) == NULL )
    {
        printf ( "Could not open the display file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE2" );          /* Acquire the device.      */
                                        /* DEVICE2 is now the      */
                                        /* default program device. */
    _Rformat ( fp,"EXAMPLE" );          /* Select the record       */
                                        /* format.                  */
    _Rwrite ( fp, "Hello", 5 );          /* Write to the default    */
                                        /* program device.         */

    rfb = _Rreadn ( fp, buf, 20, __DFT ); /* Read from the default   */
                                        /* program device.         */

    buf[rfb -> num_bytes] = '\0';

    printf ( "Response from device : %s\n", buf );

    _Rrelease ( fp, "DEVICE2" );
    _Rclose ( fp );
}

void handler ( int sig )
{
    printf ( "message = %7.7s\n", _EXCP_MSGID );
    printf ( "program continues \n" );
    signal ( SIGALL, handler );
}
}
```

Figure 134. T1520DEV – ILE C Source to Establish a Default Device

The `_Racquire()` function explicitly acquires the program device DEVICE2. DEVICE2 is the current program device. The `_Rformat()` function selects the record format EXAMPLE. The `_Rwrite()` function writes data to the default device. The `_Rreadn()` function reads the string from the current program device DEVICE2.

4. To run the program T1520DEV, enter:

```
CALL PGM(MYLIB/T1520DEV)
```

The output is as follows:

```

OUTPUT:  Hello
INPUT:   -----

```

5. Type GOOD MORNING on the input line and press Enter.

The file QPRINT contains:

```
Response from device : GOOD MORNING
```

Changing the Default Program Device

You can change the default device for a device file.

Example:

The following example illustrates how to change the default program device using the `_Rpgmdev()` function.

Note: To run this example you must use two display devices that are defined on your system in place of DEVICE1 and DEVICE2.

1. To create the display file T1520DDE using the DDS shown below, enter:

```
CRTDSPF FILE(MYLIB/T1520DDE) SRCFILE(QCPPL/QADSSRC) MAXDEV(2)
```

```

A                                DSPSIZ(24 80 *DS3)
A                                INVITE
A      R FORMAT1
A                                9 13'Name: '
A      NAME          20A  I  9 20
A                                11 10'Address: '
A      ADDRESS      25A  I 11 20
A      R FORMAT2
A                                9 13'Name: '
A      NAME          8A  I  9 20
A                                11 10'Password: '
A      PASSWORD     10A  I 11 20

```

Figure 135. T1520DDE – DDS Source for Name and Password Display

2. To override the file STDOUT with the printer file QPRINT, enter:

```
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
```

3. To create the program T1520CDV using the source shown below, enter:

```
CRTBNDC PGM(MYLIB/T1520CDV) SRCFILE(QCPPL/QACSRC)
```

```

/* This program illustrates how to change a default program device. */
/* using the _Racquire(), _Rpgmdev(), _Rreadindv() and _Rrelease() */
/* functions. */

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct{
    char name[20];
    char address[25];
}format1;
typedef struct{
    char name[8];
    char password[10];

```

```

}format2 ;

typedef union{
    format1 fmt1;
    format2 fmt2;
}formats ;

void io_error_check( _RIOFB_T *rfb )
{
    if ( memcmp(rfb->sysparm->_Maj_Min.major_rc,"00",2 ) ||
        memcmp ( rfb->sysparm->_Maj_Min.minor_rc,"00",2 ))
    {
        printf ( "I/O error occurred, program ends.\n" );
        exit ( 1 );
    }
}

int main(void)
{
    _RFILE    *fp;
    _RIOFB_T  *rfb;
    _XXIOFB_T *iofb;
    int       size;
    formats   buf;

    /* Open the device file. */

    if (( fp = _Ropen ( "*LIBL/T1520DDE", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" ); /* Acquire another device. */
                                /* Replace with the actual */
                                /* device name. */

    _Rformat ( fp,"FORMAT1" ); /* Set the record format for the */
                                /* display file. */

    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */

    io_error_check(rfb);

    _Racquire ( fp,"DEVICE2" ); /* Acquire another device. */

    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program */
                                /* device. Replace with the */
                                /* actual device name. */
                                /* Device2 implicitly acquired at */
                                /* open. */

    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
                                /* the display file. */

    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    io_error_check ( rfb );

    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
                                /* Read from the first device that */
                                /* enters data. The device becomes */
                                /* the default program device. */

    io_error_check ( rfb );

    /* Determine which terminal responded first. */

    iofb = _Riofbk ( fp );
    if ( !strncmp ( "FORMAT1 ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    {
        _Rrelease(fp, "DEVICE2" );
    }
    return(0);
}

```

The ILE C program T1520CDV uses the `_Racquire()` function to explicitly acquire another device that is named DEVICE1. DEVICE1 becomes the current program device. The `_Rpgmdev()` function changes the current device that is named DEVICE1 to DEVICE2. The `_Rreadindv()` function reads records from DEVICE1. The `_Release()` function releases DEVICE1 and DEVICE2.

4. To run the program T1520CDV, enter:

```
CALL PGM(MYLIB/T1520CDV)
```

The output is as follows:

```
Name :
Password:
```

When the application is run, a different display appears on each device. Data may be entered on both displays, but the data that is first entered is returned to the program. The output from the program is in QPRINT. For example, if the name SMITH and the address 10 MAIN ST is entered on DEVICE1 before any data is entered on DEVICE2, then the file QPRINT contains:

```
Data displayed on DEVICE1 is SMITH 10 MAIN ST
```

Note: There are two record formats that are created in the above example. One has a size of 45 characters (fmt1), and the other a size of 18 characters (fmt2). The union buf contains two record format declarations.

Obtaining Feedback Information

You can obtain additional information about the program devices associated with your application by using IBM i system feedback areas.

Example:

The following example uses the `_Riofbk()` function.

1. To create the display file T1520DDF using the DDS source shown below, enter:

```
CRTDSPF FILE(MYLIB/T1520DDF) SRCFILE(QCPPL/QADSSRC) MAXDEV(2)
```

```

A                                     DSPSIZ(24 80 *DS3)
A          R EXAMPLE
A          OUTPUT          5A  0  5 20
A          INPUT          20A  I  7 20
A                                     5 10'OUTPUT:'
A                                     7 10'INPUT:'
```

Figure 136. T1520DDF – DDS Source for a Feedback Display

2. To override the file STDOUT with the printer file QPRINT, enter:

```
OVRPRTF FILE(STDOUT) TOFILE(QPRINT)
```

3. To create the program T1520FBK using the source shown below, enter:

```
CRTBNDC PGM(MYLIB/T1520FBK) SRCFILE(QCPPL/QACSRC)
```

```

/* This program illustrates how to use the _Riofbk function to access */
/* the I/O feedback area. */
#include <stdio.h>
#include <recio.h>
#include <signal.h>
#include <xxfdbk.h>
#include <stdlib.h>
#include <string.h>
static void handler (int);

_RFILE *fp; /* Signal handler for _Racquire exceptions */

static void handler (int sig)
{
    _XXIOFB_T      *io_feedbk;
    _XXIOFB_DSP_ICF_T *dsp_io_feedbk;

    signal ( SIGIO, handler );

    io_feedbk = _Riofbk ( fp );
    dsp_io_feedbk = ( _XXIOFB_DSP_ICF_T *) ( (char *) (io_feedbk) +
        io_feedbk->file_dep_fb_offset );
    printf ( "Acquire failed\n" );
    printf ( "Major code: %2.2s\tMinor code: %2.2s\n",
        dsp_io_feedbk->major_ret_code, dsp_io_feedbk->minor_ret_code );
    exit ( 1 );
}

int main(void)
{
    char    buf[20];
    _RIOFB_T *rfb;

    if ( ( fp = _Ropen ( "MYLIB/T1520DDF", "ar+" ) ) == NULL )
    {
        printf ( "Could not open the display file\n" );
        exit ( 2 );
    }
    signal ( SIGIO, handler );

    _Racquire ( fp, "DEVICE1" ); /* Acquire the device. DEVICE1 is */
                                /* now the default program device. */
                                /* NOTE : If the device is not */
                                /* acquired, exceptions are issued. */
    _Rformat ( fp, "EXAMPLE" ); /* Select the record format. */
    _Rwrite ( fp, "Hello", 5 ); /* Write to default program device. */

                                /* Read from default program device. */
    rfb = _Rreadn ( fp, buf, 21, __DFT );

    printf ( "user entered: %20.20s\n", buf );

    _Rclose ( fp );
    return(0);
}

```

Figure 137. T1520FBK – ILE C Source to Use Feedback Information

This program uses two typedefs `_XXIOFB_T` for common I/O feedback, and `_XXIOFB_DSP_ICF_T` for display file specific I/O feedback. A pointer to the I/O feedback is returned by `_Riofbk (fp)`.

4. To run the program T1520FBK, enter:

```
CALL PGM(MYLIB/T1520FBK)
```

The output is as follows:

```
OUTPUT: Hello
INPUT:
```

The `signal()` function is called before an error to establish a signal handler. If an exception occurs during the acquire operation, the signal handler is called to write the major/minor return code to `stdout`.

```
Acquire failed
Major code: 82 Minor code: AA
```

Using Display Files and Subfiles

Display Files and Subfiles

A *display file* is used to define the format of the information that you wish to present on a display. It is also defines how that information is processed by the system on its way to and from the display.

A *subfile* is a display file that contains a group of records with the same record format that can be accessed by relative record number. The records of a subfile can be displayed on a display station. The system sends the entire group of records to the display in a single operation and receives the group from the display in another operation. The object type for both is `*FILE`.

To work with externally described display files use one of the following:

- DDS through the Code/400 editor or the SEU.
- Screen Design Aid (SDA) or DSU What-You-See-Is-What-You-Get (WYSIWYG) tools.

I/O Considerations for Display Files

- An ILE C/C++ program can process display files as program described files or as externally described files:
 - For program-described display files, specify all formatting and control information in the ILE C/C++ program that uses the file. To create a program-described display file, specify `SRCFILE(*NONE)` on the `CRTDSPF` command.
 - For externally described display files, specify all formatting and control information using DDS to describe the layout of the display. To create an externally described display file, specify the name of the member that contains the DDS source on the `SRCFILE` parameter of the `CRTDSPF` command.
- If you are using a user-defined data stream (UDDS), hexadecimal 3F (X'3F') blanks the display until the next display attribute. If any CCSID conversion takes place and a character cannot be mapped to the corresponding character in another code page, the character is mapped to hexadecimal 3F. This will blank the screen until the next display attribute. See [“Internationalizing a Program”](#) on page 407 for information on CCSIDs.
- The concept of clearing a file or opening a file using append mode does not apply to display files.

I/O Considerations for Subfiles

- The input typedef for record format subfiles will contain fields with the usage of I, O, B, and H. Input typedef for control record format subfiles will contain fields with the usage of I, B, and H.

- To use a subfile, you initialize it, for example, by reading records from a database file and writing them to a subfile. You must write them using `_Rwrited`.

Using Subfiles to Minimize I/O Operations

You can use subfiles to read or write a number of records to and from a display in one operation.

Example:

The following subfile example uses DDS from T1520DDG and T1520DDH to display a list of names and telephone numbers.

1. To create the display file T1520DDG using the DDS source shown below, enter:

```
CRTDSPF FILE(MYLIB/T1520DDG) SRCFILE(QCPPL/QADSSRC)
```

```

A          DSPSIZ(24 80 *DS3)
A          R SFL          SFL
A          NAME          10A B 10 25
A          PHONE        10A B   +5
A          R SFLCTL      SFLCTL(SFL)
A          SFLPAG(5)
A          SFLSIZ(26)
A          SFLDSP
A          SFLDSPCTL
A          22 25' <PAGE DOWN> FOR NEXT PAGE '
A          23 25' <PAGE UP> FOR PREVIOUS PAGE '

```

Figure 138. T1520DDG – DDS Source for a Subfile Display

2. To create the physical file T1520DDH using the DDS source shown below, enter:

```
CRTPF FILE(MYLIB/T1520DDH) SRCFILE(QCPPL/QADSSRC)
```

```

R ENTRY
R NAME      10A
R PHONE    10A

```

3. Type the following data into T1520DDH:

```

David      435-5634
Florence   343-4537
Irene      255-5235
Carrie     747-5347
Michele    634-4557

```

4. To create the program T1520SUB using the source shown below, enter:

```
CRTBNDC PGM(MYLIB/T1520SUB) SRCFILE(QCPPL/QACSRC)
```

```

/* This program illustrates how to use subfiles. */
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#define LEN      10
#define NUM_RECS 20
#define SUBFILENAME "*LIBL/T1520DDG"
#define PFILENAME   "*LIBL/T1520DDH"

typedef struct{
    char name[LEN];
    char phone[LEN];
}pf_t;
#define RECLLEN sizeof(pf_t)

void init_subfile(_RFILE *, _RFILE *);

int main(void)
{
    _RFILE      *pf;
    _RFILE      *subf;
    /* Open the subfile and the physical file. */

```

```

if ((pf = _Ropen(PFILENAME, "rr")) == NULL)
{
    printf("can't open file %s\n", PFILENAME);
    exit(1);
}
if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL)
{
    printf("can't open file %s\n", SUBFILENAME);
    exit(2);
}
/* Initialize the subfile with records from the physical file.      */
init_subfile(pf, subf);

/* Write the subfile to the display by writing a record to the      */
/* subfile control format.                                         */
_Rformat(subf, "SFLCTL");
_Rwrite(subf, "", 0);
_Rreadn(subf, "", 0, __DFT);

/* Close the physical file and the subfile.                         */
_Rclose(pf);
_Rclose(subf);
}
void init_subfile(_RFILE *pf, _RFILE *subf)
{
    _RIOFB_T    *fb;
    int         i;
    pf_t        record;

/* Select the subfile record format.                                */
_Rformat(subf, "SFL");
for (i = 1; i <= NUM_RECS; i++)
{
    fb = _Rreadn(pf, &record, RECLen, __DFT);
    if (fb->num_bytes != EOF)
    {
        fb = _Rwrited(subf, &record, RECLen, i);
        if (fb->num_bytes != RECLen)
        {
            printf("error occurred during write\n");
            exit(3);
        }
    }
}
}
}

```

This program uses `_Ropen()` to open subfile T1520DDG and physical file T1520DDH. The subfile is then initialized with records from the physical file. Subfile records are written to the display using the `_Rwrited()` function.

5. To run the program T1520SUB and see the output, enter:

```
CALL PGM(MYLIB/T1520SUB)
```

```

David      435-5634
Florence   343-4537
Irene      255-5235
Carrie     747-5347
Michele    643-4557

```

```

<PAGE DOWN> FOR NEXT PAGE
<PAGE UP> FOR PREVIOUS PAGE

```

Opening Display Files and Subfiles as Binary Stream Files

To open an IBM i display file or subfile as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- rb
- ab+
- wb
- ab

The CRTDSPF command is the only way to create a display file. If you use the `fopen()` function and the display file does not exist, a physical database file is created.

The valid keyword parameters are:

- type
- lrecl
- indicators

I/O Considerations for Binary Stream Subfiles

Only message subfiles are supported for binary stream subfiles.

Program Devices for Binary Stream Display Files

The program device that is associated with display files is a workstation. You establish the default device by implicitly acquiring it using the `fopen()` function.

Binary Stream Functions for Display Files and Subfiles

Use the following binary stream functions to process display files and subfiles:

- `fclose()`
- `fopen()`
- `fread()`
- `freopen()`
- `fwrite()`

Opening Display Files as Record Files

To open an IBM i display file or subfile as a record file, use the `_Ropen()` function with one of the following modes:

- `RR`
- `WR`
- `ar`
- `ar+`
- `rr+`
- `wr+`

The valid keyword parameters are:

- lrecl
- indicators
- secure
- riofb

I/O Considerations for Record Display Files

The program device that is associated with display files is a workstation. You establish the default device by implicitly acquiring it using the `_Ropen()` function. The implicitly acquired program device is

determined by the DEV parameter on the CRTDSPF, CHGDSPF, or OVRDSPF commands. If *REQUESTER is specified on the DEV parameter, then the program device from which the program was called is implicitly acquired. It becomes the default program device for I/O operations to the display file.

If *NONE is specified on the DEV parameter of the CRTDSPF, CHGDSPF, or OVRDSPF commands, you must explicitly acquire the program device with the `_Racquire()` function. The explicitly acquired program device now becomes the default device for subsequent I/O operations to the device file.

You can change the default program device in the following ways:

- Use the `Racquire()` function to explicitly acquire another program device. The device that is just acquired becomes the current program device.
- Use the `_Rpgmdev()` function to change the current program device that is associated with a file to a previously-acquired device. This program device can be used for subsequent input and output operations to the file.
- The actual program device that is read becomes the default device if you read from an invited device using the `_Rreadindv()` function.
- Use the `_Rrelease()` function to release a device from the file. When you release the device, it is no longer available for I/O operations.

I/O Considerations for Record Subfiles

I/O operations to the subfile record format do not cause data to appear on the display. You must read or write the subfile control record format to transfer data to or from the display. Use the `_Rformat()` function to distinguish between subfile record formats and subfile control formats. If the format you specify with the `_Rformat()` function refers to a subfile record format, no data is transferred to or from the display.

To read the next changed subfile record, use the `_Rreadnc()` function. This function searches for the next changed record from the current position in the file. If this is the first read operation, the first changed record in the subfile is read. If the end-of-file is reached before finding a changed record, EOF is returned in the `num_bytes` field of the `_RIOFB_T` structure.

Record Functions for Display Files and Subfiles

Use the following record functions to process display files and subfiles:

- `_Racquire()`
- `_Rclose()`
- `_Rdevatr()`
- `_Rfeed()`
- `_Rformat()`
- `_Rindara()`
- `_Riofbk()`
- `_Ropen()`
- `_Ropnfbk()`
- `_Rpgmdev()`
- `_Rread()` (subfiles)
- `_Rreadindv()`
- `_Rreadn()`
- `_Rreadnc()` (subfiles)
- `_Rrelease()`
- `_Rupdate()` (subfiles)
- `_Rupfbk()`

- `_Rwrite()`
- `_Rwrited()` (subfiles)
- `_Rwriterd()`
- `_Rwrrread()`

Using Intersystem Communication Function Files

An Intersystem Communications Function (ICF) file defines the layout of the data sent and received between two programs on different systems and links you to the configuration objects that are used to communicate with a remote system. The *ICF Programming* manual contains information about ICF files.

I/O Considerations for Intersystem Communication Function Files

- An ILE C/C++ program can process ICF files as program described files or as externally described files (the system file QSYS/QICDMF contains a system-supplied record format).
- The concept of clearing or opening a file using append mode does not apply to ICF files. If you open an ICF file using append mode (ar+ or ab+), the file is opened for input and output.
- If you want to write a variable length of data, you must use the keyword VARLEN in the DDS.
- ICF locale mode can be disabled at the application level by setting the maximum program devices number to 2 or greater for all ICF files on the CRTICFF command.

Opening ICF Files as Binary Stream Files

To open an IBM i ICF file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb`
- `wb`
- `ab`
- `ab+`

Note: The only way to create an ICF file is to use the CRTICFF command. If you use the `fopen()` function and the ICF file does not exist, a physical database file is created.

The valid keyword parameters are:

- `type`
- `indicators`

I/O Considerations for Binary Stream ICF Files

The `fwrite()` function returns the number of elements that are successfully written. When you use PDATA, the value returned by the `fwrite()` function does not take PDATA into consideration. When using PDATA, `errno` is set to ETRUNC even though all the data was successfully written.

Program Devices for Binary Stream ICF Files

The program device that is associated with ICF files is a communications session. You establish the default device by implicitly acquiring it using the `fopen()` function.

Binary Stream Functions for ICF Files

Use the following binary stream functions to process ICF files:

- `fclose()`
- `fopen()`

- `fread()`
- `freopen()`
- `fwrite()`

Opening ICF Files as Record Files

To open an IBM i ICF file as a record file, use the `_Ropen()` function with one of the following modes:

- `r`
- `r+`
- `w+`
- `a+`
- `w`
- `a`

The valid keyword parameters are:

- `indicators`
- `riofb`
- `secure`

I/O Considerations for Record ICF Files

The `_Rwrite()` function returns the number of characters that are successfully transferred across a communication line. When you use `PDATA`, unlike the `_fwrite()` function, the value that is returned by the `_Rwrite()` function (`num_bytes`) includes `PDATA`.

Program Devices for Record ICF Files

The program device that is associated with ICF files is a communications session. You establish the default device by implicitly acquiring it using the `_Ropen()` function. The implicitly acquired program device is determined by the `ACQPGMDEV` parameter on the `CRTICFF`, `OVRICFF`, or `CHGICFF` commands. If the program device name is specified on the `ACQPGMDEV` parameter the program device must be defined to the device file before it is opened. This is done by specifying the name on the `PGMDEV` parameter of the `ADDICFDEVE` or `OVRICFDEVE` commands.

If `*NONE` is specified for the `ACQPGMDEV` parameter of the `CRTICFF`, `OVRICFF`, or `CHGICFF` commands, you must explicitly acquire the program device using the `_Racquire()` function.

You can change the default program device in the following ways:

- Use the `_Racquire()` function to explicitly acquire another program device. The device that is just acquired becomes the current program device.
- Use the `_Rpgmdev()` function to change the current program device associated with a file to a previously-acquired device. This program device can be used for subsequent input and output operations to the file.
- The actual program device read becomes the default device if you read from an invited device using the `_Rreadindv()` function.
- Use the `_Rrelease()` function to release a device from the file. When you release the device, it is no longer available for I/O operations.

To release a program device, use the `_Rrelease()` function (the program device must have been previously acquired). This detaches the device from an open file; I/O operations can no longer be performed for this device. If you wish to use the device after releasing it, it must be acquired again.

All program devices are implicitly released when you close the file. If the device file has a shared open data path, the last close operation releases the program device.

Record Functions for ICF Files

Use the following record functions to process ICF files:

- `_Racquire()`
- `_Rclose()`
- `_Rdevatr()`
- `_Rfeod()`
- `_Rformat()`
- `_Rindara()`
- `_Riofbk()`
- `_Ropen()`
- `_Ropnfbk()`
- `_Rpgmdev()`
- `_Rreadindv()`
- `_Rreadn()`
- `_Rrelease()`
- `_Rupfbk()`
- `_Rwrite()`
- `_Rwriterd()`
- `_Rwrread()`

Example:

The following example gets a user ID and password from a source program and sends it to a target program. The target program checks the user ID and password for errors and sends a response to the source program.

Note: To run this example the target program T1520TGT must exist on a remote system. A communications line between the source system with program T1520ICF and the target system with program T1520TGT must be active. You also need Advanced Program to Program Communications (APPC).

1. To create the physical file T1520DDA, enter:

```
CRTPF FILE(MYLIB/T1520DDA) SRCFILE(QCPPLE/QADDSSRC)
```

```
      A                                UNIQUE
      A      R PASSWRDF
      A      USERID          8A
      A      PASSWRD         10A
      A      K USERID
```

Figure 139. T1520DDA – DDS Source for Password and User ID

2. To create the ICF file T1520DDB using the DDS source shown below:, enter:

```
CRATICFF FILE(MYLIB/T1520DDB) SRCFILE(QCPPLE/QADDSSRC)
ACQPGMDEV(CAPPC2)
```

```
      A      R SNDPASS
      A      FLD1            18A
      A      R CHKPASS
      A      FLD1            1A
      A      R EVOKPGM
      A
      A                                EVOKE(MYLIB/T1520TGT)
      A                                SECURITY(2 'PASSWRD' +
      A                                3 'USRID')
```

Figure 140. T1520DDB – DDS Source to Send Password and User ID

3. To create the ICF file T1520DDC using the DDS source shown below, enter:

```
CRTICFF FILE(MYLIB/T1520DDC) SRCFILE(QCPPLE/QADDSSRC) ACQPGMDEV(CAPPC1)
```

```

A      R RCVPASS
A      UID          8A
A      PWD          10A
A      R VRYPASS
A      CHKPASS     1A

```

Figure 141. T1520DDC – DDS Source to Receive Password and User ID

4. Create an intrasystem device INTRAC. From the command line, enter:

```
CRTDEVINTR DEVD(INTRAC) RMTLOCNAME(INTRAC) ONLINE(*NO)
```

5. Vary on the intrasystem device INTRAC. From the command line, enter:

```
VRYPFG CFGOBJ(INTRAC) CFGTYPE(*DEV) STATUS(*ON) RANGE(*OBJ)
```

6. To add a program device entry for ICF file T1520DDB, enter:

```
ADDICFDEVE FILE(MYLIB/T1520DDB) PGMDEV(CAPPC2) RMTLOCNAME( CAPPC1)
MODE(CAPPCMOD)
```

7. To add a program device entry for ICF file T1520DDC, enter:

```
ADDICFDEVE FILE(MYLIB/T1520DDC) PGMDEV(CAPPC1) RMTLOCNAME(*REQUESTER)
MODE(CAPPCMOD)
```

8. To create the program T1520ICF using the source shown below, enter:

```
CRTBNDC PGM(MYLIB/T1520ICF) SRCFILE(QCPPLE/QACSRC)
```

```

/* This program sends a userid and password to a target program */
/* on another system. The target program returns the userid and */
/* password. This program verifies the returned values. */
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define ID_SIZE 8
#define PASSWD_SIZE 10
#define RCD_SIZE ID_SIZE + PASSWD_SIZE
#define ERROR '2'
#define VALID '1'

_RFILE *fp;
void ioCheck(char *majorRc)
{
    if ( memcmp(majorRc, "00", 2) != 0 )
    {
        printf("Fatal I/O error occurred, program ends\n");
        _Rclose(fp);
        exit(1);
    }
}

int main(void)
{
    _RIOFB_T *fb;
    char idPass[RCD_SIZE];
    char buf[RCD_SIZE + 1];
    char passwordCheck=ERROR;

    /* Open the source file T1520DDB. */
    if ( (fp = _Ropen("*/LIBL/T1520DDB", "ar+") == NULL )
    {
        printf("Could not open SOURCE ICF file\n");
        exit(2);
    }

```



```

}

/* Start the target program T1520TGT.                               */
_Racquire(fp, "DEV1"); /* acquire device */
_Rformat(fp, "EVOKPGM");
fb = _Rwrite(fp, "", 0);
ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* Get the user-id and password.                                     */
memset(idPass, ' ', RCD_SIZE);
printf("Enter user-id (maximum 8 characters):\n");
scanf("%s", buf);
memcpy(idPass, buf, strlen(buf));
printf("Enter password (maximum 10 characters):\n");
scanf("%s", buf);
memcpy(idPass + ID_SIZE, buf, strlen(buf));

/* Send data to the TARGET program T1520TGT.                       */
_Rformat(fp, "SNDPASS");
fb = _Rwrite(fp, idPass, RCD_SIZE);
ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* Receive data from TARGET program T1520TGT.                     */
_Rformat(fp, "CHKPASS");
fb = _Rreadn(fp, &passwordCheck, 1, _DFT);
ioCheck(fb->sysparm->_Maj_Min.major_rc);

/* If a problem, such as a communications line is down, occurs in the */
/* TARGET program, then end the program.                             */
/* Otherwise, print the password verification.                       */

if ( passwordCheck == ERROR )
{
    _Rclose(fp);
    exit(3);
}
else if ( passwordCheck == VALID )
{
    printf("Password valid\n");
}
else
{
    printf("Password invalid\n");
}
_Rclose(fp);
return(0);
}

```

The `_Ropen()` function opens the record file T1520DDB. The `_Rformat()` function accesses the record format EVOKPGM in the file T1520DDB. The EVOKE statement in T1520DDB calls the target program T1520TGT. The `_Rformat()` function accesses the record format SNDPASS in the file T1520DDB. The user ID and password is sent to the target program T1520TGT. The `_Rformat()` function accesses the record format CHKPASS in the file T1520DDB. The received password and user ID is then verified.

9. To create the program T1520TGT using the following source, enter:

```
CRTBND C PGM(MYLIB/T1520TGT) SRCFILE(QCPPLE/QACSRC)
```

```

/* This program checks the userid and password.                     */
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

#define ID_SIZE      8
#define PASSWD_SIZE 10
#define RCD_SIZE    ID_SIZE + PASSWD_SIZE
#define ERROR      '2'
#define VALID      '1'
#define INVALID    '0'

int main(void)
{
    _RFILE *icff;
    _RFILE *pswd;
    _RIOFB_T *fb;

```

```

char      rcv[RCD_SIZE];
char      pwrld[RCD_SIZE];
char      vry;

/* Open the TARGET file T1529DDC. */
if ( (icff = _Ropen("QGPL/T1520DDC", "ar+")) == NULL )
{
    printf("Could not open TARGET icf file T1520DDC\n");
    exit(1);
}
/* Open the PASSWORD file T1520DDA. */
if ( (pswd = _Ropen("QGPL/T1520DDA", "rr")) == NULL )
{
    printf("Could not open PASSWORD file T1520DDA\n");
    exit(2);
}
/* Read the information from the SOURCE program T1520ICF. */
_Racquire(icff, "DEV1");
_Rformat(icff, "RCVPASS");
fb = _Rreadn(icff, &rcv, RCD_SIZE, __DFT);

/* Check for errors and send response to SOURCE program. */
if ( memcmp(fb->sysparm->_Maj_Min.major_rc, "00", 2) != 0 )
{
    vry = ERROR;
}
else
{
    fb = _Rreadk(pswd, &pwrld, RCD_SIZE, __DFT, &rcv, ID_SIZE);

    if ( fb->num_bytes == RCD_SIZE &&
        memcmp(pwrld + ID_SIZE, rcv + ID_SIZE, PASSWD_SIZE) == 0 )
    {
        vry = VALID;
    }
    else
    {
        vry = INVALID;
    }
}
_Rformat(icff, "VRYPASS");
_Rwrite(icff, &vry, 1);
_Rclose(icff);
_Rclose(pswd);
return(0);
}

```

The `_Ropen()` function opens the file T1520DDC. The `_Ropen()` function opens the password file T1520DDA. The `_Rformat()` function accesses the record format RCVPASS in the file T1520DDC. The `_Rreadn()` function reads the password and user ID from the source program T1520ICF. Errors are checked, and a response is sent to the source program T1520ICF.

10. To run the program T1520ICF, enter:

```
CALL PGM(MYLIB/T1520ICF)
```

After calling the program, you may enter a user ID and password. If the password is correct, Password valid appears on the display; if it is incorrect, Password invalid appears.

The output is as follows:

```
Password valid
Press ENTER to end terminal session.
```

Using Printer Files

A printer device file can be accessed with a program-described file (specify SRCFILE(*NONE) on the CRTPRTF command) or with an externally described file. The object type is *FILE. The *ADTS/400: Advanced Printer Function* manual contains information on printer files.

Program-described files allow First Character Forms Control (FCFC). To use this, include the First Character Forms Control code in the first position of each data record in the printer file. You must use a printer stream file and the `fwrite()` function.

To work with externally described printer files, use one of:

- DDS through the SEU or CODE/400 editor.
- Report Layout Utility(RLU) or DSU WYSIWYG tools.

I/O Considerations for Printer Files

If you wish to use First Character Forms Control, you must use program described printer files.

Opening Printer Files as Binary Stream Files

To open an IBM i printer file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `wb`
- `ab`

Note: The only way to create a printer file is to use the CRTPRTF command. If you use the `fopen()` function and the printer file does not exist, a physical database file is created.

The valid keyword parameters are:

- `type`
- `lrecl`
- `indicators`
- `recfm`

Binary Stream Functions for Printer Files

Use the following binary stream functions to process printer files:

- `fclose()`
- `fopen()`
- `freopen()`
- `fwrite()`

Opening Printer Files as Record Files

To open an IBM i printer file as a record file, use the `_Ropen()` function with one of the following modes:

- `wr`
- `ar`

The valid keyword parameters are:

- `lrecl`
- `indicators`
- `riofb`
- `secure`

Record Functions for Printer Files

Use the following record functions to process printer files:

- `_Rclose()`
- `_Rfeed()`
- `_Rformat()`
- `_Rindara()`
- `_Riofbk()`
- `_Ropen()`
- `_Ropnfbk()`
- `_Rupfbk()`
- `_Rwrite()`

Example:

The following example uses First Character Forms Control in a program described printer file. Employees' names and serial numbers are read from a physical file and written to the printer file.

1. To create the printer file T1520FCP, enter:

```
CRTPRTF FILE(MYLIB/T1520FCP) CTLCHAR(*FCFC) CHLVAL((1 (13)))
```

2. To create the physical file T1520FCI, enter:

```
CRTPF FILE(MYLIB/T1520FCI) RCDLEN(30)
```

3. Type the names and serial numbers as follows into T1520FCI:

```
Jim Roberts      1234567890  
Karen Smith     2314563567  
John Doe        5646357324
```

4. To create the program T1520FCF using the source shown below, enter:

```
CRTBND C PGM(MYLIB/T1520FCF) SRCFILE(QCPPLE/QACSRC)
```

```

/* This program illustrates how to use a printer stream file, the */
/* _fwrite() function and the first character forms control.      */
#include <stdio.h>
#include <string.h>
#define BUF_SIZE 53
#define BUF_OFFSET 20

int main(void)
{
    FILE      *dbf;
    FILE      *prtf;
    char buf   [BUF_SIZE];
    char tmpbuf [BUF_SIZE];

    /* Open the printer file using the first character forms control. */
    /* recfm and lrecl are required.                                   */
    prtf = fopen ("*LIBL/T1520FCP", "wb type=record recfm=fa lrecl=53" );
    dbf = fopen ("*LIBL/T1520FCI", "rb type=record blksize=0" );

    /* Print out the header information.                             */
    memset ( buf, ' ', BUF_SIZE );

    /* Use channel value 1.                                         */
    strncpy ( buf, "1 EMPLOYEE INFORMATION",47 );
    fwrite ( buf, 1, BUF_SIZE, prtf );

    /* Use single spacing.                                         */
    strncpy ( buf, "-----",47 );
    fwrite ( buf, 1, BUF_SIZE, prtf );
    /* Use triple spacing.                                         */
    strncpy ( buf, "          NAME          SERIAL NUMBER"
                ,BUF_SIZE );
    fwrite ( buf, 1, BUF_SIZE, prtf );
    strncpy ( buf, "          ----          -----"
                ,BUF_SIZE );
    fwrite ( buf, 1, BUF_SIZE, prtf );

    /* Print out the employee information.                           */
    while ( fread ( tmpbuf, 1, BUF_SIZE, dbf ))
    {
        memset ( buf, ' ', BUF_SIZE );

        /* Use double spacing.                                     */
        buf[0] = '0';
        strncpy ( buf + BUF_OFFSET, tmpbuf, strlen(tmpbuf) );
        fwrite ( buf, 1, BUF_SIZE, prtf );
    }
    fclose ( prtf );
    fclose ( dbf );
}

```

Figure 142. T1520FCF – ILE C Source to Use First Character Forms Control

The `fopen()` function opens the printer stream file T1520FCP using record at a time processing. The `fopen()` function also opens the physical file T1520FCI for record at a time processing. The `strncpy()` function copies the records into the print buffer. The `fwrite()` function prints out the employee records.

- To run the program T1520FCF, enter:

```
CALL PGM(MYLIB/T1520FCF)
```

The output file is as follows:

```

                EMPLOYEE INFORMATION
                -----
                NAME          SERIAL NUMBER
                ----          -----
                Jim Roberts   1234567890
                Karen Smith   2314563567
                John Doe      5646357324

```

The printed output file is as follows:

EMPLOYEE INFORMATION	
NAME	SERIAL NUMBER
----	-----
Jim Roberts	1234567890
Karen Smith	2314563567
John Doe	5646357324

Writing to a Tape File

You can write records to a tape file. A tape file is a device file that is used for tape units. The object type is *FILE.

I/O Considerations for Tape Files

An ILE C/C++ program can only process tape files sequentially. An ILE C/C++ program can only process tape files as program described files.

Opening Tape Files as Binary Stream Files

To open an IBM i tape file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb`
- `wb`
- `ab`

Note: The only way to create a tape file is to use the CRTTAPF command. If you use the `fopen()` function and the tape file does not exist, a physical database file is created.

The valid keyword parameters are:

- `type`
- `lrecl`
- `recfm`
- `blksize`

I/O Considerations for Binary Stream Tape Files

Blocking Binary Stream Tape Files

If your program processes tape files, performance can be improved if records are blocked.

Note: The value you specify on the `blksize` parameter for the `fopen()` function overrides the one you specified on the CRTTAPF or CHGTAPF commands. You can still override the BLKLEN parameter with the OVRTAPF command.

If you specify 0 on either BLKLEN or `blksize` the system calculates a block size for you. You can specify a value on either parameter of between 0 and 32 767 characters.

Binary Stream Functions for Tape Files

Use the following binary stream functions to process tape files:

- `fclose()`
- `fopen()`

- `fread()`
- `freopen()`
- `fwrite()`

Opening Tape Files as Record Files

To open an IBM i tape file as a record file, use the `_Ropen()` function with one of the following modes:

- `rr`
- `wr`
- `ar`

The valid keyword parameters are:

- `blkrcd`
- `lrecl`
- `secure`
- `riofb`

I/O Considerations for Record Tape Files

Using _Rfeed

The `_Rfeed()` function is valid for files opened for input and output operations with tape record files. For input operations, it returns end-of-file and positions the tape at the last volume in the file. For output operations, it forces all unbuffered data to be written to the tape.

Using _Rfeov

The `_Rfeov()` function is valid for tape record files opened for input and output operations. For input operations, it signals the end-of-file and positions the tape at the next volume. For output operations, any unwritten data is forced to the tape. An end-of-volume trailer is written to the tape which means that no data can be written after this trailer. Any write operations that take place after the `_Rfeov()` function occur on a new volume.

Blocking Record Tape Files

If your program processes tape files, performance can be improved if I/O operations are blocked. To block records, use the `blkrcd=Y` keyword on the `_Ropen()` function.

Record Functions for Tape Files

Use the following record functions to process tape files:

- `_Rclose()`
- `_Rfeed()`
- `_Rfeov()`
- `_Riofbk()`
- `_Ropen()`
- `_Ropfbk()`
- `_Rreadn()`
- `_Rupfbk()`
- `_Rwrite()`

Example:

The following example illustrates how to write to a tape file.

1. To create the tape file T1520TPF, enter:

```
CRTTAPF FILE(MYLIB/T1520TPF) DEV(TAP01) SEQNBR(*END)
LABEL(CSOURCE) FILETYPE(*SRC)
```

2. To create the source physical file QCSRC with the member CSOURCE, enter:

```
CRTSRCPF FILE(MYLIB/QCSRC) MBR(CSOURCE)
```

The CRTSRCPF command creates the physical file QCSRC with member CSOURCE in MYLIB. The following statements are copied to the tape file:

```
/* This program SQITF is called by the command SQUARE. This      */
/* program then calls another ILE C program SQ to perform      */
/* calculations and return a value.                            */
#include <stdio.h>
#include <decimal.h>
#pragma linkage(SQ, OS)      /* Tell compiler this is external call, */
                             /* do not pass by value.          */
int SQ(int);
int main(int argc, char *argv[])
{
    int *x;
    int result;
    x = (int *) argv[1];
    result = SQ(*x);
    /* Note that although the argument is passed by value, the compiler */
    /* copies the argument to a temporary variable, and the pointer to */
    /* the temporary variable is passed to the called program SQ.      */
    printf("The SQUARE of %d is %d\n", *x, result);
}
```

Figure 143. Sample Source Statements for Program T1520TAP

3. To create the program T1520TAP using the source shown below, enter:

```
CRTBNDC PGM(MYLIB/T1520TAP) SRCFILE(QCPPLE/QACSRC)
```



```

/* This program illustrates how to write to a tape file.          */
#include <stdio.h>
#include <string.h>
#include <recio.h>
#include <stdlib.h>
#define RECLEN 80

int main(void)
{
    _RFILE *tape;
    _RFILE *fp;
    char buf [92];
    int i;

/* Open the source physical file containing the C source.        */

    if ( ( fp = _Ropen ( "*/LIBL/QCSRC(CSOURCE)", "rr blkrcd=y" ) ) == NULL )
    {
        printf ( "could not open C source file\n" );
        exit ( 1 );
    }
/* Open the tape file to receive the C source statements        */

    if ( ( tape = _Ropen ( "*/LIBL/T1520TPF", "wr lrecl=92 blkrcd=y" ) ) == NULL )
    {
        printf ( "could not open tape file\n" );
        exit ( 2 );
    }

/* Read the C source statements, find their sizes                */
/* and add them to the tape file.                                */

    while ( ( _Rreadn ( fp, buf, sizeof(buf), __DFT ) -> num_bytes != EOF )
    {
        memmove ( buf, buf+12, RECLEN );
        _Rwrite ( tape, buf, RECLEN );
    }

    _Rclose ( fp );
    _Rclose ( tape );
    return(0);
}

```

Figure 144. T1520TAP – ILE C Source to Write to a Tape File

This program opens the source physical file T1520TPF. The `_Ropen()` function file QCSRC contains the member CSOURCE with the source statements. The `_Ropen()` function opens the tape file T1520TPF to receive the C source statements. The `_Rreadn()` function reads the C source statements, finds their sizes, and adds them to the tape file T1520TPF.

4. To run the program T1520TAP, enter:

```
CALL PGM(MYLIB/T1520TAP)
```

After you run the program, the tape file contains the source statements from CSOURCE.

Writing to a Diskette File

You can write records to a diskette file.

A diskette file is a device file that is used for a diskette unit. The object type is `*FILE`.

I/O Considerations for Diskette Files

A diskette unit can only be accessed with a program described file. An ILE C/C++ program can only process a diskette file sequentially.

The concept of clearing a file or opening a file using append mode does not apply to diskette files.

The diskette file label name is required when the file is opened. You specify this label name using the Override Diskette File (OVRDKTF) command.

If the diskette file is opened for input and:

- if the `lrecl` parameter is not specified or is specified as zero, the record length in the data file label on the name on the diskette is used to determine the length of the records to read.
- if the `lrecl` parameter is greater than the length of the records on the diskette file, the records that are read are padded with blanks.
- if the `lrecl` parameter is less than the length of the records on the diskette file, the records that are read are truncated.
- if the file type in the diskette file is a source file, a date and sequence number is appended at the beginning of each record. You must remove these when writing the record and add 12 bytes to the `lrecl` parameter on the open statement.

Note: Output may not always result in an I/O operation to a diskette file. The I/O buffer must contain enough data to fill an entire track on a diskette.

When opening a diskette file for output, any files existing on the diskette are deleted if the data file expiration date is less than or equal to the system date.

Opening Diskette Files as Binary Stream Files

To open an IBM i diskette file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb`
- `wb`

Note: The only way to create a diskette file is to use the CRTDKTF command. If you use `fopen()` and the diskette file does not exist, a physical database file is created.

The valid keyword parameters are:

- `type`
- `lrecl`
- `blksize`

I/O Considerations for Binary Stream Diskette Files

Blocking Binary Stream Diskette Files

If your program processes diskette files, performance can be improved if I/O operations are blocked. If you do not specify a value for the `blksize` parameter or if you specify `blksize=0` on `fopen()`, the system calculates a number of records to be transferred as a block to your program.

Binary Stream Functions for Diskette Files

Use the following binary stream functions to process diskette files:

- `fclose()`
- `fopen()`
- `fread()`
- `freopen()`
- `fwrite()`

Opening Diskette Files as Record Files

To open an IBM i diskette file as a record file, use the `_Ropen()` function with one of the following modes:

- `RI`
- `WI`

The valid keyword parameters are:

- blkrcd
- lrecl
- secure
- riofb

I/O Considerations for Record Diskette Files

The `_Rfeod()` function is valid for diskette record files opened for input and output operations. It signals the end-of-file. For output operations, it does not write any data.

Read and Write Record Diskette Files

If you read from a diskette file, the next sequential record in the diskette file is processed. Use the `_Rreadn()` function for reading diskette files and the `_Rwrite()` function for writing to diskette files.

Blocking Record Diskette Files

If your program processes diskette files, performance can be improved if records are blocked. If you specify `blkrcd=Y` on `_Ropen()`, the system calculates a number of records to be transferred as a block to your program.

Record Functions for Diskette Files

Use the following record functions to process diskette files:

- `_Rclose()`
- `_Rfeod()`
- `_Riofbk()`
- `_Ropen()`
- `_Ropfbk()`
- `_Rreadn()`
- `_Rupfbk()`
- `_Rwrite()`

Example:

The following example shows how to write to a diskette file.

1. To create the diskette file T1520DKF, enter:

```
CRTDKTF FILE(MYLIB/T1520DKF) DEV(DKT02) LABEL(FILE1)
EXCHTYPE(*I) SPOOL(*NO)
```

2. Enter:

```
CRTPF FILE(MYLIB/T1520DDI) SRCFILE(QCPPLE/QADSSRC)
SHARE(*YES)
```

To create the physical file T1520DDI using the following DDS source:

```
      A          R CUST
      A          NAME          20A
      A          AGE           3B
      A          DENTAL        6B
```

3. Type the following records into the database file T1520DDI:

Dave Bolt	35 350
Mary Smith	54 444
Mike Tomas	25 545
Alex Michaels	32 512

4. To select only records that have a value greater than 400 in the DENTAL field, enter:

```
OPNQRYF FILE((MYLIB/T1520DDI)) QRYSLT('DENTAL *GT 400') OPNSCOPE(*JOB)
```

5. To create the program T1520DSK using the source shown below, enter:

```
CRTBND C PGM(MYLIB/T1520DSK) SRCFILE(QCPPL/QACSRC)
```

```
/* This program illustrates how to write to a diskette file.          */
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

#define BUF_SIZE 30

int main(void)
{
    _RFILE *dktf;
    _RFILE *dbf;
    char buf [BUF_SIZE];

    /* Open the diskette file                                          */

    if ( ( dktf = _Ropen ( "*LIBL/T1520DKF", "wr blkrcd=y lrecl=100" ) ) == NULL )
    {
        printf ( "DISKETTE file did not open \n" );
        exit ( 1 );
    }

    /* Open the database file.                                         */

    if ( ( dbf = _Ropen ( "*LIBL/T1520DDI", "rr" ) ) == NULL )
    {
        printf ( "DATABASE file did not open\n" );
        exit ( 2 );
    }

    /* Copy all the database records meeting the OPNQRYF selection    */
    /* criteria to the diskette file.                                  */

    while ( ( _Rreadn ( dbf, buf, BUF_SIZE, __DFT ) ) -> num_bytes != EOF )
    {
        _Rwrite ( dktf, buf, BUF_SIZE );
    }
    _Rclose ( dktf );
    _Rclose ( dbf );
}
```

Figure 145. T1520DSK – ILE C Source to Write Records to a Diskette File

The `_Ropen()` function opens the diskette file T1520DKF and the database file T1520DDI. The `_Rreadn()` function reads all database records. The `_Rwrite()` function copies all database records that have a value > 400 in the DENTAL field to the diskette file T1520DKF.

6. To run the program T1520DSK, enter:

```
CALL PGM(MYLIB/T1520DSK)
```

The output to the diskette file is as follows:

Mary Smith	444
Mike Tomas	545
Alex Michaels	512

After you run the program, the diskette file contains only the records that satisfied the selection criteria.

Using Save Files

A *save file* is a file allocated in auxiliary storage that can be used to store saved data on disk (without requiring diskettes or tapes), or to receive objects sent through the network. The object type is *FILE. The *Recovering your system* manual contains information on save files.

I/O Considerations for Save Files

An ILE C/C++ program can only process save files sequentially. All records that are read or are written must be 528 characters in length. Any records that are written to another save file cannot be changed by the ILE C/C++ program.

Opening Save Files as Binary Stream Files

To open an IBM i save file as a binary stream file for record-at-a-time processing, use the `fopen()` function with one of the following modes:

- `rb`
- `wb`
- `ab`

Note: The only way to create a save file is to use the CRTSAVF command. If you use the `fopen()` function with a mode of `wb` or `ab` and the save file does not exist, a physical database file is created.

The valid keyword parameters are:

- `lrecl`
- `type`

I/O Considerations for Binary Stream Save Files

There are no special considerations for binary stream save files.

Binary Stream Functions for Save Files

Use the following binary stream functions to process save files:

- `fclose()`
- `fopen()`
- `fread()`
- `freopen()`
- `fwrite()`

Opening Save Files as Record Files

To open an IBM i save file as a record file, use the `_Ropen()` function with one of the following modes:

- `rr`
- `wr`
- `ar`

The valid keyword parameters are:

- `lrecl`
- `riofb`
- `secure`

I/O Considerations for Record Save Files

If a save file is opened for input, the `_Rfeod()` function returns an end-of-file to your program. If a save file is opened for output, the `_Rfeod()` function ensures that any data that is written to the file is forced to auxiliary storage. If you want to continue reading from or writing to the save file after calling this function, you must close the file and open it again.

Record Functions for Save Files

The following record functions can be used to process save files:

- `_Rclose()`
- `_Rfeod()`
- `_Riofbk()`
- `_Ropen()`
- `_Ropnfbk()`
- `_Rreadn()`
- `_Rupfbk()`
- `_Rwrite()`

Working with IBM i Features

This topic describes how to:

- [Understand the ILE control boundary and exception handling](#)
- [Using pointers in a program](#)
- [Using ILE C/C++ calling conventions](#)
- [Porting programs from ILE C to ILE C++](#)
- [Calling programs and procedures in a multi-language environment](#)
- [Using packed decimal data types in C programs](#)
- [Using packed decimal data types in C++ programs](#)
- [Using templates in C++ programs](#)
- [Using teraspace storage](#)
- [Casting with RunTime Type Information](#)

Handling Exceptions in a Program

This topic describes:

- [ILE language-specific error handling](#)
- [Exception messages](#)
- [How the system processes exceptions](#)
- [Detecting stream file and record file errors](#)
- [Using ILE exception handlers:](#)
 - [Direct monitor handlers](#)
 - [ILE condition handlers](#)
- [Using the C/C++ signal handler](#)
- [Using both signal handlers and ILE exception handlers](#)
- [Handling nested exceptions](#)
- [Using cancel handlers](#)
- [Example of ILE C source code that uses a variety of exception handling methods](#)

ILE Language-Specific Error Handling

Figure 146 on page 246 illustrates the complexity of ILE C/C++ language-specific error handling in comparison to OPM language-specific error handling.

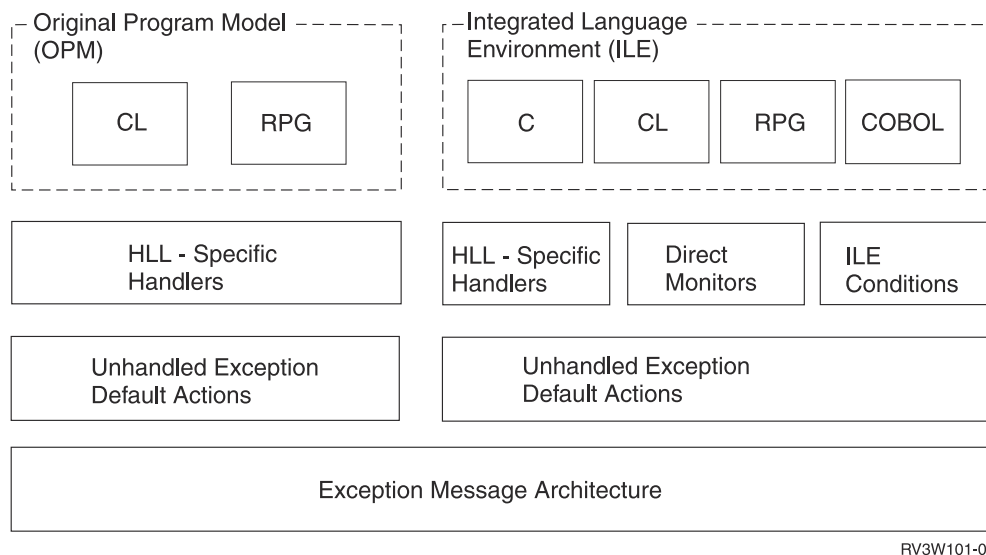


Figure 146. Error Handling for OPM and ILE

For OPM programs, language-specific error handling provides one or more handling routines for each call stack entry. The system calls the appropriate routine when an exception is sent to an OPM program.

For ILE C/C++ programs, language-specific error handling provides the same capabilities, plus additional types of exception handlers. These additional types of handlers allow you to

- Change the exception message to indicate that the exception is handled
- Bypass the language-specific error handling

The additional types of handlers for ILE C/C++ are:

- Direct monitor handler
- ILE condition handler

Exception Messages

The following are the only types of messages that are considered to be *exception messages*:

(*ESCAPE)

Indicates an error that caused a program to end abnormally. If the message type is *ESCAPE then a function check is sent to the call stack entry that is pointed to by the resume cursor.

(*STATUS)

Describes the status of work that the program is in the process of doing. If the message type is *STATUS, the program resumes without logging the exception.

(*NOTIFY)

Describes a condition that requires corrective action or reply from calling program. If the message type is *NOTIFY, the default reply is sent.

Function Check

Describes an ending condition that the program has not expected. If the message is a function check, the call stack is cancelled to the control boundary and CEE9901 is sent to the caller of the control boundary

For more information about exception messages and how they are sent, see *ILE Concepts*.

How the System Processes Exceptions

The exception message architecture of the IBM i is used to implement both exception handling and condition handling. There are cases in which exception handling and condition handling interact. For example, an ILE condition handler registered with the Register a User-Written Condition Handler

(CEEHDLR) bindable API is used to handle an exception message sent with the Send Program Message (QMHSNDPM) API.

The term exception handler is used to mean either an IBM i exception handler or an ILE condition handler.

Note: See *ILE Concepts* for more information about:

- Exception and condition handling
- Exception recovery

How the Call Message Queue Handles ILE Procedures and Functions

A message queue exists for every call stack entry within each IBM i job. This is known as a call message queue. As soon as a new entry appears in the call stack, the system creates a new call message queue. In ILE C/C++, the name of the procedure identifies the call message queue. If the procedure names are not unique, you can specify the module name, program name, or service program as well. If you determine that your handler does not recognize an exception message, the exception message can be percolated to the next handler.

How Control Boundaries Affect Exception Handling in ILE

Whenever an unhandled function check occurs or an HLL end verb is used, control is transferred to the caller of the call stack entry that represents a boundary for your application. This call stack entry is known as a control boundary.

A control boundary can be either of the following:

- Any ILE call stack entry for which the immediately preceding call stack entry is in a different non-default activation group
- Any ILE call stack entry for which the immediately preceding call stack entry is an OPM program.

For detailed information about control boundaries, see *ILE Concepts*.

Unmonitored Exceptions and Unhandled Exceptions

When an unmonitored exception occurs, the program that is running issues a function check and sends a message to the job log. If you are in debug mode and the modules of the program were created with debug data, the ILE source debugger displays the appropriate module and, if necessary, the program is added to debug mode.

When an unhandled function check occurs, ILE transfers control to the caller of the call stack entry that represents a boundary for the application. This call stack entry is known as a control boundary.

For more information about control boundaries, see:

- [“How Control Boundaries Affect Exception Handling in ILE” on page 247](#)
- *ILE Concepts*

For detailed information about the default treatment for unhandled exceptions, see *ILE Concepts*.

Example of ILE C Source Code with an Unhandled Exception

Figure 147 on page 247 shows ILE C source code with an unhandled *ESCAPE exception.

```
void fred(void)
{
    char *p = NULL;
    *p = 'x';      /* *ESCAPE exception */
}
int main(void)
{
    fred();
}
```

Figure 147. ILE C Source Code with Unhandled Exceptions

In [Figure 147 on page 247](#):

1. An exception is sent to the `fred()` function.
2. The `main()` function is the control boundary.

Note: For illustration, many of the examples refer to the `main()` function as a control boundary. If a program is running in a *NEW activation group, the program entry procedure (PEP) is the control boundary for the program.

3. The `fred()` function has no exception handlers therefore the exception is percolated to `main()`.
4. Because the `main()` function has no exception handlers and `main()` is a control boundary, the system takes the default action.
5. Because the exception is of type *ESCAPE, a function check is sent to the `fred()` function.

Note: For information about actions taken for other types of exceptions, see [“Exception Messages” on page 246](#).

6. The function check percolates to function `main()`, and again the default is taken.
7. Because the exception is of type function check, the call stack entries of the `main()` and `fred()` functions are cancelled and the CEE9901 exception is sent to the caller of function `main()`.

Nested Exceptions

A *nested exception* is an exception that occurs while another exception is being handled.

When this happens:

- Processing of the first exception is temporarily suspended.
- The system saves all of the associated information such as the locations of the handle cursor and resume cursor.
- Exception handling begins again with the most recently generated exception.
- New locations for the handle cursor and resume cursor are set by the system.
- Once the new exception has been properly handled, handling activities for the original exception normally resume.

When a nested exception occurs, both of the following are still on the call stack:

- The call stack entry associated with the original exception
- The call stack entry associated with the original exception handler

To reduce the possibility of exception handling loops, the system stops the percolation of a nested exception at the original exception handler call stack entry. Then the system promotes the nested exception to a function check message and percolates the function check message to the same call stack entry.

If the nested exception or the function check message is not handled, the application comes to an abnormal end. In this case, message CEE9901 is sent to the caller of the control boundary.

If you move the resume cursor while processing the nested exception, you can implicitly modify the original exception. To cause this to occur, do the following:

1. Move the resume cursor to a call stack entry that was promoted earlier than the call stack entry that incurred the original exception.
2. Resume processing by returning from your handler.

Detecting Stream File and Record File Errors

To detect stream file errors, check:

- The return value of a function
- The `errno` value

- The major/minor return code (for system exceptions)

To detect record file errors, check the values in the `_RIOFB_T` structure.

Note: For a list of exception messages generated by the ILE C/C++ record I/O functions, see the Record Input and Output Error Macro to Exception Mapping table in the *ILE C/C++ Runtime Library Functions*.

To detect stream file or record file errors, check the Global Variable `_EXCP_MSGID`.

Checking the Return Value of a Function

Many C and C++ runtime library functions have a return value associated with them for error-checking purposes. For example:

- The `_Rfeov()` function returns 1 if the file has moved from one volume to the next.
- The `fopen()` function returns NULL if a file is not opened successfully.

For information about the ILE C/C++ function return values, see *ILE C/C++ Runtime Library Functions*.

To verify that each runtime library function has completed successfully, a program should check the function return values.

Example:

The following figure shows how to check the return value of the `fopen()` function.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    FILE *fp;
    if (( fp = fopen ( "MYLIB/QCSRC(TEST)", "ab" )) == NULL )
    {
        printf ("Cannot open file QCSRC(TEST)\n");
        exit (99);
    }
}
```

Figure 148. ILE C Source to Check for the Return Value of `fopen()`

Checking the Errno Value

The `<errno.h>` header file contains declarations for defined error conditions. Many C functions set `errno` to specific values, depending on the type of error. These values are also defined in the `<errno.h>` header file. The implementation of `errno` contains a function call.

The following `errno` macros indicate an IBM i system exception:

- `EIOERROR`: a nonrecoverable I/O error has occurred.
- `EIORECERR`: a recoverable I/O error has occurred.

For a list of `errno` macro values, see *ILE C/C++ Runtime Library Functions*.

Initializing Errno

Your program should always initialize `errno` to 0 (zero) before calling a function because `errno` is not reset by any library functions. Check for the value of `errno` immediately after calling the function that you want to check. You should also initialize `errno` to zero after an error has occurred.

Viewing and Printing the Errno Value

Your program can use the `strerror()` and `perror()` functions to print the value of `errno`. The `strerror()` function returns a pointer to an error message string that is associated with `errno`. The `perror()` function prints a message to `stderr`. The `perror()` and `strerror()` functions should be used immediately after a function is called because subsequent calls might alter the `errno` value.

Example: Checking the errno Value for the fopen() Function

The following figure shows how to check the errno value for the fopen() function.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
    FILE *fp;
    errno = 0;
    fp = fopen("Nofile", "r");
    if ( errno != 0 )
    {
        perror("Error occurred while opening file.\n");
        exit(1);
    }
}
```

Figure 149. ILE C Source to Check the errno Value for fopen()

Checking the Major/Minor Return Code

If your program processes display, ICF, or printer files as stream files, you can check the external variable `_C_Maj_Min_rc`, which is defined in `<stdio.h>`. The following figure shows the definition of this structure.

```
typedef struct _Major_Minor_rc
{
    char major_rc[2];
    char minor_rc[2];
} _Major_Minor_rc;
extern _Major_Minor_rc _C_Maj_Min_rc;
```

Figure 150. `_C_Maj_Min_rc` Type Definition

Checking the System Exceptions for Record Files

To detect record file errors, you can check some values in the `_RIOFB_T` structure, which is defined in `<recio.h>`. Both the `num_bytes` field and the `sysparm` field contain information regarding record file I/O errors.

The following figure shows the type definition of the `_RIOFB_T` structure:

```
typedef struct {
    unsigned char    *key;
    _Sys_Struct_T    *sysparm; 1, 3
    unsigned long    rrrn;
    long             num_bytes; 1
    short            blk_count;
    char             blk_filled_by;
    int              dup_key    : 1;
    int              icf_locate : 1;
    int              reserved1  : 6;
    char             reserved2[20];
} _RIOFB_T;
```

Figure 151. `_RIOFB_T` Type Definition

Note:

1. If your program processes display, ICF, or printer files as record files, you can check the values in the `num_bytes` field in the `_RIOFB_T` structure and the major/minor return code fields in the `sysparm` area of the `_RIOFB_T` structure. The `num_bytes` field indicates if the I/O operation was successful.
2. If your program processes database files as stream files, you can check the values in some fields of the `_RIOFB_T` structure.
3. The `sysparm` field points to a structure that contains the major/minor return code for display, ICF, or printer files. The definition of `_Sys_Struct_T` structure is shown below:

```
typedef struct {          /* System specific information */
    void *sysparm_ext;
    _Maj_Min_rc_T _Maj_Min;
    char reserved1[12];
} _Sys_Struct_T;
```

Figure 152. *_Sys_Struct_T* Type Definition

The following figure shows the definition of the `_Maj_Min_rc_T` structure:

```
typedef struct {
    char major_rc[2];
    char minor_rc[2];
} _Maj_Min_rc_T;
```

Figure 153. *_Maj_Min_rc_T* Type Definition

Checking the Global Variable `_EXCP_MSGID`

The global variable `_EXCP_MSGID` is set whenever a stream or record I/O function gets an exception. The global variable `_EXCP_MSGID`, declared in the `<stddef.h>` header file, contains the exception message ID. See the Record Input and Output Error Macro to Exception Mapping table in the Runtime Considerations section of the *ILE C/C++ Runtime Library Functions* for information about the `_EXCP_MSGID` setting after an IBM i exception.

Using ILE Exception Handlers

Types of Exception Handlers

The types of exception handlers that you can use in ILE are:

- Direct monitor handlers, which are enabled with the `#pragma exception_handler` directive
- ILE condition handlers that allow you to register a condition handler at runtime by using the bindable API `CEEHDLR`
- the C/C++ `signal()` function

Using ILE Direct Monitor Handlers

Direct monitor handlers monitor for exceptions that are based on exception classes and message identifiers. They let you directly register an exception monitor for a limited number of C/C++ source statements. Direct monitors are usually the fastest handlers.

Using the *pragma* Directives

In ILE C/C++, the `#pragma exception_handler` and `#pragma disable_handler` directives enable direct monitor handlers.

The `#pragma exception_handler` enables a direct monitor handler from `#pragma exception_handler` to `#pragma disable_handler` without considering program logic in between.

When using these directives, you must include the `<except.h>` header file in your source code .

Note: The `#pragma exception_handler` directive can monitor only those exception message types listed in the [exception messages list](#).

A direct monitor handler may be either of the following:

- Code following a label defined within the function containing the `#pragma exception_handler`
- A function with a return type of `void`. The function should take one parameter; a pointer to a structure of type `_INTRPT_HndlR_Parms_T`. The `_INTRPT_HndlR_Parms_T` type is defined in the `<except.h>` file.

Using Communications Area Variables

A communications area variable may be specified on the `#pragma exception_handler`. The use of this variable depends on whether the handler is a label or a function. If the handler is a label then the communications area is used as storage for the standard exception handler parameter block of type `_INTRPT_Hndlr_Parms_T` (defined in `<except.h>`).

The definition of the structure `_INTRPT_Hndlr_Parms_T` is:

```
typedef _Packed struct {
    unsigned int    Block_Size;          /* Size of the parameter block */
    _INVFLAGS_T    Tgt_Flags;           /* Target invocation flags */
    char           reserved[8];         /* reserved */
    _INVPTR        Target;              /* Current target invocation */
    _INVPTR        Source;              /* Source invocation */
    _SPCPTR        Com_Area;            /* Communications area */
    char           Compare_Data[32];    /* Compare Data */
    char           Msg_Id[7];           /* Message ID */
    char           reserved1;           /* 1 byte pad */
    _INTRPT_Mask_T Mask;                /* Interrupt class mask */
    unsigned int    Msg_Ref_Key;        /* Message reference key */
    unsigned short  Exception_Id;       /* Exception ID */
    unsigned short  Compare_Data_Len;   /* Length of Compare Data */
    char           Signal_Class;        /* Internal signal class */
    char           Priority;             /* Handler priority */
    short          Severity;            /* Message severity */
    char           reserved3[4];
    int            Msg_Data_Len;        /* Length of available message data */
    char           Mch_Dep_Data[10];    /* Machine dependent data */
    char           Tgt_Inv_Type;        /*Invocation type (in MIMCHOBS.H)*/
    _SUSPENDPTR    Tgt_Suspend;        /* Suspend pointer of target */
    char           Ex_Data[48];         /* First 48 bytes of exception data */
} _INTRPT_Hndlr_Parms_T;
```

Figure 154. Definition of Structure `_INTRPT_Hndlr_Parms_T`

The system fills in the structure prior to giving control to the label. If the storage that is required for the exception handler parameter block exceeds the storage that is defined by `com_area`, the remaining bytes are truncated. If the handler is a function, the system passes a pointer to a structure of type `_INTRPT_Hndlr_Parms_T` to the function. A pointer to the communications area is available inside the structure.

Scoping Direct Monitor Handles

The direct monitor handlers are scoped at compile time to the code between the `#pragma exception_handler` directive and the `#pragma disable_handler` directive. For example, the `#pragma exception_handler` directive is scoped to a block of code independent of the program logic.

The following figure provides an example.

```
volatile int ca=0;
if (ca != 0){
    #pragma exception_handler(my_handler, ca,0,_C2_MH_ESCAPE)
}
else {
    raise(SIGINT);/* Signal will be caught by my_handler */
}
#pragma disable_handler
```

Figure 155. ILE C Source to Scope Direct Monitor Handlers

In Figure 155 on page 252:

- The `#pragma exception_handler` directive is enabled around the call to the `raise()` function.
- The conditional expression `if (ca != 0)` has no effect on enabling the direct monitor handler. The logic path for the conditional expression `if (ca != 0)` is never taken. Instead, `my_handler` is enabled.

Using Exception Classes

Note: The `#pragma exception_handler` directive can monitor only those exception message types listed in the [exception messages list](#).

Exception classes indicate the type of exception (for example, *ESCAPE, *NOTIFY, *STATUS, function check) and, for machine exceptions, the low level type (for example, pointer-not-valid or divide-by-zero).

The handler gains control if the exception falls into one or more of the exception classes that are specified on the `#pragma exception_handler` directive.

Note:

The Runtime Considerations section of the *ILE C/C++ Runtime Library Functions* contains a table of exception classes.

The following figure provides an example.

```
#include <except.h> 1
/* Just monitor for pointer not valid exceptions */
#pragma exception_handler(eh, 0, _C1_POINTER_NOT_VALID, 0)
/* Monitor for all *ESCAPE messages */
#pragma exception_handler(eh, 0, 0, _C2_MH_ESCAPE) 2
/* Although the following is valid, there is no need to specify */
/* _C1_POINTER_NOT_VALID because it is covered by _C2_MH_ESCAPE */
#pragma exception_handler(eh, 0, _C1_POINTER_NOT_VALID, _C2_MH_ESCAPE) 2
/* To monitor for only specific messages, use the extended form of */
/* pragma exception_handler. */
/* The following #pragma will only monitor for MCH3601 *ESCAPE msg. */
#pragma exception_handler (eh, 0, 0, _C2_MH_ESCAPE, _CTLA_HANDLE, "MCH3601") 2
```

Figure 156. ILE C Source to Use Exception Classes

Note:

1. Macros for the IBM i machine exception classes are defined in the ILE C/C++ include file `<except.h>`.
2. To monitor for machine exceptions, you can specify the machine exception class or you can specify all *ESCAPE exceptions. In [Figure 156](#) on page 253, all machine exceptions are mapped to the *ESCAPE type exception. If the message type is *ESCAPE, a function check is sent to the call stack entry that is pointed to by the resume cursor.
3. You can monitor for the exception class values for class1 and class2. The value of class2 must be only one of the following exception classes:
 - _C2_MH_ESCAPE
 - _C2_MH_STATUS
 - _C2_MH_NOTIFY
 - _C2_MH_FUNCTION_CHECK

Specifying Control Actions

The `#pragma exception_handler` directive allows you to specify a control action that is to be taken during exception processing.

The five control actions that can be specified, as defined in the `<except.h>` header file, are:

_CTLA_INVOKE

This control action will cause the function that is named on the directive to be called and will not handle the exception. The exception will remain active and must be handled by using QMHCHGEM or one of the ILE condition-handling APIs.

_CTLA_HANDLE

This control action will cause the function or label that is named on the directive to get control and it will handle and log the exception implicitly. The exception will no longer be active when the handler gets control.

`_CTLA_HANDLE_NO_MSG`

This control action is the same as `_CTLA_HANDLE` except that the exception is NOT logged. The message reference key in the parameter block that is passed to the handler will be zero.

`_CTLA_IGNORE`

This control action will handle and log the exception implicitly and will not pass control to the handler function named on the directive; that is, the function named will be ignored. The exception will no longer be active, and processing will resume at the instruction immediately following the instruction that caused the exception.

`_CTLA_IGNORE_NO_MSG`

This control action is the same as `_CTLA_IGNORE` except that *NOTIFY messages will be logged.

Example:

The following figure shows how the control action parameter can be specified on the `#pragma exception_handler` directive.

```
#include <except.h>
#include <stdio.h>
void myhandler(void) {
    printf("In handler - something's wrong!\n");
    return;
}
int main(void) {
    int *ip;
    volatile int com_area;
    #pragma exception_handler(myhandler, com_area, 0, _C2_ALL, \
                             _CTLA_IGNORE) 1
    *ip = 5;
    printf("Passed the exception.\n"); 2
}
```

Figure 157. ILE C Source to Handle Exceptions

Note:

1. The control action `_CTLA_IGNORE` will cause the exception to be handled without calling the handler function.
2. The output of this code is the message `Passed the exception`, followed by the exception message. This code will generate the exception message `MCH3601 (Pointer not set)`.

Specifying Message Identifiers

The `#pragma exception_handler` directive can specify one or more specific or generic message identifiers on the directive. When one or more identifiers are specified on the directive, the direct monitor handler will take effect only when an exception occurs whose identifier matches one of the identifiers on the directive.

To specify message identifiers on the directive, you must specify a control action to be taken. The class of the exception must be in one of the classes specified on the directive.

Example:

The following example shows a `#pragma exception_handler` directive that enables a monitor for a single specific message, `MCH3601`:

```
#pragma exception_handler (myhandler, com_area, 0, _C2_ALL, \
                           _CTLA_HANDLE, "MCH3601")
```

Example:

The following is an example of a `#pragma exception_handler` directive that enables a monitor for several floating-point exceptions:

```
#pragma exception_handler (myhandler, com_area, _C1_ALL, _C2_ALL, \
                           _CTLA_IGNORE, "MCH1206 MCH1207 MCH1209 MCH1213")
```


Note: The ability to specify generic message identifiers can be used to simplify the directive.

Example:

In the following example, a monitor is enabled for any exception whose identifier begins with MCH12:

```
#pragma exception_handler (myhandler, com_area, _C1_ALL, _C2_ALL, \  
    _CTLA_IGNORE, "MCH1200")
```

Example of Source Code that Uses a Direct Monitor Handler

The following figure shows the source for a program MYPGM:

```
/* MYPGM *PGM */  
#include <except.h>  
#include <stdio.h>  
void my_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms);  
void main_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms);  
void fred(void)  
{  
    char *p = NULL;  
    #pragma exception_handler(my_handler, 0,0,_C2_MH_ESCAPE)  
    *p = 'x'; /* exception */  
    #pragma disable_handler  
}  
int main(void)  
{  
    #pragma exception_handler(main_handler, 0,0,_C2_MH_ESCAPE)  
    fred();  
}
```

Figure 158. T1520XH1 – ILE C Source to Use Direct Monitor Handlers – main()

In Figure 158 on page 255:

- The procedure main() registers the direct monitor handler main_handler
- The procedure main() calls fred(), which registers the direct monitor handler my_handler.
- The fred() function gets an exception which causes my_handler to get control, followed by main_handler. The main() function is a control boundary.
- The exception is considered unhandled so a function check is sent to fred().
- The handlers my_handler and main_handler handle *ESCAPE messages only, so neither is called again.
- The function check goes unhandled at main() so the program ends abnormally and CEE9901 is sent to the caller of main().

Example of Source that Illustrates How to Use Direct Monitor Handlers

```
/* This program illustrates how to use direct monitor handlers. */  
#include <stdio.h>  
#include <signal.h>  
#include <recio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <except.h> /* Include except.h even though it is included */  
/* in the signal.h header file. */  
#define FILE_NAME "QTEMP/MY_FILE"  
#define RCD_LEN 80  
#define NUM_RCD 5  
#pragma datamodel(p128)  
typedef struct error_code{  
    int byte_provided;  
    int byte_available;  
    char exception_id[7];  
    char reserve;  
    char exception_data[1];  
}error_code_t;  
static int handle_flag;  
#pragma linkage(QMHCHGEM, OS)  
void QMHCHGEM(_INVPTR *, int, unsigned int, char *,  
    char *, int, error_code_t *);
```

```

/* The signal handler. */
#pragma datamodel(pop)
static void sig_handler(int sig)
{
    printf("In signal handler\n");
    printf("Exception message ID is %7.7s\n", _EXCP_MSGID);
}
/* The direct monitor handler. */
static void exp_handler(_INTRPT_Hndlr_Parms_T * __ptr128 exp_info)
{
    error_code_t error_code;
    printf("In direct monitor handler\n");
    printf("Exception message ID is %3.3s%04x\n",
        exp_info->Compare_Data,
        (unsigned) exp_info->Exception_Id);
/* Call QMHCHGEM API to handle the exception. */
if ( handle_flag )
{
    error_code.byte_provided = 8;
    QMHCHGEM(&(exp_info->Target), 0, exp_info->Msg_Ref_Key,
        "HANDLE ", "", 0, &error_code);
}
}
/* The function to read a file. */
static void read_file(_RFILE *fp)
{
    int i = 1;
    while ( _Rreadn(fp, NULL, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
}
int main(void)
{
    _RFILE *fp;
    int i;
    volatile int com;
    char buf[RCD_LEN];
    char cmd[100];
/* Create a file. */
sprintf(cmd, "CRTPF FILE(%s) RCDLEN(%d)", FILE_NAME, RCD_LEN);
system(cmd);
/* Open the file for write. */
if ( (fp = _Ropen(FILE_NAME, "wr")) == NULL )
{
    printf("Open for write fails\n");
    exit(1);
}
/* Write some data into the file. */
memset(buf, '1', RCD_LEN);
for ( i = 0; i < NUM_RCD; i++ )
{
    _Rwrite(fp, buf, RCD_LEN);
}
_Rclose(fp);
/* Open the file for the first read. */
if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
{
    printf("Open for the first read fails\n");
    exit(2);
}
/* Read until end-of-file. */
/* Since no signal handler or direct monitor handler is set up,
/* the EOF exception is ignored. The default value for SIGIO is
/* SIG_IGN.
i = 1;
printf("The first read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
{
    printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The first read finishes\n");

/* Set up a direct monitor handler and a signal handler.
/* Tell the direct monitor handler to handle the exception.
/* The direct monitor handler (exp_handler) calls the message
/* handler API QMHCHGEM with the parameter *HANDLE. This marks the
/* exception as handled.
/* Use exception classes to handle machine exceptions.
handle_flag = 1;
#pragma exception_handler(exp_handler, com, 0, \

```

```

        _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
signal(SIGIO, sig_handler);
/* Open the file for the second read. */
if ( (fp = _Ropen(FILE_NAME, "rr") == NULL )
{
    printf("Open for the second read fails\n");
    exit(3);
}
/* Read until end of file. */
/* When the EOF exception is generated, the direct monitor handler */
/* is called first. Since it marks the exception as handled, */
/* the signal handler is not called. */
i = 1;
printf("The second read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
{
    printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The second read finishes\n");
/* Disable the direct monitor handler. */
#pragma disable_handler
/* Set up a direct monitor handler and a signal handler. */
/* Set the global variable handle_flag to zero so that the */
/* direct monitor will not handle the exception. */
handle_flag = 0;
#pragma exception_handler(exp_handler, com, 0, \
        _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
signal(SIGALL, sig_handler);
/* Open the file for the third read. */
if ( (fp = _Ropen(FILE_NAME, "rr") == NULL )
{
    printf("Open for the third read fails\n");
    exit(4);
}
/* Read until end-of-file. */
/* When the EOF exception is generated, the direct monitor handler */
/* is called first. Since the exception is not marked as */
/* handled, the signal handler is then called. */
i = 1;
printf("The third read starts\n");
while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
{
    printf("Read record %d\n", i++);
}
_Rclose(fp);
printf("The third read finishes\n");

/* Disable the direct monitor handler. */
#pragma disable_handler
/* Set up a direct monitor handler and a signal handler. */
#pragma exception_handler(exp_handler, com, 0, \
        _C2_MH_ESCAPE | _C2_MH_NOTIFY | _C2_MH_STATUS)
signal(SIGIO, sig_handler);
/* Open the file for the fourth read. */
if ( (fp = _Ropen(FILE_NAME, "rr") == NULL )
{
    printf("Open for the fourth read fails\n");
    exit(5);
}
/* Read until end-of-file. */
/* The EOF exception is generated in function read_file. Since */
/* there is no direct monitor handler for the read_file function, */
/* the signal handler is called. */
/* The direct monitor handler in main() is not called because the */
/* exception was mapped to SIGIO and the signal handler gets called */
/* at function read_file. */
printf("The fourth read starts\n");
read_file(fp);
_Rclose(fp);
printf("The fourth read finishes\n");
/* Disable the direct monitor handler. */
#pragma disable_handler
}

```

Example of a Service Program that Provides Direct Monitor Handle

The following figure shows the source for the service program HANDLERS:

```

#include <signal.h>
#include <stdio.h>
/* HANDLERS *SRVPGM (created with activation group *CALLER) */
void my_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    return;
}
void main_handler(_INTRPT_Hndlr_Parms_T * __ptr128 parms)
{
    printf("In main_handler\n");
}

```

Figure 159. T1520XH2 – ILE C Source to Use Direct Monitor Handlers – Service Program

Example that Uses Labels Instead of Functions as Handlers

The following example illustrates direct monitor handlers using labels instead of functions as the handlers:

```

#include <except.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
void sig_hndlr(int);
void sig_hndlr(int sig){
    printf("Signal handler should not have been called\n");
}
int main(void)
{
    int a=0;
    char *p=NULL;
    volatile _INTRPT_Hndlr_Parms_T ca;
    /* Set up signal handler for SIGFPE. The signal handler function */
    /* should never be invoked, since the exception will be handled */
    /* by the direct monitor handlers. */
    if( signal(SIGFPE,sig_hndlr) == SIG_ERR )
    {
        printf("Could not set up signal handler for SIGFPE\n");
    }
    /* The following direct monitor will */
    /* trap and handle any *ESCAPE exceptions. */
    #pragma exception_handler(LABEL_1, ca, 0, _C2_MH_ESCAPE, \
        _CTLA_HANDLE)
    /* Generate exception(divide by zero). The CTL_ACTION specified */
    /* should take effect (exception handled and logged), execution */
    /* resumes at LABEL_1. */
    a/=a;
    printf ("We should never reach this point\n");
    LABEL_1: printf("The MCH1211 exception was handled\n");
    #pragma disable_handler
    /* The following direct monitor will */
    /* only trap and handle MCH3601 exceptions */
    #pragma exception_handler(LABEL_2, ca, 0, _C2_MH_ESCAPE, \
        _CTLA_HANDLE, "MCH3601")
    /* Generate MCH3601(*ESCAPE message). The CTL_ACTION specified */
    /* should take effect (exception handled and logged), execution */
    /* resumes at LABEL_2. */
    *p='X';
    printf ("We should never reach this point\n");
    LABEL_2: printf("The MCH3601 exception was handled\n");
}

```

Figure 160. T1520XH3 – ILE C Source to Use Direct Monitors with Labels as Handlers

The output is:

```

The MCH1211 exception was handled
The MCH3601 exception was handled

```

Using ILE Condition Handlers

ILE condition handlers allow you to register one or more condition handlers at runtime. To register an ILE condition handler, use the Register ILE Condition Handler (CEEHDLR) bindable API. Include the

<lecond.h> header file in your source code when using these APIs. ILE condition handlers may be unregistered by calling the Unregister ILE Condition Handler (CEEHDLU) bindable API.

Condition handlers are exception handlers that are registered at runtime by using the Register ILE Condition Handler (CEEHDLR) bindable API. They are used to handle, percolate or promote exceptions. The exceptions are presented to the condition handlers in the form of an ILE condition.

When to Use an ILE Condition Handler

If you want to have a consistent mechanism of condition handling across several ILE languages (or for scoping exception handling to a call stack entry), use the ILE bindable API CEEHDLR. Unlike the signal handler, which is scoped to the activation group, CEEHDLR is scoped to the function that calls it.

The ILE condition handler uses ILE conditions to allow greater cross-system consistency. An ILE condition is a system-independent representation of an error condition in an HLL.

Example of ILE Source that Uses Condition Handlers

The following example shows the source for a program MYPGM:

```
#include <lecond.h>
#include <stdio.h>
void my_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new );
void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new );
void fred(void)
{
    _HDLR_ENTRY hdlr=my_handler;
    char *p = NULL;
    CEEHDLR(&hdlr, NULL,NULL);
    *p = 'x'; /* exception */
    CEEHDLU(&hdlr,NULL);
}
int main(void)
{
    _HDLR_ENTRY hdlr=main_handler;
    CEEHDLR(&hdlr,NULL,NULL);
    fred();
}
```

Figure 161. T1520XH5 – ILE C Source to Use ILE Condition Handlers – main()

In the example MYPGM:

- The procedure main() registers the condition handler main_handler.
- The procedure main() calls the function fred() which registers the condition handler my_handler.
- Function fred() gets an exception causing my_handler to get control, followed by main_handler.

The main() function is a control boundary. The exception is considered unhandled, so a function check is sent to function fred(). Handlers my_handler and main_handler are called again, this time for the function check. Neither of them handle the function check, so the program ends abnormally and a CEE9901 message is sent to the caller of the main() function.

Example of a Service Program that Provides ILE Condition Handlers

The following example shows the source for the service program HANDLERS:

```
#include <signal.h>
#include <stdio.h>
#include <lecond.h>
/* HANDLERS *SRVPGM (*CALLER) */
void my_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new)
{
    return;
}
void main_handler(_FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new)
{
    printf("In main_handler\n");
}
```

Figure 162. T1520XH6 – ILE C Source to Use ILE Condition Handlers – Service Program

Examples of Handling an Exception

The following example shows how to use a condition handler to handle an exception. In the example:

- ILE condition handler `cond_hdlr` is registered in the `main()` function using `CEEHDLR`.
- An MCH1211 (divide-by-zero) exception then occurs. Handler `cond_hdlr` is called and it indicates that the exception should be handled.
- Control then resumes in the `main()` function.

The following steps create and run the source in [Figure 163 on page 260](#).

1. To create the program T1520IC6, enter:

```
CRTBND CPG(MYLIB/T1520IC6) SRCFILE(QCPPL/QACSRC)
```

2. To run the program T1520IC6, enter:

```
CALL PGM(MYLIB/T1520IC6)
```

The output is:

```
condition was raised: Facility_ID = MCH, MsgNo = 0x1211
The condition was handled.
Press ENTER to end terminal session.
```

```
/* This program uses the ILE bindable API CEEHDLR to handle a          */
/* condition.                                                         */
#include <stdio.h>
#include <stdlib.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in main().      */
void cond_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    *rc = CEE_HDLR_RESUME; /* handle the condition */
    printf("condition was raised: Facility_ID = %.3s, MsgNo = 0x%4.4x\n",
           cond->Facility_ID, cond->MsgNo);
}
int main(void)
{
    _HDLR_ENTRY hdlr = cond_hdlr;
    _FEEDBACK fc;
    int x,y;
    int zero = 0;
    /* Register the condition handler.                                  */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(88);
    }
    /* Cause a divide by zero condition.                               */
    x = y / zero;
    /* The code resumes here after the condition has been handled.    */
    printf("The condition was handled.\n");
}
```

Figure 163. T1520IC6 – ILE C Source to Use ILE Condition Handlers

Example of Handling a Divide-By-Zero Exception

In this example:

- The bindable API `CEEHDLR` registers the `main_hdlr` in function `main()`.
- The bindable API `CEEHDLR` registers the `fred_hdlr` in function `fred()`.
- An MCH1211 (divide-by-zero) exception occurs.
- Handler `fred_hdlr` is called to test if the exception is an MCH1211.
- The result code in the condition handler is set to percolate to the next condition handler.
- Handler `fred_hdlr` returns without handling the exception, causing `main_hdlr` to be called.

- The user-supplied token is updated to the value '1' and the result code is set to handle the exception.
- Handler `main_hdlr` returns, and the exception is handled.
- Control resumes in `fred()` following the statement that caused the divide-by-zero.

This example uses the source shown in [“T1520IC7 – ILE C Source to Percolate a Message to Handle a Condition”](#) on page 261.

1. To create the program T1520IC7, enter:

```
CRTBNDC PGM(MYLIB/T1520IC7) SRCFILE(QCPPLP/QACSRC)
```

2. To run the program T1520IC7, enter:

```
CALL PGM(MYLIB/T1520IC7)
```

The output is:

```
in fred_hdlr, percolate exception.
in main_hdlr: Facility_ID = MCH, MsgNo = 0x1211
Resume here because resume cursor not moved and main_hdlr handled the exception.
A condition was percolated from fred() to main() and was then handled.
Press ENTER to end terminal session.
```

T1520IC7 – ILE C Source to Percolate a Message to Handle a Condition

```
/* This program uses the ILE bindable API CEEHDLR to enable handlers */
/* that percolate and handle a condition. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>
/* A condition handler registered by a call to CEEHDLR in fred(). */
void fred_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    if (!memcmp(cond->Facility_ID, "MCH", 3) && cond->MsgNo == 0x1211)
    {
        *rc = CEE_HDLR_PERC; /* ... let it percolate to main ... */
        printf("in fred_hdlr, percolate exception.\n");
    }
    else
    {
        *rc = CEE_HDLR_RESUME; /* ... otherwise handle it. */
        printf("in fred_hdlr, handle exception.\n");
    }
}
/* A condition handler registered by a call to CEEHDLR in main(). */
void main_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    printf("in main_hdlr: Facility_ID = %.3s, MsgNo = 0x%4.4x\n",
        cond->Facility_ID, cond->MsgNo);
    **(_INT4 **)token = 1; /* Update the user's token. */
    *rc = CEE_HDLR_RESUME; /* Handle the condition */
}
int fred(void)
{
    _HDLR_ENTRY hdlr = fred_hdlr;
    _FEEDBACK fc;
    int x,y, zero = 0;
    /* Register the handler without a token. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    printf("Resume here because resume cursor not moved and main_hdlr\n"
        " handled the exception\n");
}
int main(void)
{
    _HDLR_ENTRY hdlr = main_hdlr;
```

```

    _FEEDBACK fc;
    volatile _INT4 token=0, *tokenp = &token;
    /* Register the handler with a token of type _INT4.          */
    CEEHDLR(&hdlr, (_POINTER *)&tokenp, &fc);

    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(99);
    }
    fred();
    /* See if the condition handler for main() updated the token. */
    if (*tokenp == 1)
        printf("A condition was percolated from fred() to main() and"
              " was then handled.\n");
}

```

Example of Promoting an Exception

In the following example:

- In `main()`, the bindable API `CEEHDLR` registers the `main_hdlr`.
- In `fred()`, the bindable API `CEEHDLR` registers the `fred_hdlr`.
- An MCH1211 (divide-by-zero) exception occurs and the handler `fred_hdlr` is called.
- The handler `fred_hdlr` moves the resume cursor to the resume point in the `main()` function using the bindable API `CEEMRCR`.
- The handler `fred_hdlr` builds a condition token for CEE9902, and the result code is set to promote.
- The handler `fred_hdlr` returns, and the original MCH1211 is promoted to a CEE9902.
- The handler `main_hdlr` is called because of the CEE9902 exception and the result code is set to handle the condition.
- The handler `main_hdlr` returns, and the CEE9902 is handled.
- Control resumes in the statement following the call to `fred()` in `main()`

The source code in program T1520IC8 is shown in [“T1520IC8 — ILE C Source to Promote a Message to Handle a Condition” on page 262.](#)

1. To create the program T1520IC8, , enter:

```
CRTBND CPGM(MYLIB/T1520IC8) SRCFILE(QCPPLE/QACSRC)
```

2. To run the program T1520IC8, enter:

```
CALL PGM(MYLIB/T1520IC8)
```

The output is status information messages:

```

in fred_hdlr: moving resumes. Facility_ID = MCH, MsgNo = 0x1211
promoting condition....
A condition was promoted from MCH1211 to CEE9902 by fred() and was handled by
the condition handler enabled in main().
Press ENTER to end terminal session.

```

T1520IC8 — ILE C Source to Promote a Message to Handle a Condition

```

/* This program uses the ILE bindable API CEEHDLR to promote a      */
/* divide by zero condition to a CEE9902.                            */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <leawi.h>

/* A condition handler registered by a call to CEEHDLR in fred().    */
void fred_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{

```



```

_INT4 type=1;
CEEMRCR(&type,NULL);
/* Move the resume cursor to the resume point */
/* in main(). */
/* If its a divide by zero error ... */
printf("in fred_hdlr: moving resumes. ");
printf("Facility_ID = %.3s, MsgNo = 0x%4.4x\n",
      cond->Facility_ID, cond->MsgNo);
if (!memcmp(cond->Facility_ID, "MCH", 3) && cond->MsgNo == 0x1211)
{
    *rc = CEE_HDLR_PROM; /*... Promote the condition to unexpected error.*/
    *new = *cond;
    memcpy(new->Facility_ID, "CEE", 3);
    new->MsgNo = 9902;
    printf("promoting condition....\n");
}
else
{
    *rc = CEE_HDLR_PERC; /*...Otherwise,Percolate to the next handler. */
    printf("percolating condition....\n");
}
}
/* A condition handler registered by a call to CEEHDLR in main(). */
void main_hdlr( _FEEDBACK *cond, _POINTER *token, _INT4 *rc, _FEEDBACK *new )
{
    if (!memcmp(cond->Facility_ID,"CEE",3) && cond->MsgNo == 9902;)
        **(_INT4 **)token = 1; /* Got the promoted CEE9902. */
    else
        **(_INT4 **)token = 2; /* It is not a CEE9902. */
    *rc = CEE_HDLR_RESUME; /* Handle the condition. */
}
int fred(void)
{
    _HDLR_ENTRY hdlr = fred_hdlr;
    _FEEDBACK fc;
    int x,y, zero = 0;
    /* Register the handler without a token. */
    CEEHDLR(&hdlr, NULL, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(88);
    }
    /* Cause a divide by zero condition. */
    x = y / zero;
    /* This is not the resume point because of the call to CEEMRCR in */
    /* fred_hdlr. */
    {
        printf("This is not the resume point: should not get here\n");
    }
}

```

```

int main(void)
{
    _HDLR_ENTRY hdlr = main_hdlr;
    _FEEDBACK fc;
    volatile _INT4 token=0, *tokenp = &token;
    /* Register the handler with a token of type _INT4. */
    CEEHDLR(&hdlr, (_POINTER *)&tokenp, &fc);
    if (fc.MsgNo != CEE0000)
    {
        printf("Failed to register the condition handler\n");
        exit(99);
    }
    fred();
    /* See if the condition handler for main() received the promoted */
    /* condition. */
    if (*tokenp == 1)
        printf("A condition was promoted from MCH1211 to CEE9902 by "
              "fred() and was handled by the condition handler enabled "
              "in main().\n");
}

```

Using the C/C++ Signal Handler

HLL-specific handlers are the language features that are defined for handling errors. The ILE C/C++ `signal()` function can be used to handle exception messages.

IBM i exceptions are mapped to C and C++ signals by the ILE C/C++ runtime environment. A signal handler determines the course of action for a signal. You cannot register a signal handler in an activation group that is different from the one you wish to call it from. If a signal handler is in a different activation group from the occurrence of the signal it is handling, the behavior is undefined.

When to Use the Signal Handler

For portable code across multiple platforms, use only the `signal()` function to handle exception. ILE condition handlers should be used if a consistent mechanism for handling exceptions across ILE enabled languages is required. If portability across ILE-enabled platforms is a concern, then ILE condition handlers and the `signal()` function can be used. Otherwise, all three types of handlers may be used. See [Using Both C/C++ Signal and ILE Exception Handlers](#).

Note:

Stream I/O functions trap the SIGIO signal, which is sent when normal data, OOB data, error conditions, or just about anything happens on any type of socket. Signal handlers that are registered for SIGIO are not called for exceptions that are generated when processing stream files.

Using the `signal()` function will always handle the exception implicitly (unless the signal action is SIG_DFL, in which case it would percolate the exception); with direct monitor handlers you either have to specify a control action that will implicitly handle the exception (`_CTLA_HANDLE`, `_CTLA_HANDLE_NO_MSG`, `_CTLA_IGNORE`, or `_CTLA_IGNORE_NO_MSG`), or you have to handle the exception explicitly within the handler function (when the control action `_CTLA_INVOKE` is specified), using either QMHCHGEM or an ILE condition handling API.

Note: Direct monitors are usually the fastest handlers.

The HLL-specific handler, which is the signal handler in ILE C, is global. It is enabled for all function calls in the activation group the `signal()` function is called. In ILE, condition handlers and direct monitor handlers are scoped to the function that enables them or until they are disabled in that function.

The following example illustrates that if you do not want to change the state of a signal handler when the signal function returns, then you must manage the state of the signal handler explicitly.

```
#include <signal.h>
void f(void)
{
    void (*old_state)(int);
    /* Save old state of signal action */
    old_state = signal(SIGALL, handlꝛ);
    /* Other code in your application */
    /* Reset state of signal          */
    signal(SIGALL, old_state);
}
```

Figure 164. ILE C Source to Manage the State of a Signal Handler

ILE condition handlers and direct monitor handlers do not have this requirement because they are not global handlers.

Raising Signals

Signals are raised implicitly or explicitly. To *explicitly* raise a signal, use the `raise()` function. Signals are *implicitly* raised by the IBM i when an exception occurs. For example, if you call a program that does not exist, an implicit signal is raised indicating that the program object could not be found.

Signal Handling Function Prototypes

The header file `<signal.h>` contains a number of function prototypes that are associated with signal handling.

The following functions can be used with signal handling in your program:

- `raise()`

- `signal()`
- `_GetExcData()`

The `_GetExcData()` function is an ILE C/C++ extension that allows you to obtain information about the exception message associated with the signal and returns a structure containing information about the exception message. The `_GetExcData()` function returns the same structure that is passed to the `#pragma exception_handler` directive.

The `signal()` function specifies the action that is performed when a signal is raised. There are ten signals that are represented as macros in the `<signal.h>` header file. In addition, the macro `SIGALL` has the semantics of a signal but with some unique characteristics. The ten signals are as follows:

SIGABRT

Abnormal program end.

SIGFPE

Arithmetic operation error, such as dividing by zero.

SIGILL

An instruction that is not allowed.

SIGINT

System interrupt, such as receiving an interactive attention signal.

SIGIO

Record file error condition.

SIGOTHER

All other `*ESCAPE` and `*STATUS` messages that do not map to any other signals.

SIGSEGV

The access to storage is not valid.

SIGTERM

A end request is sent to the program.

SIGUSR1

Reserved for user-defined signal handler.

SIGUSR2

Reserved for user-defined signal handler.

`SIG_IGN` and `SIG_DFL` are signal actions that are also included in the `<signal.h>` header file.

SIG_IGN

Ignore the signal.

SIG_DFL

Default action for the signal.

`SIGALL` is an ILE C/C++ extension that allows you to register your own default-handling function for all signals whose action is `SIG_DFL`. This default-handling function can be registered by using the `signal()` function with `SIGALL`, as shown in the example section. A function check is not a signal and cannot be monitored for by the signal function. `SIGALL` cannot be signaled by the `raise()` function.

How the ILE C/C++ Runtime Environment Handles Signals

When a signal is received, the Integrated Language Environment C/C++ runtime environment handles the signal in one of three ways:

- If the value of the function is `SIG_IGN`, then the signal is ignored because the exception is handled by the runtime environment and no signal handler is called. If the message that is mapped to the signal is an `*ESCAPE` or `*NOTIFY` message, then it is placed in the job log.
- If the value of the function is a pointer to a function, then the function that is addressed by the pointer is called.

- If the value of the function is SIG_DFL, then the system uses the value registered for SIGALL (choosing one of the three ways described here). If the value of the function for SIGALL is SIG_DFL then the exception is percolated.

Note: The value of the function is the function argument on the call to the `signal()` function.

Resetting the Signal Action

A signal handling function is called when an exception occurs or when a signal is raised. A handler is defined with the `signal()` function. The value you assign on the `sig` parameter is associated with the function referred to on the `funct` parameter.

When a signal handler is called in a C program, its corresponding signal action is set to SIG_DFL. You must reset the signal action if you want to handle the same signal again. If you do not, the default action is taken on subsequent exceptions. The handling of the signal can be reset from inside or outside the handler by calling `signal()`.

Example:

The following figure shows source code that resets signal handlers.

```
signal ( SIGINT, &myhandler );
raise ( SIGINT );          /* signal is handled by myhandler.    */
...
raise ( SIGINT );          /* signal is handled by SIG_DFL.    */
...
signal ( SIGINT, &myhandler ); /* reset signal handler to myhandler. */
raise ( SIGINT );          /* signal is handled by myhandler.    */
```

Figure 165. Resetting Signal Handlers

In [Figure 165 on page 266](#):

- The default can be reset to SIG_IGN, another handler, or the same handler. You can recursively call the signal handler. Once stacked, multiple signal handler calls behave like any other calls. For example, if the action signal to the previous caller is chosen, the control will not be returned to the preceding caller (even if that call is another signal handler) but goes back to the previous caller.
- The `signal()` function returns the address of the previous signal handler for the specified signal, and sets the address of the new signal handler.

Stacking Signal Handlers

You can stack the signal handlers yourself using the value returned by `signal()`, as shown in the following figure.

```
void (*func1) ();
void (*func2) ();
func1 = signal ( SIGINT, &handler2 ); /*func1 contains the address of */
                                     /*a previous signal handler or */
                                     /*SIG_DFL if no handler has been */
                                     /*defined. */
func2 = signal ( SIGINT, func1);      /*func2 contains the address of */
                                     /*handler2. */
```

Figure 166. Stacking Signal Handlers

Example: Setting Up a Signal Handler

The following example shows how to set up a signal handler. The example illustrates that when there is no signal handler set up the default action for SIGIO is SIG_IGN. The exception is ignored. When a signal handler is set up for SIGIO, the signal handler is called.

Instructions

1. To create the program T1520SIG, enter:

```
CRTBND C PGM(MYLIB/T1520SIG) SRCFILE(QCPPLP/QACSRC)
```

[“Source Code Sample that Sets Up Signal Handlers”](#) on page 267 shows the source code.

2. To run program T1520SIG, enter:

```
CALL PGM(MYLIB/T1520SIG)
```

The output appears in a series of screens:

```
The first read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
```

```
The first read finishes
The second read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In SIGIO handler
Exception message ID is CPF5001
Signal raised is 9
The second read finishes
The third read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In SIGALL handler
Exception message ID is CPF5001
Signal raised is 9
The third read finishes
Press ENTER to end terminal session.
==>
F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom F19=Left F20=Right F21=User Window
```

Source Code Sample that Sets Up Signal Handlers

```
#include <stdio.h>
#include <signal.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define FILE_NAME "QTEMP/MY_FILE"
#define RCD_LEN 80
#define NUM_RCD 5
/* The signal handler for SIGIO. */
static void handler_SIGIO(int sig)
{
    printf("In SIGIO handler\n");
    printf("Exception message ID is %7.7s\n", _EXCP_MSGID);
    printf("Signal raised is %d\n", sig);
}
/* The signal handler for SIGALL. */
static void handler_SIGALL(int sig)
{
    _INTRPT_Hndlr_Parms_T data;
    _GetExcData(&data);
    printf("In SIGALL handler\n");
    printf("Exception message ID is %7.7s\n", data.Msg_Id);
    printf("Signal raised is %d\n", sig);
}
int main(void)
{
    _RFILE *fp;
    int i;
    char buf[RCD_LEN];
    char cmd[100];
```

```

/* Create a file. */
    sprintf(cmd, "CRTPF FILE(%s) RCDLEN(%d)", FILE_NAME, RCD_LEN);
    system(cmd);
/* Open the file for write. */
    if ( (fp = _Ropen(FILE_NAME, "wr")) == NULL )
    {
        printf("Open for write fails\n");
        exit(1);
    }
/* Write some data into the file. */
    memset(buf, '1', RCD_LEN);
    for ( i = 0; i < NUM_RCD; i++ )
    {
        _Rwrite(fp, buf, RCD_LEN);
    }
    _Rclose(fp);
/* Open the file for the first read. */
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the first read fails\n");
        exit(2);
    }
/* Read until end-of-file. */

/* Since there is no signal handler set up and the default
/* action for SIGIO is SIG_IGN, the EOF exception is ignored.
    i = 1;
    printf("The first read starts\n");
    while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
    _Rclose(fp);
    printf("The first read finishes\n");
/* Set up a signal handler for SIGIO.
    signal(SIGIO, handler_SIGIO);
/* Open the file for the second read.
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the second read fails\n");
        exit(3);
    }
/* Read until end of file.
/* Since a signal handler is set up for SIGIO, the signal
/* handler is called when the EOF exception is generated.
    i = 1;
    printf("The second read starts\n");
    while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
    _Rclose(fp);
    printf("The second read finishes\n");
/* Set up a signal handler for SIGALL.
    signal(SIGALL, handler_SIGALL);
/* Open the file for the third read.
    if ( (fp = _Ropen(FILE_NAME, "rr")) == NULL )
    {
        printf("Open for the third read fails\n");
        exit(4);
    }
/* Read until end of file.
/* Since there is no signal handler for SIGIO but there is a
/* signal handler for SIGALL, the signal handler for SIGALL
/* is called when the EOF exception is generated. But
/* the signal ID passed to the SIGALL signal handler is still
/* equal to SIGIO.
    i = 1;
    printf("The third read starts\n");
    while ( _Rreadn(fp, buf, RCD_LEN, __DFT)->num_bytes != EOF )
    {
        printf("Read record %d\n", i++);
    }
    _Rclose(fp);
    printf("The third read finishes\n");
}

```

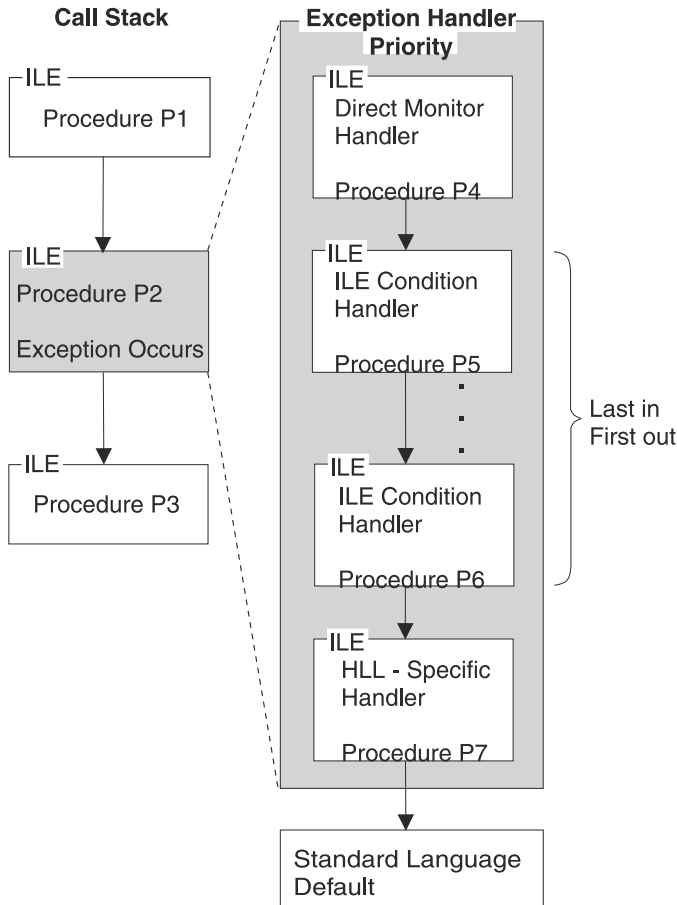
Using Both C/C++ Signal and ILE Exception Handlers

Exception handler priority becomes important if you use both language-specific error handling (C signal) and additional ILE exception handler types.

Order of Priority

For the call stack entry that incurred the exception, the system calls handlers in the following prioritized order:

1. Direct monitors
2. ILE condition handlers
3. `signal()`



RV2W1041-3

Figure 167. Exception Handler Priority

Example of Using a Direct Monitor Handler and Signal Handler Together

The following example shows you how to use the `#pragma exception_handler` directive and the `signal()` function together. This example also shows how an exception is handled using SIGIO. An end-of-file message is mapped to SIGIO. The default for SIGIO is SIG_IGN. It also shows that when both a HLL-specific handler and direct monitor handler are defined, the direct monitor handler is called first.

This example uses the source shown in [“Example of Source that Illustrates How to Use Direct Monitor Handlers”](#) on page 255.

1. To create the program T1520ICA, enter:

```
CRTBND C PGM(MYLIB/T1520ICA) SRCFILE(QCPPLE/QACSRC)
```

2. To run the program T1520ICA, enter:

```
CALL PGM(MYLIB/T1520ICA)
```

The first screen of output is shown below:

```
The first read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
The first read finishes
The second read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In direct monitor handler
Exception message ID is CPF5001
The second read finishes
The third read starts
Read record 1
```

The second screen of output follows:

```
Read record 2
Read record 3
Read record 4
Read record 5
In direct monitor handler
Exception message ID is CPF5001
In signal handler
Exception message ID is CPF5001
The third read finishes
The fourth read starts
Read record 1
Read record 2
Read record 3
Read record 4
Read record 5
In signal handler
Exception message ID is CPF5001
The fourth read finishes
Press ENTER to end terminal session.
===>
F3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
F18=Bottom F19=Left F20=Right F21=User Window
```

Handling Nested Exceptions

Exceptions can be nested. A nested exception is an exception that occurs while another exception is being handled. When this happens, the processing of the first exception is temporarily suspended. Exception handling begins again with the most recently generated exception.

Note: If a nested exception causes the program to end, the exception handler for the first exception may not complete.

Example:

The following example shows a nested exception.


```

#include <signal.h>
void hdlr_hdlr(_INTRPT_HndlR_Parms_T * __ptr128 parms)
{
    /* Handle exception 2 using QMHCHGEM. */
}
void main_hdlr(_INTRPT_HndlR_Parms_T * __ptr128 parms)
{
#pragma exception_handler(hdlr_hdlr,0,0,_C2_MH_ESCAPE)
    /* Generate exception 2. */
    /* Handle exception 1 using QMHCHGEM. */
}
int main(void)
{
#pragma exception_handler(main_hdlr,0,0,_C2_MH_ESCAPE)
    /* Generate exception 1. */
}

```

Figure 168. ILE C Source to Nest Exceptions

In this example, the `main()` function generates an exception which causes `main_hdlr` to get control. The handler `main_hdlr` generates another exception which causes `hdlr_hdlr` to get control. The handler `hdlr_hdlr` handles the exception. Control resumes in `main_hdlr`, and it handles the original exception.

As this example illustrates, you can get an exception within an exception handler. To prevent exception recursion, exception handler call stack entries act like control boundaries with regards to exception percolation. Therefore it is recommended that you monitor for exceptions within your exception handlers.

Using Cancel Handlers



Cancel handlers are used by C++ to call destructors during stack unwinding. It is recommended that you use the C++ try-catch-throw feature to ensure objects are destructed properly.

Note: For information about using the try-catch-throw feature, see the *ILE C/C++ Language Reference*.

A cancel handler may be enabled around a body of code inside a function. When a cancel handler is enabled it only gets control if the suspend point of the call stack entry is inside that code (within the `#pragma cancel_handler` and `#pragma disable_handler` directives), and the call stack entry is cancelled.

The `#pragma cancel_handler` directive provides a way to statically register a cancel handler within a call stack entry (or suspend point within a call stack entry). The Register Call Stack Entry Termination User Exit Procedure (CEERTX) and the Unregister Call Stack Entry Termination User Exit Procedure (CEETUX) ILE bindable APIs provide a way of dynamically registering a user-defined routine to be executed when the call stack entry for which it is registered is cancelled.

Cancel handlers provide an important function by allowing you to get control for clean-up and recovery actions when call stack entries are ended by something other than a normal return.

On the `#pragma cancel_handler` directive, the name of the cancel handler routine (a bound ILE procedure) is specified, along with a user-defined communications area. It is through the communications area that information is passed from the application to the handler function. When the cancel handler function is called, it is passed a pointer to a structure of type `_CNL_HndlR_Parms_T` which is defined in the `<except.h>` header file. This structure contains a pointer to the communications area in addition to some other useful information that is passed by the system. This additional information includes the reason why the call was cancelled.

Example:

The following simple example illustrates the use of the ILE Cancel Handler mechanism. This capability allows an application the opportunity to have a user-provided function called to perform things such as error reporting and logging, closing of files, etc. when a particular function invocation is cancelled. The usual ways that cause cancelation to occur are: using the `exit()` function or the `abort()` function, using

the `longjmp()` function to jump to an earlier call and having a CEE9901 Function Check generated from an unhandled exception.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <except.h>
/*-----*/
/* The following function is called a "cancel handler". It is */
/* registered for a particular invocation (function) with the */
/* #pragma cancel_handler directive. The variable identified */
/* on this directive as the "communications area" can be accessed */
/* using the 'Com_Area' member of the _CNL_Hndlr_Parms_T structure. */
/*-----*/
void CancelHandlerForReport( _CNL_Hndlr_Parms_T *cancel_info ) {
    printf("In Cancel Handler for function 'Report'...\n");
    /* Changing the value in the communications area will update the */
    /* 'return_code' variable in the invocation being cancelled */
    /* (in function 'Report' in this example). Note that the */
    /* ILE C compiler will issue a warning for the following */
    /* statement since it uses a non-ISO C compliant technique. */
    /* However, this will not affect the expected runtime behavior. */
    /* Set "return_code" in Report to an arbitrary number. */
    *(volatile unsigned *)cancel_info->Com_Area = 500;
    printf("Communication Area now has the value: %d\n",
        *(volatile unsigned *)cancel_info->Com_Area );
    printf("Leaving Cancel Handler for function 'Report'...\n");
}

```

```
/*-----*/
/* The following function is also called a cancel handler but has */
/* been registered for the 'main' function. That is, when the */
/* 'main' function is cancelled, this function will automatically */
/* be called by the system. */
/*-----*/
void CancelHandlerForMain( _CNL_Hndlr_Parms_T *cancel_info ) {
    printf("In Cancel Handler for function 'main'...\n");
    /* Changing the value in the communications area will update the */
    /* 'return_code' variable in the invocation being cancelled */
    /* (in function 'main' in this example). Note that the */
    /* ILE C compiler will issue a warning for the following */
    /* statement since it uses a non-ISO C compliant technique. */
    /* However, this will not affect the expected runtime behavior. */
    /* Set "return_code" in main to an arbitrary number. */
    *(volatile unsigned *)cancel_info->Com_Area = 999;
    printf("Communication Area now has the value: %d\n",
        *(volatile unsigned *)cancel_info->Com_Area );
    printf("Leaving Cancel Handler for function 'main'...\n");
}
/*-----*/
/* The following is simple function that registers another function */
/* (named 'CancelHandlerForReport' in this example) as its "cancel */
/* handler". When 'exit()' is used from this function, then this */
/* invocation and all prior invocations are cancelled by the system */
/* and any registered cancel handlers functions are automatically */
/* called. */
/*-----*/
void Report( void ) {
    volatile unsigned return_code; /* communications area */
    #pragma cancel_handler( CancelHandlerForReport, return_code )
    printf("in function Report()...about to call 'exit'...\n");
    /* Using the exit function will cause this function invocation */
    /* and all function invocations within this program to be */
    /* cancelled. If any of the functions being cancelled have */
    /* cancel handlers enabled, then those cancel handler functions */
    /* will be called by the system after each cancellation. */
    exit( 99 ); /* exit with an arbitrary value */
    printf("in function Report() just after calling 'exit'...\n");
    #pragma disable_handler
}
/*-----*/
/* In the 'main()' function a cancel handler is registered so that */
/* the function 'CancelHandlerForMain()' is called if 'main()' is */
/* cancelled. */
/*-----*/
int main( void ) {
    volatile unsigned return_code; /* communications area */
}

```

```

#pragma cancel_handler( CancelHandlerForMain, return_code )
return_code = 0;          /* initialize return code which will */
                          /* eventually be set in the cancel handler */
printf("In main() about to call Report()...\n");
Report();
printf("...back from calling Report(). \n");
printf("return_code = %d \n", return_code );
#pragma disable_handler
}

```

```

/*-----*/
/* This program will result in the following screen output: */
/* */
/* In main() about to call Report()... */
/* in function Report()...about to call 'exit'... */
/* In Cancel Handler for function 'Report' ... */
/* Communication Area now has the value: 500 */
/* Leaving Cancel Handler for function 'Report'... */
/* In Cancel Handler for function 'main' ... */
/* Communication Area now has the value: 999 */
/* Leaving Cancel Handler for function 'main'... */
/*-----*/

```

Example: Using a Variety of Ways to Detect Errors and Handle Exceptions

The following example shows a number of ways to handle errors, including:

- Checking the major/minor return code
- Checking errno
- Getting error information from the _EXCP_MSGID global variable
- Signal handling with signal

Instructions

1. To create the display file T1520DDJ using the DDS source shown in [Figure 169 on page 274](#), type:

```
CRTDSPF FILE(MYLIB/T1520DDJ) SRCFILE(QCPPL/QADDSSRC)
```

2. To create the program T1520EHD using the source shown below, type:

```
CRTBND CPGM(MYLIB/T1520EHD) SRCFILE(QCPPL/QACSRC)
```

Note: Program T1520EHD uses `signal()` function to raise a SIGIO signal, which is sent when normal data, OOB data, error conditions, or just about anything happens on any type of socket.

3. If you need to add library MYLIB to the library list, enter:

```
ADDLIB LIB(MYLIB)
```

4. To run the program T1520EHD, enter:

```
CALL PGM(MYLIB/T1520EHD)
```

The output is:

```

                PHONE BOOK
                Name: Smith, John
                Address: 2711 Westsyde Rd.
                Phone #: 721-9729
<ENTER> : Saves changes
f3      : Exits with changes saved
f5      : Brings back original field values

```

Source Code Samples

The figures in this section show the source code used in [“Instructions” on page 273](#).

```

A          DSPSIZ(24 80 *DS3)
A          INDARA
A      R  PHONE
A          CF03(03 'EXIT')
A          CF05(05 'REFRESH')
A          7 28'Name:'
A      NAME      11A B 7 34
A          9 25'Address:'
A      ADDRESS   20A B 9 34
A          11 25'Phone #:'
A      PHONE_NUM 8A B 11 34
A          1 35'PHONE BOOK'
A          DSPATR(HI)
A          16 19'<ENTER> : Saves changes'
A          17 21'f3   : Exits with changes saved'
A          18 21'f5   : Brings back original field values'
A 05          21 32'Screen refreshed'
A 05          DSPATR(HI)

```

Figure 169. T1520DDJ – DDS Source for a Phone Book Display

```

/* This program illustrates how to:                                     */
/* - check the major/minor return code                               */
/* - check the errno global variable                                */
/* - get error information from the                                  */
/*   _EXCP_MSGID global variable                                   */
/* - use the signal function.                                       */
/*                                                                    */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
#include <errno.h>
#include <signal.h>
#include <recio.h>

#define IND_ON '1'
#define IND_OFF '0'
#define HELP 0
#define EXIT 2
#define REFRESH 4
#define FALSE 0
#define TRUE 1

typedef struct PHONE_LIST_T {
    char name[11];
    char address[20];
    char phone[8];
}PHONE_LIST_T;
void error_check(void);
/* The error checking routine.                                       */
void error_check(void)
{
    if (errno == EIOERROR || errno == EIORECERR)
        printf("id:%7.7s\n", _EXCP_MSGID);
    if (strcmp(_Maj_Min_rc.major_rc,"00",2) ||
        strcmp(_Maj_Min_rc.minor_rc,"00",2))

        printf("Major : %2.2s\tMinor : %2.2s\n",
            _Maj_Min_rc.major_rc,_Maj_Min_rc.minor_rc);
    errno = 0;
}
/* The signal handler routine.                                       */

void sighd(int sig)
{
    signal(SIGIO,&sighd);
}

/* M A I N L I N E

int main(void)
{
    FILE *dspf;
    PHONE_LIST_T phone_inp_rec,
        phone_out_rec = { "Smith, John",
            "2711 Westsyde Rd. ",
            "721-9729" };
}

```

Figure 170. T1520EHD – ILE C Source to Handle Exceptions

```

_SYSindara indicator_area;
int ret_code;

errno = 0;

signal(SIGIO,&signd); /* Register signd as a handler for I/O exceptions */

if ((dspf = fopen("*LIBL/T1520DDJ", "ab+ type=record indicators=y"))
    == NULL)
{
    printf("Display file could not be opened");
    exit(1);
}
_Rindara((_RFILE *) dspf,indicator_area);
_Rformat((_RFILE *) dspf,"PHONE");

memset(indicator_area,IND_OFF,sizeof(indicator_area));
do
{
    ret_code = fwrite(&phone_out_rec,1,sizeof(phone_out_rec),dspf);
    error_check(); /* Write the records to the display file. */
    ret_code = fread(&phone_inp_rec,1,sizeof(phone_inp_rec),dspf);
    error_check(); /* Read the records from the display file. */
    if (indicator_area[EXIT] == IND_ON)
        phone_inp_rec = phone_out_rec;
}
while (indicator_area[REFRESH] == IND_ON);

_Rclose((_RFILE *)dspf);
}

```

Figure 171. T1520EHD – ILE C Source to Handle Exceptions continued

Using pointers in a Program

In ISO C, a *pointer type* is derived from a function type, a data object type, or an incomplete type. On the IBM i, pointer types can also be derived from other IBM i entities such as system objects (for example, programs), code labels, and process objects. These pointer types are usually referred to as IBM i pointers and they are used extensively in the ILE C Machine Interface Library and in the ILE C/C++ exception handling structures and functions.

Note: The *ILE C/C++ for AS/400 MI Library Reference* contains information on ILE C Machine Interface Library Functions.

This topic describes:

- [IBM i pointer types](#)
- [Using an open pointer](#)
- [Using other pointers](#)
- [Declaring pointer variables](#)
- [Pointer casting](#)

IBM i pointer Types

The IBM i pointer types are:

Open

Pointers that can hold any of the other pointer types

Space

Generic pointers to data objects.

Function

System pointers to *PGM objects or procedure pointers to bound ILE procedures.

System

Pointers to system objects such as queues, indexes, libraries, and *PGM objects.

Label

Pointers to fixed locations within the executable code of a procedure or function.

Invocation

Pointers to process objects for procedure (function) calls under ILE, or program calls under EPM or OPM.

Suspend

Pointer to the location in a procedure where control has been suspended.

These pointer types, as well as pointers to data objects and incomplete types, are not data-type compatible with each other. For example, a variable declared as a pointer to a data object cannot be assigned the value of a function pointer or system pointer. A system pointer cannot be compared for equality with an invocation pointer or pointer to a data object. The above is not true for open pointers.

Note:

- Label pointers are only used by the `setjmp` macro.
- An open pointer is a pseudo-pointer type. It may contain any other pointer type, but it is not a pointer type unto itself.

Using Open Pointers

Open pointers inhibit optimization. Use them only when absolutely necessary. Before using open pointers in an ILE C/C++ program, consider the following characteristics and constraints:

- An open pointer maps to a void pointer.
- An open (void) pointer can hold any type of pointer. It is data-type compatible with all pointer types on the system. No compile time error occurs when you cast an open pointer to other pointer types, or when you cast other pointer types to an open pointer.

Note: You may receive a runtime exception if the pointer contains a value unsuitable for the context (for example, a system pointer in a pointer addition expression).

- An open pointer can be assigned to any type of pointer.

Note: You may receive a runtime exception if the type of pointer held in the open pointer is not data-type compatible with the target of the assignment.

- An open pointer can be compared for equality (`==`, `!=`) to any pointer type.
- An open pointer can be compared in a relational operation (`<`, `>`, `<=`, `>=`) to another open pointer or to a data object pointer expression other than the NULL pointer.

Note: You may receive a runtime exception if the type of pointer that is held in the open pointer is not a pointer to a data object.

Using Pointers Other than Open Pointers

Before you use pointers in an ILE C/C++ program, consider the following characteristics and constraints:

- In an equality operation (`==`, `!=`):
 - A NULL pointer can be assigned to and compared for equality (`==`, `!=`) with a pointer of any type.
 - A pointer can be assigned to and compared for equality (`==`, `!=`) only with a pointer of the same type or an open pointer; otherwise a compile time error occurs.
 - The conditional expression `if (!ptr)` is equivalent to the expression `if (ptr == NULL)`.
- In a relational operation (`<`, `>`, `<=`, `>=`):
 - A NULL pointer cannot be used with any pointer type.
 - Only pointers to data objects or open pointers that contain pointers to data objects can be used, otherwise a compile time error or runtime exception might occur.
- In arithmetic operations (`+`, `-`, `++`, `--`, only pointers to data objects can be used, otherwise a compile time error will occur.

Declaring Pointer Variables

You can use ILE C/C++ to declare

- Pointers to data
- Function pointers (pointers to bound ILE procedures)

Declaring IBM i Pointer Variables in C and C++

Pointers to *PGM objects (programs) can be declared in either of the following ways:

- By declaring a pointer to a typedef that has been specified to have OS-linkage with the `#pragma linkage` directive or extern OS linkage.
- By declaring a system pointer (`_SYSPTR`).

You can declare variables of the other IBM i pointer types by using the type definitions (typedefs) that are provided by the ILE C `<pointer.h>` header file.

Figure 172 on page 278 shows IBM i C pointer declarations. Figure 173 on page 278 shows IBM i C++ pointer declarations.

```
#include <pointer.h> /* The pointer header file. */
_SYSPTR sysp; /* A system pointer. */
_SPCPTR spcp; /* A space pointer. */
_INVPTR invp; /* An invocation pointer. */
_OPENPTR opnp; /* An open pointer. */
_LBLPTR lblp; /* A label pointer. */
void (*fp) (int); /* A function pointer. */
#pragma datamodel (p128)
#pragma linkage (OS_FN_T, OS)
#pragma datamodel(pop)
typedef void (OS_FN_T) (int); /* Typedef of an OS-linkage function.*/
OS_FN_T * os_fn_p; /* An OS-linkage function pointer. */
int * ip; /* A pointer to a data object. */
```

Figure 172. ILE C Source to Declare Pointer Variables

```
#include <pointer.h> /* The pointer header file. */
_SYSPTR sysp; /* A system pointer. */
_SPCPTR spcp; /* A space pointer. */
_INVPTR invp; /* An invocation pointer. */
_OPENPTR opnp; /* An open pointer. */
_LBLPTR lblp; /* A label pointer. */
void (*fp) (int); /* A function pointer. */
#pragma datamodel (p128)
/* Typedef of an OS-linkage function. */
extern "OS" typedef void (OS_FN_T) (int);
#pragma datamodel(pop)
int * ip; /* A pointer to a data object. */
```

Figure 173. ILE C++ Source to Declare Pointer Variables

Declaring a Function Pointer to a Bound Procedure in ILE C

A function pointer is a pointer that points to either a bound procedure (function) within an ILE program object, or an OS-linkage program object (system pointer) in the system.

Figure 174 on page 279 shows you how to declare a pointer to a bound procedure (a function that is defined within the same ILE program object):


```

int fct1( void ) {...}
int fct2( void ) {...}
int (*fct_ptr)(void) = fct1;
int main()
{
    fct_ptr();           /* Call fct1() using fct_ptr.      */
    fct_ptr = fct2;     /* Dynamically set fct_ptr to fct2.*/
    fct_ptr();           /* Call fct2() using fct_ptr.      */
}

```

Figure 174. ILE C Source to Declare a Pointer to a Bound Procedure

Declaring a Function Pointer with OS-Linkage in ILE C and ILE C++

Pointers to OS-linkage functions (programs) and system pointers (`_SYSPTR`) are data-type compatible. You can use a system pointer to hold the address of a program and then call that program through the system pointer.

Note: A call through a system pointer that contains the address of a system object that is not a program results in undefined behavior.

To force the ILE C compiler to associate system pointer types with the IBM i pointer types, do both of the following tasks:

- Define system pointer types as pointers to void (`void *`).
- Define the `#pragma` pointer directives in the header file.

Figure 175 on page 279 shows you how to declare a pointer to an IBM i program as a function pointer with OS-linkage in ILE C. Figure 176 on page 280 shows you how to declare a pointer to an IBM i program as a function pointer with OS-linkage in ILE C++.

Note: If the `#pragma` linkage OS directive is omitted from the code, the ILE C compiler assumes that `os_fct_ptr` is a pointer to a bound C function returning void, and will issue a compilation error for incompatible pointer types between `os_fct_ptr` and the system pointer returned by `rslvsp()`.

```

#include <miptrnam.h>
#include <stdio.h>
#pragma datamodel(p128)
typedef void (OS_fct_t) ( void );
#pragma linkage(OS_fct_t,OS)
#pragma datamodel(pop)
int main ( void )
{
    OS_fct_t *os_fct_ptr;  char   pgm_name[10];
    printf("Enter the program name : \n");
    scanf("%s", pgm_name);
    /* Dynamic assignment of a system pointer to program "MYPGM" */
    /* in *LIBL. The rslvsp MI library function will resolve to */
    /* this program at runtime and return a system pointer to */
    /* the program object. */
    os_fct_ptr = rslvsp(_Program, pgm_name, "*LIBL", _AUTH_OBJ_MGMT);
    os_fct_ptr();           /* OS-linkage *PGM call using a */
                           /* pointer. */
}

```

Figure 175. ILE C Source to Declare a Pointer to an IBM i Program as a Function Pointer

```

#include <miptrnam.h>
#include <stdio.h>
extern "OS" typedef void (OS_fct_t) (void);
int main ( void )
{
    OS_fct_t *os_fct_ptr;
    char    pgm_name[10];
    printf("Enter the program name : \n");
    scanf("%s", pgm_name);
    /* Dynamic assignment of a system pointer to program "MYPGM" */
    /* in *LIBL. The rslvsp MI library function will resolve to */
    /* this program at runtime and return a system pointer to */
    /* the program object. */
    os_fct_ptr = (OS_fct_t*) rslvsp(_Program, pgm_name, "*LIBL", _AUTH_OBJ_MGMT);
    os_fct_ptr();          /* OS-linkage *PGM call using a pointer */
}

```

Figure 176. ILE C++ Source to Declare a Pointer to an IBM i Program as a Function Pointer

Casting Pointers

In the C language, *casting* is a construct to view a data object temporarily as another data type.

When you cast pointers, especially for non-data object pointers, consider the following characteristics and constraints:

- You can cast a pointer to another pointer of the same IBM i pointer type.
 - Note:** If the ILE C compiler detects a type mismatch in an expression, a compile time error occurs.
- An open (void) pointer can hold a pointer of any type. Casting an open pointer to other pointer types and casting other pointer types to an open pointer does not result in a compile time error.
 - Note:** You might receive a runtime exception if the pointer contains a value unsuitable for the context.
- When you convert a valid data object pointer to a signed or unsigned integer type, the return value is the offset of the pointer. If the pointer is NULL, the conversion returns a value of zero (0).
 - Note:** It is not possible to determine whether the conversion originated from a NULL pointer or a valid pointer with an offset 0.
- When you convert a valid function (procedure) pointer, system pointer, invocation pointer, label pointer, or suspend pointer to a signed or unsigned integer type, the result is always zero.
- When you convert an open pointer that contains a valid space address, the return value is the offset that is contained in the address.
- You can convert an integer to pointer, but the resulting pointer value cannot be dereferenced. The right four bytes of such a pointer will contain the original integer value, and this value can be recovered by converting the pointer back to an integer.
 - Note:** This marks a change from behavior exhibited in earlier versions of ILE C, where integer to pointer conversions always resulted in a NULL pointer value.

Example:

Figure 177 on page 281 shows IBM i pointer casting:

```

#include <pointer.h>
#pragma datamodel(p128)
#pragma linkage(TESTPTR, OS)
#pragma datamodel(pop)
void TESTPTR(void);      /* System pointer to this program */
_SYSPTR sysp;           /* System pointer */
_OPENPTR opnp;          /* open pointer */
void (*fp)(void);       /* function pointer */
int i = 1;               /* integer */
int *ip = &i;           /* Space pointer */
void main (void) {
    fp = &main;          /* initialize function pointer */
    sysp = &TESTPTR;     /* initialize system pointer */

    i = (int) ip;        /* segment offset stored in i */
    ip = (int *) i;      /* address stored is invalid */
    i = (int) fp;        /* zero is stored in i */
    i = 2;
    fp = (void (*)()) i; /* address stored is invalid */
    i = 3;
    sysp = (_SYSPTR) i;  /* address stored is invalid */
    opnp = &i;           /* address of i stored in open pointer */
    i = (int) opnp;      /* offset of space pointer contained
                          /* in open pointer is stored in i */

    i = 4;
    opnp = (_OPENPTR) i; /* address stored is invalid */
    i = (int) opnp;      /* i is set to integer value stored (4)*/
}

```

Figure 177. ILE C Source to Show IBM i pointer casting

Example: Passing IBM i pointers as Arguments on a Dynamic Program Call to Another ILE C Program

The following example demonstrates how to pass IBM i pointers as arguments on a dynamic program (OS-linkage) call to another ILE C program.

The example consists of two ILE C programs. Program T1520DL8 passes several types of IBM i pointers as arguments to Program T1520DL9. Program T1520DL9 receives the arguments and checks them to make sure that they were passed correctly.

Instructions

1. To create the MYLIB library, enter:

```
ADDLIB LIB(MYLIB)
```

2. To create the program T1520DL8 using the source shown in [Figure 178 on page 282](#), enter:

```
CRTBND CPGM(MYLIB/T1520DL8) SRCFILE(QCPPL/QACSRC)
```

3. To create the program T1520DL9 using the source shown in [Figure 179 on page 283](#), enter:

```
CRTBND CPGM(MYLIB/T1520DL9) SRCFILE(QCPPL/QACSRC)
```

4. To run the program T1520DL8, enter:

```
CALL PGM(MYLIB/T1520DL8)
```

The output is as follows:

```
Pointers passed correctly.
Press ENTER to end terminal session.
```

Source Code Samples

```
/* This program passes several types of pointers as arguments */
/* to another ILE C program T1520DL9. */
#include <stdio.h>
#include <pointer.h>
#pragma(p128)
typedef struct {
    _SPCPTR spcptr; /* A space pointer. */
    _SYSPTR sysptr; /* A system pointer. */
    void (*fnptr)(); /* A function pointer. */
} PtrStructure;
#pragma linkage (T1520DL9, OS)

#pragma datamodel(pop)
void T1520DL9 (PtrStructure *, _SPCPTR, _SYSPTR, void (*)());
void function1(void) /* A function definition. */
{
    printf("Hello!\n");
}

int main(void)
{
    int i = 4;
    PtrStructure ptr_struct;
    /* Make assignments to the fields of ptr_struct. */
    ptr_struct.spcptr = (_SPCPTR)&i; /* A space pointer. */
    ptr_struct.sysptr = (_SYSPTR)T1520DL9; /* A system pointer. */
    ptr_struct.fnptr = &function1; /* A function pointer. */

    /* Call T1520DL9, passing the address of ptr_struct and other */
    /* valid pointer arguments. */
    T1520DL9(&ptr_struct, (_SPCPTR)&i, (_SYSPTR)T1520DL9, &function1);
}
```

Figure 178. T1520DL8 – ILE C Source that Uses IBM i pointers

```

/* This program receives the arguments from T1520DL8 and checks them */
/* to make sure they were passed correctly. */
#include <stdio.h>
#include <pointer.h>
typedef struct {
    _SPCPTR spcptr; /* A space pointer. */
    _SYSPTR sysptr; /* A system pointer. */
    void (*fnptr)(); /* A function pointer. */
} PtrStructure;

int main( int argc, char **argv)
{
    _OPENPTR openptr; /* An open pointer. */
    _SPCPTR spcptr; /* A space pointer. */
    _SYSPTR sysptr; /* A system pointer. */
    void (*fnptr)(); /* A function pointer. */
    PtrStructure *ptr_struct_ptr;
    int error_count = 0;
    /* Receive the structure pointer passed into a local variable. */
    ptr_struct_ptr = (PtrStructure *)argv[1];

    /* Receive the pointers passed into an open pointer, */
    /* then assign them to pointers of their own type. */
    openptr = (_OPENPTR)argv[2];
    spcptr = openptr; /* A space pointer. */

    openptr = (_OPENPTR)argv[3];
    sysptr = openptr; /* A system pointer. */

    openptr = (_OPENPTR)argv[4];
    fnptr = openptr; /* A function pointer. */

    /* Test the pointers passed with the pointers in ptr_struct_ptr. */

    if (spcptr != ptr_struct_ptr->spcptr)
        ++error_count;
    if (sysptr != ptr_struct_ptr->sysptr)
        ++error_count;
    if (fnptr != ptr_struct_ptr->fnptr)
        ++error_count;

    if (error_count > 0)
        printf("Pointers not passed correctly.\n");
    else
        printf("Pointers passed correctly.\n");
    return;
}

```

Figure 179. T1520DL9 – ILE C Source that Uses IBM i pointers

Using ILE C/C++ Call Conventions

ILE calling conventions are discussed as they apply to programs compiled with the ILE C/C++ compiler. For information about calling conventions that apply to multi-language applications, see [“Working with Multi-Language Applications”](#) on page 311.

This topic describes:

- [Program and procedure calls](#)
- [Renaming programs and procedures](#)
- [Calling programs that have library qualification \(using bindable APIs\)](#)
- [Calling an ILE C++ program from ILE C](#)
- [Accessing C++ classes from ILE C](#)

Note: The terms *parameter* and *argument* are used interchangeably.

Program and Procedure Calls

In ILE, program processing occurs at the procedure level. ILE programs consist of one or more modules which consist of one or more procedures.

In ILE, you can call either a program (*PGM) or an ILE procedure. The calling program must identify whether the target of the call statement is a program or an ILE procedure.

C/C++ ILE conventions differ for calling programs and for calling ILE procedures.

ILE C and C++ modules can contain only one `main()` procedure, but can contain many subordinate procedures (functions). Certain other ILE languages allow only one procedure.

For more information about the calls to programs and procedures, see *ILE Concepts*.

Using Dynamic Program Calls

You can use dynamic program calls to call OPM, EPM or ILE programs. Unlike OPM and EPM programs, ILE programs are not limited to using dynamic program calls. ILE programs can use static procedure calls or procedure pointer calls to call other procedures.

A *dynamic program call* is a special form of procedure call; it is a call to the program entry procedure. A *program entry procedure* is the procedure designated at program creation time to receive control when a program is called. In other words, calling a program entry procedure is the same as calling another program's `main()` function.

When dynamic program calls are executed, the called program's name is resolved to an address at runtime, just before the calling program passes control to the called program for the first time.

Dynamic program calls include calls to:

- ILE programs

Note: When an ILE program is called, the program entry procedure receives the program parameters and is given initial control for the program. All procedures within the program become available for procedure calls.

- EPM programs
- OPM programs
- Non-bindable APIs

How the ILE Call Stack Is Used to Control Program Flow

The *call stack* is a list of call stack entries, in a last-in-first-out (LIFO) order. A *call stack entry* is a call to a program or procedure. There is one call stack per job.

When an ILE program is called, the program entry procedure is first added to the call stack. After the program entry procedure is called, control is given to the main entry point in the program (`main()` for C or C++) which is pushed onto the stack.

Figure 180 on page 285 shows a call stack for an program consisting of an OPM program which calls an ILE program consisting of two modules: a C++ module containing the program entry procedure and the associated user entry procedure, and a C module containing a regular procedure. The most recent entry is at the bottom of the stack.

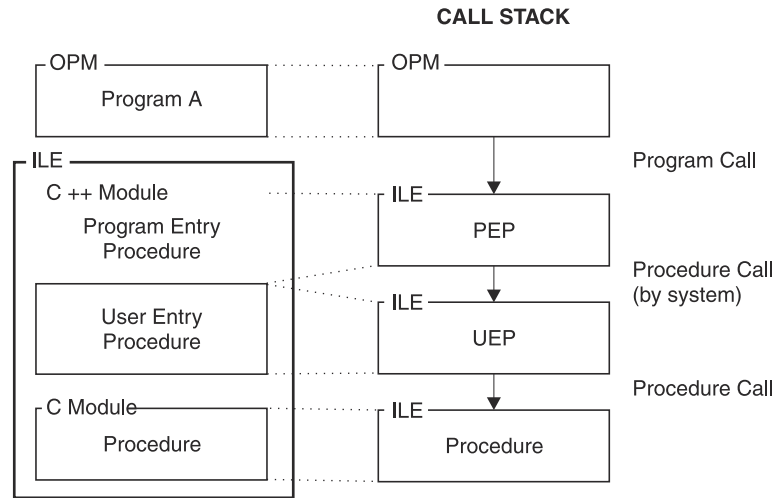


Figure 180. Program and Procedure Calls on the Call Stack

Note: In a dynamic program call, the calls to the program entry procedure and the user entry procedure (UEP) occur together, because the call to the UEP is automatic. In later diagrams involving the call stack, the two steps of a dynamic program call are combined.

For more information about the call stack, see *ILE Concepts*.

Renaming Programs and Procedures

You might want to rename a program or procedure (function) for the following reasons:

- To give an ILE procedure a more descriptive name, to make it easy to identify for maintenance purposes. For example, an ILE procedure name QRZ1233 could be renamed to checkmod.
- To change the name of a program that contains an illegal character. For example, A-B is not a valid name in a C++ program.

You can use the `#pragma map` directive to map an internal identifier to an IBM i-compliant name (10 characters or less for program names and one or more characters for ILE procedure names) in your program.

Note: For the syntax and description, see the *ILE C/C++ Compiler Reference*.

The `#pragma map` directive can appear anywhere in the source file within a single compilation unit. It can appear before any declaration or definition of the named object, function or operator. The identifiers appearing in the directive, including any type names used in the prototype argument list, are resolved as though the directive had appeared at file scope, independent of its actual point of occurrence.

As an example, see [Figure 181 on page 285](#). In this code, there are two functions named `func()`. One is a regular function, prototyped `int func(int)` and the other is a class member function `void func(void)`. To avoid confusion, they are renamed `funcname1`, and `funcname2` using the `#pragma map` directive.

```
int func(int);

class X
{
public:
    void func(void);
#pragma map(func, "funcname1") //maps ::func
#pragma map(X::func, "funcname2") //maps X::func
};
```

Figure 181. Example of Using the `#pragma map` Directive to Rename Functions

Note: Mapping can be based on parameter type as well as scope. For example, `void func(int); void func(char); #pragma map(func(int),"intFunc") #pragma map(func(char), "charFunc")`. This does not work with the *PRV option.

Calling Programs that Have Library Qualification

You can call a program with a library qualification by using bindable APIs with library qualification.

Example:

The program T2123API uses DSM ILE bindable API calls to create a window and echo whatever is entered. The *System API Reference* contains information on the ILE bindable APIs. The prototypes for the DSM APIs are in the `<qsnssess.h>` header file. The extern "OS nowiden" return type API(`arg_list`); is specified for each API where return type is void or whatever type is returned by the API, and `arg_list` is the list of parameters taken by the API. This ensures any value argument is passed by value indirectly.

```
// This program uses Dynamic Screen Manager API calls to
// create a window and echo whatever is entered. This is an
// example of bound API calls. Note the use of extern linkage
// in the <qsnssess.h> header file. OS, nowiden ensures that a
// pointer to an unwidened copy of the argument is passed to the
// API.
// Use BNDDIR(QSNAPI) on the CRTPGM command to build this
// example.

#include <stddef.h>
#include <string.h>
#include <iostream.h>
#include <qsnapi.h>

#define BOTLINE " Echo lines until:  PF3 - exit"

// DSM Session Descriptor Structure.

typedef struct{
    Qsn_Ssn_Desc_T sess_desc;
    char          buffer[300];
}storage_t;

void F3Exit(const Qsn_Ssn_T *Ssn, const Qsn_Inp_Buf_T *Buf, char *action)
{
    *action = '1';
}

int main(void)
{
    int i;
    storage_t storage;

    // Declarators for declaring windows. Types are from the <qsnssess.h>
    // header file.

    Qsn_Inp_Buf_T  input_buffer = 0;
    Q_Bin4         input_buffer_size = 50;
    char           char_buffer[100];
    Q_Bin4         char_buffer_size;

    Qsn_Ssn_T      session1;
    Qsn_Ssn_Desc_T *sess_desc = (Qsn_Ssn_Desc_T *) &storage;
    Qsn_Win_Desc_T win_desc;
    Q_Bin4         win_desc_length = sizeof(Qsn_Win_Desc_T);
    char           *botline = BOTLINE;
    Q_Bin4         botline_len = sizeof(BOTLINE) - 1;
    Q_Bin4         sess_desc_length = sizeof(Qsn_Ssn_Desc_T) +
                                     botline_len;

    Q_Bin4         bytes_read;

    // Initialize Session Descriptor DSM API.

    QsnInzSsnD( sess_desc, sess_desc_length, NULL);

    // Initialize Window Descriptor DSM API.

    QsnInzWinD( &win_desc, win_desc_length, NULL);
```


- Declare any C++ functions that you want to call as external.
- Specify the linkage convention for the call.

Specifying the Linkage Convention

Use the `#pragma linkage` and `#pragma argument` directives to specify the linkage convention. See Table 26 on page 312 and “Accessing ILE C Procedures from Any ILE Program” on page 331 for more information on using these directives.

When specifying linkage conventions, consider the following C++ characteristics:

- A C++ program uses the standard OS linkage calling convention. Use the `#pragma linkage` directive to flag the function call as an external program call.
- When you call C++ functions, you must make sure that the sender and receiver both agree on the type of parameter being passed, whether it is by pointer or by value, and whether parameters are widened. For example, if the function you are calling was declared as `extern "C" nowiden`, you must use the `#pragma argument(func, nowiden)` directive in the function declaration in ILE C.
- You can declare a C++ function as external by explicitly declaring the function within the C++ code using either `extern "C"` or `extern "C" nowiden`. You can add `#ifdef` statements to the function declarations in the header files used by both C and C++ modules, as shown in [Figure 182 on page 288](#).

```
#ifdef __cplusplus
extern "C" {
    #endif
    function declarations
    #ifdef __cplusplus
    }
    #endif
```

Figure 182. An ILE C++ Function Declared As an External Function

Note: These statements are declared with C linkage.

Example: An ILE C Program that Uses C++ Objects

This program shows how you can access the data members in C++ classes from source code written in C.

Program Structure

The program consists of these files:

- A C++ source file `hourclas.cpp` which contains:
 - Definitions of one base class, `HourMin`, and two derived classes, `HourMinSec1` and `HourMinSec2`
 - Three function prototypes with `extern "C"` linkage:
 - `extern "C" void CSetHour(HourMin *)`
 - `extern "C" void CSetSec(HourMin *)`
 - `extern "C" void CSafeSetHour(HourMin *)`
 - The definition of a function with `extern "C"` linkage, `extern "C" void CXXSetHour(HourMin * x)`
 - A `main()` function containing the program logic

See “C++ Source File `hourclas.cpp` Definitions Used by C Source File `hour.c`” on page 289.

- A C source file `hour.c` which contains:
 - A structure `CHourMin` that maps to the C++ class `HourMin` in file `hourclas.cpp`
 - Definitions of the three functions with `extern "C"` linkage declared in `hourclas.cpp`
 - The definition of a function `CSafeSetHour()`

C++ Source File hourclas.cpp Definitions Used by C Source File hour.c

```
#include <iostream.h>

class HourMin {          // base class
protected:
    int h;
    int m;

public:
    void set_hour(int hour) { h = hour % 24; } // keep it in range
    int  get_hour()         { return h; }
    void set_min(int min)  { m = min % 60; } // keep it in range
    int  get_min()         { return m; }
    HourMin(): h(0), m(0) {}
    void display() { cout << h << ':' << m << endl; }
};

// derived from class HourMin
class HourMinSec1 : public HourMin {
public:
    int s;
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int  get_sec()         { return s; }
    HourMinSec1() { s = 0; }
    void display() { cout << h << ':' << m << ':' << s << endl; }
};

// has an HourMin contained inside
class HourMinSec2 {
private:
    HourMin a;
    int s;

public:
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int  get_sec()         { return s; }
    HourMinSec2()         { s = 0; }
    void display() {
        cout << a.get_hour() << ':' << a.get_min() << ':' << s << endl; }
};

extern "C" void CSetHour(HourMin *); // defined in C
extern "C" void CSetSec(HourMin *); // defined in C
extern "C" void CSafeSetHour(HourMin *); // defined in C

// wrapper function to be called from C code */
extern "C" void CXXSetHour(HourMin * x) {
    x->set_hour(99); // much like the C version but the C++
                    // member functions provide some protection
                    // expect 99 % 24, or 3 to be the result
}

// other wrappers may be written to access other member functions
// or operators ...

main() {
    HourMin hm;
    hm.set_hour(18); // supper time;
    CSetHour(&hm); // pass address of object to C function
    hm.display(); // hour is out of range

    HourMinSec1 hms1;
    CSetSec((HourMin *) &hms1)
    hms1.display();

    HourMinSec2 hms2;
    CSetSec(&hms2);
    hms2.display();

    CSafeSetHour(&hm); // pass address to a safer C function
    hm.display(); // hour is not out of range
}
```

```

/* C code  hour.c */

struct CHourMin {
    int hour;
    int min;
};

void CSetHour(void * v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want
    p->hour = 99;             // with power comes responsibility (oops!)
}

struct CHourMinSec {
    struct CHourMin hourMin;
    int sec;
};

// handles both HourMinSec1, and HourMinSec2 classes

void CSetSec(void *v) {
    struct CHourMinSec * p;
    p = (struct CHourMinSec *) v; // force it to the type we want
    p->sec = 45;
}

void CSafeSetHour(void *v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want

    // ... do things with p, but be careful
    // ...
    // use a C++ wrapper function to access C++ function members
    CXXSetHour(p); // almost the same as p->hour = 99
}

```

Figure 183. C Source file `hour.c` that Uses Definitions from C++ Source File `hourclas.cpp`

Program Flow

As shown in “C++ Source File `hourclas.cpp` Definitions Used by C Source File `hour.c`” on page 289, the `main()` C++ function:

1. Instantiates an object `hm` of the base class `HourMin`
2. Assigns a value to the `h` variable (hour) in the base class
3. Passes the address of the base class to the function `CSetHour()` defined in the C source file `hour.c` (see [Figure 183 on page 290](#)), which assigns a new value to `h` in the base class
4. Displays the value of `h` in the base class
5. Instantiates an object `hms1` of the derived class `HourMinSec1`
6. Passes the address of this object class to the function `CSetSec()` defined in the C source file `hour.c` (see [Figure 183 on page 290](#)), which assigns a value to `s` in the object
7. Displays the value of `s` in the object
8. Instantiates an object `hms2` of the class `HourMinSec2` which contains the class `HourMin`
9. Passes the address of this new object to the function `CSetSec()` defined in the C source file `hour.c` (see [Figure 183 on page 290](#)), which assigns a value to `s` in the object
10. Displays the value of `s` in the object
11. Passes the address of the base class object to function `SafeSetHour()` defined in the C source file `hour.c` which passes the address back to a function `CXXSetHour()` defined in the C++ source file `hourclas.cpp` (see “C++ Source File `hourclas.cpp` Definitions Used by C Source File `hour.c`” on page 289)

Program Output

The program output is:

```
99:0
0 -:0 :45
0 -:0 :45
3 -:0
Press ENTER to end terminal session.
```

Accessing C++ Classes from ILE C

You can access existing C++ classes from other languages (such as ILE C), but you need to write your own functions to display and manipulate the data members of these classes.

A shared C/C++ header for class MyClass might look like the following:

```
/* myclass.h */
#ifdef __cplusplus
class MyClass {
public:
MyClass()
{
n = new int[100];
}
~MyClass()
{
delete [] n;
}
int &operator[] (int i)
{
return n[i];
}
private:
int *n;
};
#else
struct MyClass;
MyClass *createMyClass();
void destroyMyClass(MyClass*);
int *MyClassIndex(int);
#endif
```

Figure 184. Example of a Shared C/C++ Header File

Mapping a C++ Class to a C Structure

A C++ class without virtual functions can be mapped to a corresponding C structure, but there are fundamental differences between both data types. The C++ class contains data members and member functions to access and manipulate these data members. The corresponding C structure contains only the data members, but not the member functions contained in the C++ class.

Figure 185 on page 291 shows the C++ class Class1 and Figure 186 on page 292 shows the corresponding C structure.

```
class Class1
{
public:
    int m1;
    int m2;
    int m3;
    f1();
    f2();
    f3();
};
```

Figure 185. Example of C++ Class without Virtual Functions

```

struct Class1
{
    int m1;
    int m2;
    int m3;
};

```

Figure 186. Example of C Structure that Corresponds to C++ Class without Virtual Functions

To access a C++ class from a C program you need to write your own functions to inspect and manipulate the class data members directly.

Note: While data members in the C++ class can be public, protected, or private, the variables in the corresponding C structure are always publicly accessible. **You might eliminate the safeguards built into the C++ language.**

You can use C++ operators on this class if you supply your own definitions of these operators in the form of member functions.

When you write your own C++ classes that you want to access from other languages:

- Do not use static data members in your class, because they are not part of the C++ object that is passed to the other language.
- Do not use virtual functions in your class, because you cannot access the data members because the alignment of the data members between the class and the C structure is different.

Note: By making all data members of a class publicly accessible to programs written in other languages, you might be breaking data encapsulation.

Example: An ILE C Program that Uses C++ Objects

This program shows how you can access the data members in C++ classes from source code written in C.

Program Files and Structures

The program consists of these files:

- A C++ source file `hourclas.cpp` which contains:
 - Definitions of one base class, `HourMin`, and two derived classes, `HourMinSec1` and `HourMinSec2`
 - Three function prototypes with `extern "C"` linkage:
 - `extern "C" void CSetHour(HourMin *)`
 - `extern "C" void CSetSec(HourMin *)`
 - `extern "C" void CSafeSetHour(HourMin *)`
 - The definition of a function with `extern "C"` linkage, `extern "C" void CXXSetHour(HourMin * x)`
 - A `main()` function containing the program logic

Note: See [“C++ Source File `hourclas.cpp` that Contains Classes Used by C Source File `hour.c`”](#) on page 293.

- A C source file `hour.c` which contains:
 - A structure `CHourMin` that maps to the C++ class `HourMin` in file `hourclas.cpp`
 - Definitions of the three functions with `extern "C"` linkage declared in `hourclas.cpp`
 - The definition of a function `CSafeSetHour()`

Note: See [“C Source File `hour.c` that Uses C++ Classes Defined in Source File `hourclas.cpp`”](#) on page 294.

Program Description

As shown in “C++ Source File hourclas.cpp that Contains Classes Used by C Source File hour.c” on page 293, the main() function of the hourclas.cpp program:

- Instantiates an object hm of the base class HourMin
- Assigns a value to the h variable (hour) in the base class
- Passes the address of the base class to the function CSetHour() defined in the C source file hour.c, which assigns a new value to h in the base class

Note: “C Source File hour.c that Uses C++ Classes Defined in Source File hourclas.cpp” on page 294 shows the source code in hour.c.

- Displays the value of h in the base class
- Instantiates an object hms1 of the derived class HourMinSec1
- Passes the address of this object class to the function CSetSec() defined in the C source file hour.c, which assigns a value to s in the object
- Displays the value of s in the object
- Instantiates an object hms2 of the class HourMinSec2 which contains the class HourMin
- Passes the address of this new object to the function CSetSec() defined in the C source file hour.c, which assigns a value to s in the object
- Displays the value of s in the object
- Passes the address of the base class object to function SafeSetHour().

Note: The function SafeSetHour() is defined in the C source file hour() which passes the address back to a function CXXSetHour() defined in hourclas.cpp

C++ Source File hourclas.cpp that Contains Classes Used by C Source File hour.c

```
#include <iostream.h>

class HourMin {          // base class
protected:
    int h;
    int m;

public:
    void set_hour(int hour) { h = hour % 24; } // keep it in range
    int  get_hour()         { return h; }
    void set_min(int min)  { m = min % 60; } // keep it in range
    int  get_min()         { return m; }
    HourMin(): h(0), m(0) {}
    void display() { cout << h << ':' << m << endl; }
};

// derived from class HourMin
class HourMinSec1 : public HourMin {
public:
    int s;
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int  get_sec()        { return s; }
    HourMinSec1() { s = 0; }
    void display() { cout << h << ':' << m << ':' << s << endl; }
}; // has an HourMin contained inside
class HourMinSec2 {
private:
    HourMin a;
    int s;

public:
    void set_sec(int sec) { s = sec % 60; } // keep it in range
    int  get_sec()        { return s; }
    HourMinSec2()         { s = 0; }
    void display() {
        cout << a.get_hour() << ':' << a.get_min() << ':' << s << endl; }
};

extern "C" void CSetHour(HourMin *); // defined in C
```

```

extern "C" void CSetSec(HourMin *); // defined in C
extern "C" void CSafeSetHour(HourMin *); // defined in C

// wrapper function to be called from C code */
extern "C" void CXXSetHour(HourMin * x) {
    x->set_hour(99); // much like the C version but the C++
                    // member functions provide some protection
                    // expect 99 % 24, or 3 to be the result
}

// other wrappers may be written to access other member functions
// or operators ...

main() {

    HourMin hm;
    hm.set_hour(18); // supper time;
    CSetHour(&hm); // pass address of object to C function
    hm.display(); // hour is out of range

    HourMinSec1 hms1;
    CSetSec((HourMin *) &hms1)
    hms1.display();

    HourMinSec2 hms2;
    CSetSec(&hms2);
    hms2.display();

    CSafeSetHour(&hm); // pass address to a safer C function
    hm.display(); // hour is not out of range
}

```

C Source File hour.c that Uses C++ Classes Defined in Source File hourclas.cpp

```

/* C code hour.c */

struct CHourMin {
    int hour;
    int min;
};

void CSetHour(void * v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want
    p->hour = 99; // with power comes responsibility (oops!)
}

struct CHourMinSec {
    struct CHourMin hourMin;
    int sec;
};

// handles both HourMinSec1, and HourMinSec2 classes

void CSetSec(void *v) {
    struct CHourMinSec * p;
    p = (struct CHourMinSec *) v; // force it to the type we want
    p->sec = 45;
}

void CSafeSetHour(void *v) {
    struct CHourMin * p;
    p = (struct CHourMin *) v; // force it to the type we want

    // ... do things with p, but be careful
    // ...
    // use a C++ wrapper function to access C++ function members

    CXXSetHour(p); // almost the same as p->hour = 99
}

```

Program Output

The program output is:


```
99:0
0 -:0 :45
0 -:0 :45
3 -:0
Press ENTER to end terminal session.
```

Porting Programs from Another Platform to ILE

If you are developing new code, follow the ISO guidelines as much as possible and avoid platform-specific extensions. In general, your code should be portable.

This section describes:

- [Limitations to porting code from C to ILE C++](#)
- [Modifying calls of ILE C++ objects](#)
- [Using BCD macros to port coded decimal objects to ILE C++](#)
- [Header files that work with both C and C++](#)
- [Handling the stricter C++ type checking](#)
- [Declaring unsigned char pointers as unsigned char variables](#)
- [Initializing character arrays in C++](#)
- [Specifying access to string literals](#)
- [Avoiding uncaught C++ exceptions by scoping to a single activation group](#)

Limitations to Porting Code to ILE C or C++

This section describes some limitations to porting code to ILE C and C++.

File Inclusions

C++ In ILE C++:

- The include file name must be a valid workstation file name, for example "file_name" or <file_name>.
- Include files cannot reference *LIBL or *CURLIB values.

C You can use these values in ILE C include names. For example, ("*LIBL/ABC", *LIBL/ABC/*All) "...).

Platform-Specific Extensions

Platform specific extensions, for example, _Far16 and _Pasca116 are platform-specific pointers that are not portable.

Members of a Union

C++ Because an object of a class with a constructor cannot be a member of a union, the _DecimalT class template in ILE C++ cannot be used as a member of a union.

Members of a Structure

C++ In ILE C++, if a _DecimalT class template is a member of a struct, that struct cannot be initialized with an initializer list.

C The structure in ILE C is shown in the following figure:

```
typedef struct {
    char s1;
    decimal(5,3) s2;
}s_type;

s_type s ={'+', 12.345d};
```

Figure 187. Example of ILE C Structure Definition that Cannot Be Ported to ILE C++

► **C++** In ILE C++ you need to rewrite the code as shown in the following figure:

```
struct s_type {
    char s1;
    decimal(5,3) s2;
    s_type (char c, decimal(5,3) d) : s1(c), s2(d) {}
};
s_type s ('+', __D("12.345")) ;
```

Decimal Constants

► **C++** The decimal constant defined using the suffix D or d is not supported by the C++ `_DecimalT` class template. Instead, a string literal embraced by `__D` is used to represent a packed decimal constant. The decimal constant `123.456D` defined in ILE C is equivalent to `__D("123.456")` in ILE C++.

Decimal Constants and Case Statements

The `__D` macro is used to simplify code that requires the frequent use of the `_ConvertDecimal` constructor. Because the `__D` macro is equivalent to the `_ConvertDecimal` constructor, the `__D` macro cannot be used with a case statement. A valid case statement uses an integral constant expression. This code shown in the following figure results in a compiler error:

```
decimal(4,3) op;

switch int(op) {
    case int(__D("1.3")):
        .....
        break;
}
```

Figure 188. Example of Code with Decimal Constants and a Case Statement that Are Incompatible

Note: The compiler flags the case statement indicating that the case expression is not an integral constant expression.

Library QSYS.LIB under IFS

► **C++** The integrated file system provides a common interface to store and operate on information in stream files. The C stream I/O functions and the C++ stream I/O classes are implemented through the integrated file system. There are seven file systems in the integrated file system. The library (QSYS.LIB) file system maps to the IBM i file system but using this system under the integrated file system presents some limitations:

- Logical files are not supported
- The only types of physical files supported are program-described files containing a single field and source physical files containing a single text field
- Byte-range locking is not supported (*System API Reference*)
- If any job has a database file member open, only one job is given write access to that file at any time; other jobs are allowed read access only

Stream I/O for programs compiled with the ILE C++ compiler defaults to using the integrated file system. The ILE C compiler defaults to C stream I/O. If you have programs that use database or DDM files, your best choice is to use the `SYSIFCOPT(*NOIFSIO)` compiler option. This ensures that you compile your

existing programs using the IBM i file system and not the integrated file system. Compiling programs that use restricted database or DDM files under the integrated file system results in a runtime error.

Teraspace Considerations

See [“Binary Compatibility Considerations When Porting Code in a Teraspace Environment”](#) on page 397.

Modifying Calls of ILE C++ Objects

ILE C source code that calls C++ objects must be modified to run under ILE C++.

For example:

- The `extern` linkage specification with the function definition or declaration must be used instead of the `#pragma linkage` or `#pragma argument` directives.
- The `#pragma map` directive has some semantic differences.
- There is a difference in the way the `#pragma argument` directive and the `extern` linkage specification handle function definitions. Both generate the same code when processing a function call but the `#pragma argument` directive does not affect parameters within the function definition. The `extern` linkage specification does affect parameters within the function definition.

Differences in Header Files

C In ILE C, the header file `<decimal.h>` must be included in the source prior any usage of the packed decimal data type.

C++ In ILE C++ `<bcd.h>` must be included instead.

Differences in Linkage Specification

The figures in this section illustrate differences in linkage specification in ILE C and ILE C++. The same function is performed in the source code found in [Figure 189 on page 297](#), [Figure 190 on page 297](#), and [Figure 191 on page 298](#).

```
Module1.c
extern void foo (int *i, char **s)
{
    *s = *i ? "Not Zero" : "Zero";
}
```

Figure 189. Example of ILE C Source Code Using the `extern` Linkage Specification

```
C
Module2.c
extern void foo (int, char *);
#pragma argument (foo, VREF)
int main()
{
    char *s;
    foo (1, s);
}
```

Figure 190. Example of ILE C Source Code Using the `#pragma argument` Linkage Specification

C++

```

Module3.C

extern "VREF" void foo (int i, char *s)
{
    s = i ? "Not Zero" : "Zero";
}

int main()
{
    char *s;
    foo (1, s);
}

```

Figure 191. Example of ILE C++ Source Code Using the extern Linkage Specification

Differences in Function Definitions

C++ This code shows how extern "OS" with a function definition is used to replace the #pragma linkage directive. See ["Using ILE C/C++ Call Conventions" on page 283](#) for additional information.

Do not use this:	Use this:
<pre> #pragma datamodel (p128) typedef void (FUNC)(int); #pragma linkage (FUNC, OS) #pragma datamodel(pop) </pre>	<pre> extern "OS" typedef void (FUNC) (int); </pre>
<pre> typedef void (FUNC)(int) extern "OS" FUNC; //error </pre>	

Using BCD Macros to Port Coded Decimal Objects to ILE C++

The Binary Coded Decimal Class Library for IBM i is provided so that you can create binary coded decimal objects that are compatible with the ILE **packed decimal** data types.

The macros shown in the following figure are used by the `_DecimalT` class template to maintain compatibility with ILE C:

```

#define decimal      _Decimal
#define digitsof    __digitsof
#define precisionof __precisionof
#define _Decimal(n,p) _DecimalT<n,p>
#define __digitsof(DecName) (DecName).DigitsOf()
#define __precisionof(DecName) (DecName).PrecisionOf()

```

Figure 192. BCD Macros that Port Code from ILE C to ILE C++

Examples

The following figures show the source to code a packed decimal data type in ILE C and ILE C++:

```

// ILE C program
#include <decimal.h>

void main()
{
    int dig, prec;
    decimal(9,3) d93;
    dig = digitsof(d93);
    prec = precisionof(d93);
}

```

Figure 193. ILE C Source Code to Port Code to a Packed Decimal Data Type

```

// C++ program
#include <bcd.h>

void main()
{
    int dig, prec;
    _DecimalT<9,3> d93;
    dig = d93.DigitsOf();
    prec = d93.PrecisionOf();
}

```

Figure 194. ILE C++ Source Code to Port Code a Packed Decimal Data Type

The following figure shows you that by using the macros defined in <bcd.h>, you can use the same ILE C shown in the first program:

```

// C++ program using the macro
#include <decimal.h>

void main()
{
    int dig, prec;
    decimal(9,3) d93;
    dig = digitsof(d93);
    prec = precisionof(d93);
}

```

Figure 195. Example of Using BCD Macros to Port Code to ILE C++

Note: The <decimal.h> header file is specified because <decimal.h> includes the <bcd.h> header file.

Mapping Class Template Instantiations to ILE C Syntax

To map the class template instantiation to the desired ILE C syntax, C++ uses the macros shown in the following figure:

```

#define decimal      _Decimal
#define _Decimal(n,p) _DecimalT<n,p>

```

Figure 196. BCD Macros that Map C++ Class Template Instantiations to ILE C Syntax

Note: ILE C code using the `decimal(n,p)` specifier can be ported to C++ without any modification.

The `_DecimalT<n,p>` specifier supported by ILE C is not supported by the C++ compiler. To use the `_DecimalT<n,p>` specifier, you need to insert a zero explicitly at the type specifier. For example, you must change `decimal(10)` to `decimal(10,0)`.

Handling Extra Precision for Multiplication and Division

The `_DecimalT` class template allows a maximum of 62 and 93 digits as the internal results for the multiplication and division operations respectively. This is different from the ILE C packed decimal data type in which a maximum of 31 digits is used for both operations.

Note: This internal result is different from the intermediate result. The internal result is designated to store the temporary result during the operation. After the operation is completed, the internal result is converted to the intermediate result and returned to the caller.

Determining the Number of Digits in an Object

C In ILE C, when you use the `__digitsof` operator with a packed decimal data type the result is an integer constant. The `__digitsof` operator can be applied to a packed decimal data type or a packed decimal constant expression. The `__digitsof` operator returns the total number of digits *n* in a packed decimal data type.

To determine the number of digits n in a packed decimal data type, follow the example shown in [Figure 197 on page 300](#):

```
#include <decimal.h>

int n,n1;
decimal (5, 2) x;

n = __digitsof(x);          /* the result is n=5 */
n1 = __digitsof(1234.567d); /* the result is n1=7 */
```

Figure 197. Example of Code that Determines the Number of Digits in a Packed Decimal Data Type.

C++ In ILE C++, when you use the member function `DigitsOf()` with a `_DecimalT` class template the result is an integer constant. The member function `DigitsOf()` can be applied to a `_DecimalT` class template object. The member function `DigitsOf()` returns the total number of digits n in a `_DecimalT` class template object.

To determine the number of digits n in a `_DecimalT` class template object, follow the example shown in [Figure 198 on page 300](#):

```
#include <bcd.h>

int n,n1;
_DecimalT <5, 2> x;

n = x.DigitsOf();          // the result is n=5
```

Figure 198. Example of Code that Determines the Number of Digits in a _DecimalT Class Template Object

Determining the Number of Digits in an Internal Packed Decimal Data Object

C In ILE C, when you use the `__precisionof` operator with a packed decimal data type the result is an integer constant. The `__precisionof` operator can be applied to a packed decimal data type or a packed decimal constant expression. The `__precisionof` operator tells you the number of decimal digits p of the packed decimal data type.

To determine the number of decimal digits p of the packed decimal data, follow the example shown in [Figure 199 on page 300](#):

```
#include <decimal.h>

int p,p1;
decimal (5, 2) x;

p=__precisionof(x);          /* The result is p=2 */
p1=__precisionof(123.456d); /* The result is p1=3 */
```

Figure 199. Example of Code that Determines the Number of Decimal Digits in an Internal Packed Decimal Data Object

C++ In ILE C++, when you use the member function `PrecisionOf()` with a `_DecimalT` class template the result is an integer constant. The member function `PrecisionOf()` can be applied to a `_DecimalT` class template object. The member function `PrecisionOf()` tells you the number of decimal digits p of the `_DecimalT` class template object.

To determine the number of decimal digits p of the `_DecimalT` class template object, follow the example shown in :

```
#include <bcd.h>

int p,p1;
_DecimalT <5, 2> x;

p=x.PrecisionOf();           // The result is p=2
```

Figure 200. Example of Code that Determines the Number of Decimal Digits in an Internal `_DecimalT` Class Object

Formatting the Value of a Formatted C Input or Output Function

The behavior of the `fprintf()`, `sprintf()`, `vfprintf()`, `vprintf()` and `vsprintf()` functions is the same as the `printf()` function. The behavior of the `fscanf()` and `sscanf()` functions is the same as the `scanf()` function.

To control the format of the output use the *flags*, *width* and *precision* fields of the `printf()` function.

To control the format of the `scanf()` function use the *** and *width* fields of the `scanf()` function. See *ILE C/C++ Runtime Library Functions* for information on these fields.

Print Function Flags

Table 22 on page 301 describes the flag characters and their meanings for `D(n, p)` conversions.

Flag	Meaning
#	The result always contains a decimal-point character, even if no digits follow it.
0	Leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed.
-	The result is always left-justified within the field.
+	The result always begins with a plus or minus sign.
space	The result is always prefixed with a space where the result of a signed conversion is no sign or the signed conversion results in no characters.

Print Function Field Width

The optional minimum field width for the `printf()` function indicates the minimum number of digits to appear in the integral part, fractional part or both parts of a `_DecimalT` class template object. If there are fewer characters than the field width, then the field is padded with spaces. The field width can be an ***. If *n* is an ***, the value of *n* is derived from the corresponding position in the parameter list.

Print Function Field Precision

The optional precision for the `printf()` function indicates the number of digits to appear in the fractional part of a `_DecimalT` class template object. The default precision is *p*. If precision is greater than *p*, extra zeros are padded. If precision is less than *p*, rounding is performed. The field precision can be an ***. If *p* is an ***, the value of *p* is derived from the corresponding position in the parameter list.

Conversion Specifiers

The conversion specifier for the `printf()` function is:

D(n, p)

The `_DecimalT` class template object is converted in the style [-] ddd.ddd where the number of digits after the decimal-point character is equal to the precision of the specification. If the precision is missing, it is taken as 0; if the precision is zero and the *#flag* is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is truncated to the appropriate number of digits. The `(n, p)` descriptor is used to describe the

characteristic of the `_DecimalT` class template object. Both `n` and `p` have to be in the form of decimal integers. If `p` is missing, a default value of zero is assumed. If the specifier is in another form not stated above, the behavior is undefined.

If `n` and `p` of the variable to be printed do not match with the `n` and `p` in the conversion specifier `%D(n,p)`, the behavior is undefined. Use the unary operators `__digitsof(expression)` and `__precisionof(expression)` in the argument list to replace the `*` in `D(*,*)` whenever the size of the resulting class of a `_DecimalT` class template object expression is not known.

The conversion specifier for the `scanf()` function is as follows:

D(n,p)

Matches a decimal number, the expected form of the subject sequence is an optional plus or minus sign, then a non-empty sequence of digits optionally containing a decimal point. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists entirely of white space, or if the first non-white-space character is other than a sign, a digit, or a decimal-point character.

Porting Conditional Operators to ILE C or C++

C In ILE C, if both the second and third operands have an arithmetic type, the usual arithmetic conversions are performed to bring them to a common type.

C++ In C++, both the second and third expressions must be of the same class. If either expression has a different class, then you must cast the second or third expression so that it has the same class.

The following figure illustrates an example where the conditional expression fails because the second and third expressions are not of the same class.

```
#include <bcd.h>

main()
{
  int var_1;
  decimal(4,2) op_1_1 = __D("12.34");
  decimal(10,2) op_1_2 = __D("123.45");
  var_1 = (op_1_1 < op_1_2) ? (op_1_1 + 3) : op_1_2;
}
```

Figure 201. Example of a Conditional Expression that Fails because of Class Differences

To use the conditional operator with the `_DecimalT` class template you can do either of the following:

- Use an explicit cast on the second expression so that it has the same type as the third expression
- Use the same type of variables

Example of an Explicit Cast that Resolves Class Differences between Expression

The following figure shows an explicit cast on the second expression so that it has the same class as the third expression:

```
#include <bcd.h>

main()
{
  int var_1;
  decimal(4,2) op_1_1 = __D("12.34");
  decimal(10,2) op_1_2 = __D("123.45");
  var_1 = (op_1_1 < op_1_2) ? (_DecimalT<10,2>).__D(op_1_1 + 3) : op_1_2;
}
```

Figure 202. Example of an Explicit Cast that Resolves Class Differences between Expressions

Example of Use of a Consistent Variable Type

The following figure shows how to use the same type of variables:


```

#include <bcd.h>

main()
{
int var_1;
decimal(10,2) op_1_1 = __D("12.34");
decimal(10,2) op_1_2 = __D("123.45");
var_1 = (op_1_1 < op_1_2) ? op_1_1 : op_1_2;
}

```

Figure 203. Example of Use of a Consistent Variable Type

Note: The + 3 was removed from the second expression because (op_1_1 + 3) results in `_Decimal<13,2>`.

Porting ILE C Packed Decimal Data Types to the `_DecimalT` Class Template

In the class template `_DecimalT`, neither the constructor nor the assignment operator are overloaded to take any of the class template instantiations from `_DecimalT`. For this reason, explicit type casting that involves conversion from one `_DecimalT` class template object to another cannot be done directly. Instead, the macro `__D` must be used to embrace the expression that requires explicit type casting. This program in the following figure is written in ILE C:

```

#include <decimal.h>

void main ()
{
decimal(4,0) d40 = 123D;
decimal(6,0) d60 = d40;
d60 = d40;
decimal(8,0) d80 = (decimal(7,0))1;
decimal(9,0) d90;
d60 = (decimal(7,0))12D;
d60 = (decimal(4,0))d80;
d60 = (decimal(4,0))(d80 + 1);
d60 = (decimal(4,0))(d80 + (float)4.500);
}

```

Figure 204. ILE C Code that Uses Packed Decimal Data Types

This source needs to be rewritten in ILE C++ as shown in the following figure:

```

#include
int main ( void )
{
    _DecimalT<4,0> d40 = __D("123");           // OK
    _DecimalT<6,0> d60 = __D(d40);             // Because no constructor
                                              // exists that can convert
                                              // d40 to d60. macro __D is
                                              // needed to convert d40 into
                                              // an intermediate type first.

    _DecimalT<8,0> d80 = (_DecimalT<8,0>)1;    // OK
                                              // Type casting an int,
                                              // not a decimal(n,p)

    d60 = d40;                                 // OK. This is different from
                                              // the second statement in
                                              // which the constructor was
                                              // called. In this case, the
                                              // assignment operator is called
                                              // and the compiler converts
                                              // d40 into the intermediate
                                              // type automatically.

    _DecimalT<9,0> d90;                         // OK
    d60 = (_DecimalT<7,0>).__D("12");          // OK
    d60 = (_DecimalT<4,0>).__D(d80);
    d60 = (_DecimalT<4,0>).__D(d80 + 1);
    d60 = (_DecimalT<4,0>).__D(d80 + (float)4.500); // In these three cases,
                                              // the resultant classes
                                              // of the expressions are
                                              // _DecimalT<n,p>. macro __D
                                              // is needed to convert
                                              // them to an intermediate
                                              // type first.
}

```

Figure 205. ILE C++ Code that Uses the `_DecimalT` Class Template Instead of the C Packed Decimal Data Types

Differences in Using Packed Structures

The **_Packed** keyword tells the compiler to ignore the padding and pack the structure as much as possible.

C In ILE C, this keyword can be used in a structure definition and type definition.

C++ In ILE C++, the same keyword can be used only in a type definition.

	ILE C	ILE C/C++
<pre>typedef _Packed struct { . . }ps_t;</pre>	ok	ok
<pre>_Packed struct { . . }ps_v;</pre>	ok	error

Therefore, you must make sure the **_Packed** keyword is used only in type definitions located in the header file.

C In the ILE C/C++ compiler, the `#pragma pack` directive applies only to C programs. The ILE C `#pragma pack` directive is not compatible with the Windows `#pragma pack` directive.

Differences in Error Checking

This section describes error checking for:

- Invalid decimal format

- Mathematical operators

Invalid Decimal Format

C In ILE C, packed decimal is implemented as a native data type. This allows an error such as invalid decimal format to be detected at compile time.

C++ In C++, detection of a similar error is deferred until runtime, as shown in the following examples:

```
#define _DEBUG 1
#include <bcd.h>
int main ( void )
{
  _DecimalT<10,20> b= __D("ABC"); // Runtime exception is raised
}
```

and

```
#define _DEBUG 1
#include <bcd.h>
int main ( void )
{
  _DecimalT<33,2> a;           // Max. dig. allow is 31. Again,
                             // runtime exception is raised
}
```

Note: Some errors can be detected at compile time, for example: `n<1, (_Decimal<-33, 2>)`.

Mathematical Operators

C ILE C provides additional error checking on the sign or the digit codes of the packed decimal operand. Valid signs are hex A-F. Valid digit range is hex 0-9. If the decimal operand is in error, ILE C generates an error message. This additional checking is not present in the `_DecimalT` class template.

C++ The following code results in an error message in ILE C but not in ILE C++:

```
#include <decimal.h>
int main ( void )
{
  decimal(10,2) a, b;
  int c;
  c = a > b; // a and b are not valid packed decimals because
            // a and b are not initialized
}
```

Header Files that Work with Both C and C++

C header files are not generally usable by C++. Structures, unions and type definitions may be all right as well as variables. Care must be used in function prototypes and pragmas.

Using Dual Function Prototypes

To allow your header files to be used by ILE C and ILE C++ compilers, all functions with "OS" linkage type require dual prototypes, as shown in the following figure:

```

#ifdef __cplusplus
    extern "linkage-type" //linkage type "OS"
#else
    #pragma linkage(function_name,linkage_type)
#endif
void function_name(...);

```

Figure 206. Example of a Single Set of Dual Prototypes that Allow a Header File to be Used by Both ILE C and ILE C++

If you have a list of functions that need dual prototypes, you can use the syntax shown in the following figure:

```

#ifdef __cplusplus
    extern "linkage-type" { //linkage type "OS"
#else
    #pragma linkage(function_name1, linkage_type)
    .
    .
    #pragma linkage(function_nameN, linkage_type)
#endif
void function_name1(...);
.
.
void function_nameN(...);
#ifdef __cplusplus
}
#endif

```

Figure 207. Example of Multiple Sets of Dual Prototypes that Allow a Header File to be Used by Both ILE C and ILE C++

Permitting ILE C Programs to Access C++ Linkage Functions

 C++

Wrap your header files in the construct shown in the following figure:

```

.
.
.
#ifdef __cplusplus
    extern "C" {1
    #pragma info(none)
#else
    //only if you have #pragma
    #pragma nomargins nosequence //nomargin and #pragma checkout in the
    #pragma checkout(suspend) //header file
#endif
.
.
.
#ifdef __cplusplus2
    #pragma info(restore)
}
#else
    #pragma checkout(resume)
#endif
}

```

Figure 208. Example of Construct that Permits ILE C Programs to Access C++ Linkage Functions

Note:

1. The linkage specification `extern "C"` informs the compiler that all functions prototyped will have C linkage. C++ linkage functions cannot be called from C using the C++ internal name. See [“Using ILE C/C++ Call Conventions”](#) on page 283 for information on calling functions from other languages.
2. The macro `__ILEC400__` can replace `__cplusplus` but `__cplusplus` is preferred because it is portable to other implementations.

Including QSYSINC Header Files

C++ To use the QSYSINC header files in ILE C++ you need to use the following convention `#include <file/header.h>`. For example, to include QSYSINC/MIH/SYSEPT you can use `#include <mih/sysept.h>`.

Note: This form also works for ILE C.

Handling the Stricter C++ Type Checking

Type checking is stricter in C++ than it is in C.

This section describes how to:

- Resolve integer data type size issues by using the `#pragma enum` directive
- Resolve incompatible pointer types for various scenarios

Resolving Integer Data Type Size Issues

C++ In ILE C++, the enum size is always the size of an integer unless the `#pragma enum` directive is used.

To resolve any problem with the enum type size, use the `#pragma enum` directive, as shown in the following figure:

```
.
.#pragma enum (2)
enum { a=0xffff} A; //sizeof(A)=2;
#pragma enum ()
.
```

Figure 209. Example of `#pragma enum` Directive that Resolves Data Type Size Issues

Resolving Incompatible Pointer Types

C	C++
In ISO C, a pointer to void can be assigned to a pointer of any other type. You do not need to cast the pointer explicitly.	C++ allows void pointers to be assigned only to other void pointers. If you use C memory functions that return void pointers (such as <code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code>), each void pointer must be cast to an appropriate pointer type before the code is compiled. Note: You can use the <code>new</code> and <code>delete</code> operators instead of <code>malloc()</code> and <code>free()</code> .

Table 24. Resolving Incompatible Pointer Types (continued)

C	C++
<p>The C compiler compiles source code that uses <code>memcmp()</code> to compare a constant <code>char</code> array to a volatile <code>char</code> array.</p>	<p>When attempting to compile source code that uses <code>memcmp()</code> to compare a constant <code>char</code> array to a volatile <code>char</code> array, the C++ compiler generates an error message (for example, <code>volatile unsigned char cannot be converted to a const void pointer</code>). You cannot use a constant pointer where a volatile pointer is expected unless you cast a void pointer to the appropriate pointer type before compiling the code.</p> <p>Note: You can use the <code>new</code> and <code>delete</code> operators instead of <code>malloc()</code> and <code>free()</code>.</p>
<p>Note: See the <i>ILE C/C++ Language Reference</i> for more detailed information on compatibility.</p>	

Disabling Name Mangling to Avoid Undefined Name Errors

C++ All C++ function names are mangled to enable function overloading. You receive an undefined names error when you bind ILE C/C++ functions with mangled names, for example, `LocateSpaces__FPc`.

C In ILE C, the service program relationship is `LocateSpaces__FPc == LocateSpaces` or `LocateSpace__FPc == LocateSpace`. If you are porting ILE C code and you want to disable function name mangling, use `extern "C"` around the function name.

Resolving Type Mismatches with the C++ Function Prototype

C++
 ILE C++ requires:

- Full prototype declarations
- All declarations of a function must match the unique definition of a function
- The type defined in a pointer declaration must match the type defined in the function prototype

Note: ISO C has no such restrictions.

Example of Function Prototype Mismatch

```
#include <signal.h>
void (*sig_handler)(int); 1
typedef void (*SIG_T)(); // function pointer typedef of type void (*) () 2
extern "C" void (*signal (int, void(*) (int))) (int); // function pointer
// prototype with return
// type void(*) (int) 1
SIG_T oldsig = signal (SIGALL, sig_handler); // function pointer definition
// of type SIG_T
```

Figure 210. Example of Type Mismatch

Note:

1. Because of the type mismatch between the type defined in function pointer prototype ("void(*)()") and the type definition in the function prototype (void (*)(int)), the example generates the following error:

```
CZP0257(30) An object or reference of type "void (*)()" cannot be
initialized with an expression of type "extern "C" void (*)(int)"
```

2. The *int* parameter does not exist in the type definition SIG_T.

Handling the Function Prototype Mismatch

To handle the type mismatch, you can:

- Assign the pointer to a variable of a different type by using a cast expression
- Use the DFTCHAR compiler option when you compile the C++ source to set the default char type.

Note: For more information on these option, see the *ILE C/C++ Compiler Reference*.

Declaring unsigned char Pointers as unsigned char Variables

➤ C++

QXX functions return unsigned char pointers. ILE C allows you to assign a signed char to an unsigned char pointer. This is not valid in C++.

unsigned char pointers must be declared as unsigned char variables in the source code as shown in the following figure:

```
#include <xxcvt.h> //void(QXXITOP(unsigned char *pptr, int digits, int
//fraction, int value);

#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0;
    int value = 116;
    QXXITOP (pptr, digits, fraction, value);
}
```

Figure 211. Code that Declares an unsigned char Pointer as an unsigned char Variable

Initializing Character Arrays

➤ C++

In C++, when you initialize character arrays, a trailing '\0' (zero of type char) is appended to the string initializer. You cannot initialize a character array with more initializers than there are array elements.

➤ C

In ISO C, space for the trailing '\0' can be omitted in this type of information.

For example, the following initialization is **not valid in C++**:

```
char v[3] = "asd"; //not valid in C++, valid in ISO C
//because four elements are required
```

This initialization produces an error because there is no space for the implied trailing '\0' (zero of type char). The following initialization, for instance, **is valid in C++**:

```
char v[4] = "asd"; //valid in C++
```

Note: For more detailed information on compatibility, see the *ILE C/C++ Language Reference*.

Specifying Access to String Literals

To place strings into read/write memory, you must place the `#pragma strings` directive before any C or C++ code in a file.

If you use the `READONLY` option, you specify that the compiler may place strings into read-only memory.

If you use the `WRITEABLE` option, you specify that the compiler must place strings into writeable memory. Strings are writeable by default.

C C strings are read/write by default.

C++ C++ strings are read only by default.

Note: This pragma will override the `*STRDONLY` option on the `Create Module` or `Create Bound Program` commands.

For the syntax of the `#pragma strings` directive, see the *ILE C/C++ Compiler Reference*.

Avoiding Uncaught Exceptions by Scoping to a Single Activation Group

C++ In ILE, the effect of the `set_terminate()` function is scoped to an activation group. The following figure provides an example of how this can affect the compilation of code that is ported from a non-ILE platform.

```
// File main.c

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <terminat.h>

void a();
void my_terminate();

int main() {2
    set_terminate(my_terminate);
    try {
        a();
    }
    catch(...) cout << "failed" << endl;
}

// File term.c1

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
void my_terminate() {
    cout << "failed" << endl;
}
void a() { throw 7; }3
```

Figure 212. Example of Code Ported to ILE that Results in an Uncaught Exception

In Figure 212 on page 310:

1. Both `my_terminate()` and `a()` reside in a service program, which runs, by default, in a single activation group (for example, "B").
2. The `main()` procedure runs, by default, in another activation group (for example, "A").
3. In this scenario, the exception thrown by `a()` cannot percolate up to `main()` because the activation group in which `set_terminate` is executed is terminated before `my_terminate()` can be invoked, unless the default activation treatment is overridden.

As a result, a CEE9901 error message is sent to `main()`.

When you port code from another platform to ILE, you need to ensure that the following functions run in the same activation group:

- Your exception handler (`my_terminate()`, in this example)
- The function that throws the exception (`a()`, in this example)

Working with Multi-Language Applications

Read this section to learn ILE calling conventions and how they apply to multi-language applications. For information about calling conventions that apply to programs compiled with the ILE C/C++ compiler, see [“Using ILE C/C++ Call Conventions” on page 283](#).

You can call OPM, EPM or ILE programs using dynamic program calls. A dynamic program call is a call made to a program object (*PGM). Unlike OPM and EPM programs, ILE programs are not limited to using dynamic program calls. ILE programs can use static procedure calls or procedure pointer calls to call other procedures.

This topic describes:

- [Inter-language procedure calls](#)
- [ILE conventions for calling any program \(*PGM\)](#)
- [Calling any program from ILE C/C++](#)
- [Passing arguments from a CL program to an ILE C++ program](#)
- [Accessing ILE C procedures from any ILE program](#)
- [Using a linkage specification in an ILE C++ dynamic program call](#)

Note: The terms *parameter* and *argument* are used interchangeably.

Inter-Language Procedure Calls

ILE C allows arguments to be passed between procedures that are written in different ILE high-level languages (HLLs).

The calling function must ensure that the arguments are the size and type that are expected by the called function.

Note: For inter-language static procedure calls, operational descriptors can be used to resolve the differences in the representation of character strings, if:

- The static call is to a procedure written in a language other than C or C++.
- Values of this data type are passed as arguments.

ILE C provides a `#pragma` argument directive to simplify calls to bound procedures in languages such as ILE COBOL and ILE RPG. The `#pragma` argument directive allows arguments to be passed by mechanisms other than the standard C mechanism.

By default, ILE C procedures pass and accept arguments *by value*, which means that the value of the data object is placed directly into the argument list. Passing arguments by value allows you to widen integers and floating point values.

ILE RPG passes and accepts arguments *by reference*, which means that a pointer to the data object is placed into the argument list. Changes that are made by the called procedure to the argument are reflected in the calling procedure. Additionally, ILE RPG can pass arguments by value.

ILE COBOL passes arguments both by reference and *by content*, which means that the value of the data object is copied to a temporary location. The address of the copy, a pointer, is placed into the argument list. Additionally, ILE COBOL can pass arguments by value.

Note: EPM C and Pascal procedures or functions cannot call ILE C procedures. OPM programs cannot call any ILE procedures (including ILE C procedures).

[Table 25 on page 312](#) shows the default argument that passes methods on procedure calls.

ILE HLL	Pass Argument	Receive Argument
ILE C default	By value	By value
ILE C with #pragma argument OS	By reference	By reference
ILE C with #pragma linkage OS directive	Use OS-linkage when calling external programs.	By reference
ILE COBOL default	By reference	By reference
ILE CL	By reference	By reference
ILE RPG default	By reference	By reference

ILE Conventions for Calling Any Program (*PGM)

If you have an ILE C/C++ program calling a program (*PGM), use the OS calling convention for ILE C/C++ in your ILE C/C++ source to tell the compiler that PGMNAME is an external program, not a bound ILE procedure.

Note: The OS calling convention for ILE C is the #pragma linkage (PGMNAME, OS) directive. The OS calling convention for ILE C++ is extern "OS".

This section provides examples that illustrate dynamic program call conventions for ILE C/C++ programs that call any external program. As shown in Table 26 on page 312, ILE C uses the same convention when calling all external programs other than an EPM entry point.

Action	Program Call Convention
ILE C calling *PGM where *PGM is <ul style="list-style-type: none"> • ILE C • OPM COBOL • OPM RPG • OPM CL • OPM BASIC • OPM PL/1 • EPM C • EPM PASCAL • EPM FORTRAN; • ILE COBOL • ILE RPG • ILE CL • C++ 	#pragma linkage (PGMNAME, OS) For example, <pre style="background-color: #f0f0f0; padding: 10px;"> void PGMNAME(void); #pragma linkage (PGMNAME, OS) /* Other code */ /* Dynamic call to program PGMNAME */ PGMNAME(); </pre>

Table 26. Dynamic Program Calling Conventions (continued)

Action	Program Call Convention
ILE C calling an EPM entry point	<pre>#pragma linkage (QPXXCALL, OS) For example, #include <xxenv.h> /* The xxenv.h header file holds */ /* the prototype for QPXXCALL */ /* The #pragma linkage (QPXXCALL, OS) */ /* is in this header file. */ /* Other code. */ /* Dynamic call to program QPXXCALL. */ /* Dynamic call to EPM entry point using QPXXCALL: */ /* the name of the entry point is entname, envid */ /* names the user-controlled environment, the */ /* program and library name is given by envpgm, */ /* parm1 and parm2 are arguments passed to entname. */ QPXXCALL(entname, envid, &envpgm, parm1, parm1);</pre>

Mixing Recursive and Non-Recursive Calls

Extra care is required when one language uses recursive calls and another uses non-recursive calls. For example:

- An active ILE C++ procedure (that is, one which is on the call stack) can be called recursively (that is, before it returns control to its caller).
- An ILE COBOL procedure must be called with a non-recursive call. In other words, an ILE COBOL procedure which is on the call stack cannot be called until it returns control to its caller and is removed from the stack.

Do not use an ILE C++ procedure to call an ILE COBOL procedure that might call another, already active, ILE COBOL procedure.

Figure 213 on page 313 illustrates that such a call does not work. Assume that procedure A is an ILE C++ procedure, procedures B and C are ILE COBOL procedures, and that these procedures are in the same program. If procedure A calls procedure B, then procedure B can call neither procedure A nor B. If procedure B returns control to procedure A, and if procedure A then calls procedure C, procedure C can call procedure B but not procedure A or C.

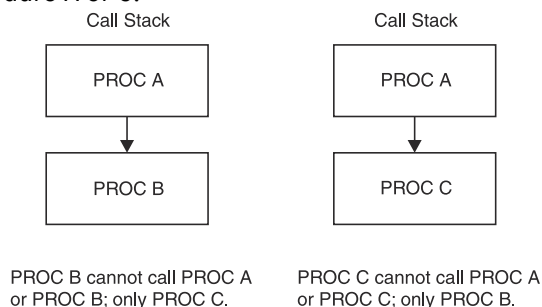


Figure 213. ILE C++ Procedures Cannot Call Active ILE COBOL Procedures

Similarly, you cannot call OPM COBOL programs that are already on the call stack.

Passing Arguments from an ILE Program to a Non-EPM Program

When passing arguments to any program other than an EPM entry point, use the following conventions:

- The program name that the ILE C/C++ program calls must be in uppercase. You can use the #pragma map directive to map an internal identifier longer than 10 characters to an object name that is compliant with IBM i (10 characters or less) in your program.

- The return code for the program call can be retrieved by declaring the program to return an integer. For example:

```
int PGMNAME(void);
#pragma linkage(PGMNAME, OS)
```

Note: The declared function in the ILE source must return either `int` or `void`. No other type is allowed.

Figure 214. Example of Using the #pragma linkage(PGMNAME, OS) Directive to Retrieve Returned Function Results

The value that is returned on the call is the return code for the program call. If the program being called is an ILE program, this return code can also be accessed using the `_LANGUAGE_RETURN_CODE` macro defined in the header file `<mlib.h>`. If the program being called is an EPM or OPM program, use the Retrieve Job Attributes (RTVJOBA) command to access this return code.

- If you use the `#pragma linkage (PGMNAME, OS)` directive, all arguments (except pointers and aggregates) are copied to temporary variables by the compiler. Pointers to the temporary variables are passed to the called program. Non-pointer arguments are passed by value-reference.

Value reference (sometimes referred to as by value, indirectly) refers to the parameter passing mechanism where:

- A non-pointer value is copied to a temporary variable, and the address of the temporary variable is passed on. The changes that are made to the variable in the called program are not reflected in the calling program.
 - If the argument you are passing is an aggregate name or a pointer, then the argument is passed directly, and a temporary variable is not created. This means that the data that is referred to by the array or pointer can be changed by the called program.
- If you want to pass arguments by reference, you must use the address of (&) operator.

When passing arguments to an EPM entry point, use the following conventions:

- If you have an ILE C program calling an EPM default entry point, then use the `#pragma linkage (PGMNAME, OS)` directive in your ILE C source to tell the compiler that PGMNAME is an external program, not a bound ILE procedure.
- If you have an ILE C program calling an EPM non-default entry point, you must use the EPM API QPXXCALL. QPXXCALL can also be used to call EPM default entry points. Because QPXXCALL is an OPM program, you must use the `#pragma linkage (QPXXCALL, OS)` directive in your ILE C source.

Passing Arguments from an ILE Program to an EPM Program

Typically, an ILE program does not receive any function checks from an EPM program because the ILE program does not monitor for an explicit signal from the `raise()` function. The EPM environment generates a diagnostic message as a result of the *ESCAPE message generated by the `raise()` function.

When an ILE program calls an EPM C program which implicitly raises a signal (as a result of an *ESCAPE message), the ILE program can monitor for, and handle, the implicit signal raised by the *ESCAPE message.

Using a Linkage Specification in a C++ Dynamic Program Call

You can call OPM, ILE, or EPM programs from a C++ program. OPM, ILE or EPM programs can also call a C++ program.

 C++

C++ provides a linkage specification to enable dynamic program calls and sharing of data between them. For a syntax diagram and additional information, see the *ILE C/C++ Language Reference*.

Valid String Literals

The *"string-literal"* is used to specify the linkage associated with a particular function. The string literals used in linkage specifications are not case-sensitive. The valid string literals for the linkage specification to call programs are:

"OS"

OS linkage call

"OS nowiden"

OS linkage call without widened parameters. See ["Specifying that a Function Has External \(OS\) Linkage"](#) on page 317 for details.

Linkage Specification

C++

If you want a C++ program to call an ILE, OPM, or EPM program (*PGM), use the `extern "OS"` linkage specification in your C++ source to tell the compiler that the called program is an external program, not a bound ILE procedure. For example, if you want a C++ program to call an OPM COBOL program (*PGM) this `extern "OS"` linkage specification in your C++ source tells the compiler that `COBOL_PGM` is an external program, not a bound ILE procedure.

```
extern "OS" void COBOL_PGM(void);
```

If you want an ILE, OPM or EPM program to call a C++ program, use the ILE, OPM, or EPM language-specific call statement.

Calling Any ILE Program from ILE C/C++

Passing Parameters from ILE C++ to a Different High-Level Language

To share data between programs (or between procedures), you need to pass parameters which both programs can use to the called program or procedure. In C++, you use the linkage specification to tell the compiler which parameter passing convention to use on the external call.

C++

When passing parameters from C++ to a different high-level language (HLL) consider:

- Parameter passing style of the HLLs

Each HLL has its own way of passing parameters. Parameters can be passed as:

- A pointer to the value
- A pointer to a copy of the value
- The value itself

C++ passes parameters in all three ways. For information on these styles, see ["Using Default Parameter Passing Styles"](#) on page 319.

- Interlanguage data compatibility

Different HLLs support different ways of representing data. Pass only parameters which have a data type common to the calling and called program or procedure. If you are not sure of the exact format of the data that is passed to your program, you may specify to the users of your procedure that an operational descriptor can be passed to provide additional information regarding the format of the passed parameters. For more information, see and ["Using Operational Descriptors to Pass Parameters of Unknown Data Type"](#) on page 320.

Using Different Linkage Specifications

Table 27 on page 316 shows the effect of using different linkage types when passing parameters:

Table 27. Effects of Various Linkage Specifications

Linkage	Name Mangled	Parameter Passing	Parameter Widening	Comments
"C++"	Yes	C++	No	This is the default.
"C"	No	C++	Yes	Used to call a function (procedure) written in ILE C.
"C nowiden"	No	C++	No	Used to call a function (procedure) written in ILE C.
"OS"	No	OS	Yes	Used to call an external program written in any OPM/EPM/ILE language.
"OS nowiden"	No	OS	No	Used to call an external program written in any OPM/EPM/ILE language.
"RPG"	No	OS	No	Used to call a procedure written in ILE RPG. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.
"COBOL"	No	OS	No	Used to call a procedure written in ILE COBOL. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.
"CL"	No	OS	No	Used to call a procedure written in ILE CL. Parameters with address types (pointer or reference) are passed by value directly. All other parameters are passed by value indirectly.
"ILE"	No	OS	Yes	Used to call a procedure written in an ILE language. Identical to RPG, COBOL, and CL specifications. If the particular language in which the function was written is unknown to the programmer, use this linkage. If you have C code that uses the <code>#pragma argument</code> directive and you plan to port this code to C++ then use the <code>extern "ILE"</code> linkage specification.
"ILE nowiden"	No	OS	No	Used to call a procedure written in an ILE language.
"VREF"	No	OS	Yes	Used to call a procedure written in an ILE language. Parameters are passed by value indirectly.
"VREF nowiden"	No	OS	No	Used to call a procedure written in an ILE language. Parameters are passed by value indirectly.

Specifying that a Function Has ILE Linkage

The `extern` keyword followed by the string literals "RPG", "COBOL", or "CL" are used to specify that the function has "ILE" linkage. These string literals perform the same function as the `#pragma argument`

directive in ILE C. The "VREF" linkage also performs the same as the *VREF* parameter on the `#pragma` argument directive.

Note: For more information on the `#pragma` argument directive, see the *ILE C/C++ Compiler Reference*.

Specifying ILE, CL, COBOL, and RPG Linkage

Specifying an ILE, CL, COBOL, or RPG linkage for a function tells the compiler:

- Arguments passed as values or non-pointer arguments are copied to temporary variables and the addresses of the temporary variables are passed to the called procedure.
- Pointer arguments are passed directly to the called procedure.
- If `extern "ILE nowiden"` is used, then the parameters and return value passed between programs and procedures are not widened; specifying any other ILE linkage widens the parameters.
- Function names are not mangled.

Specifying VREF Linkage

Specifying a VREF linkage is identical to specifying an ILE linkage except that pointer parameters are stored in a temporary variable and the address of the temporary variable is passed as the actual argument.

Specifying that a Function Has External (OS) Linkage

The `extern` specifier followed by the *string-literal* "OS" or the *string-literal* "OS nowiden" is used to declare external programs. These programs may then be called in the same way as a regular function.

When an OS linkage function is called from a C++ program, the compiler generates code and performs the following tasks in sequence:

1. If `extern "OS"` is used, then the parameters and return value are widened.
If `extern "OS nowiden"` is used, then the parameters and return value passed between programs are not widened.
2. Parameters that are passed by value are copied to temporary variables and the addresses of the temporary variables are passed to the called program.
If a temporary variable is created for a structure, the temporary variable has the same structure and information as the `struct` parameter passed.
3. Parameters that were passed by reference are still passed by reference.
4. Arrays and pointers are passed by reference.
5. If the argument you are passing is an array name or a pointer, then the argument is passed directly, and a temporary variable is not created. The data referenced by the array or pointer can be changed by the called program.
6. The function name is not mangled.

The program name that the C++ dynamic program calls must be in uppercase. You can use the `#pragma` `map` directive to map an internal identifier longer than 10 characters to an object name that is compliant with IBM i (10 characters or less) in your program. See [“Renaming Programs and Procedures”](#) on page 285.

The return code for the dynamic program call can be retrieved by declaring the program to return an integer:

```
extern "OS" int PGMNAME(void);
```

The value returned on the call is the return code for the dynamic program call. If the program being called is a C++ program, this return code can be accessed using the `_LANGUAGE_RETURN_CODE` macro defined in the header file `<mlib.h>`. A C++ program returns four bytes in the `_LANGUAGE_RETURN_CODE`. If the program being called is an EPM or OPM program, this return code can be accessed using the IBM i Retrieve Job Attributes (RTVJOBA) command.

When a function is called from an OS linkage function pointer, the compiler generates the same code sequence it does when calling an OS linkage function.

Non-pointer arguments are passed by value reference, and changes made to the variables in the called program are not reflected in the calling C++ program.

Specifying that a Function Has C Linkage

Specifying C linkage for a function tells the compiler:

- Parameters are passed using C++ conventions
- Parameters for functions declared with `extern "C"` are widened
- The function name is not mangled

The `extern` keyword followed by the *string-literal* `"C"` or the *string-literal* `"C nowiden"` is used to specify that the function is declared to have "C" linkage instead of "C++" linkage.

Using Different Linkage Specifications (C++ Only)



When using the `extern "literal"` statement, consider that:

- The *string-literal* parameter is case-insensitive. For example, `extern "OS NOWIDEN"`, `extern "OS nowiden"`, and `extern "os NoWiden"`, although different in case are all handled in the same way.
- The name of the function must follow the naming conventions of the other language. For example, for OS linkage specifications, the program name and all IBM i objects must be uppercase. See [“Renaming Programs and Procedures”](#) on page 285.
- A type definition of a function can be declared to have linkage information. After the type definition is declared, it can be used to declare functions of a particular linkage. The type definition declaration must be enclosed by braces: `{ }`.
- A function cannot be assigned directly to a pointer to a function with a different linkage. A type cast may be used to make this assignment possible. Type casting helps you eliminate parameter mismatching problems without excess constraint. See [“Type Casting to Override a Function without Overriding Linkage”](#) on page 321.
- Functions that take function pointers as parameters may **not** be overloaded based on the linkage of the function pointers, as shown in the following code sample:

```
// Using the typedef declarations above
void foo (OS);
void foo (CPP); // undefined behavior, foo already declared
```

- Functions that are defined with non-C++ linkage specifications accept parameters using the appropriate convention for that linkage. You do not need to widen parameters, as show in the following code sample:

```
extern "C" void foo (char); // chars are widened in C
// In another compilation unit we then have
extern "C" void foo (char c) // this parameter is correctly widened
{
    // implementation of foo (char);
}
```

- Attempting to define a function with either `extern "OS"`, `extern "OS nowiden"`, or `extern "builtin"` linkage results in undefined behavior, as shown in the following code sample:

```
extern "OS" void FOOPGM (char); // declaration: OK
extern "OS" void FOOPGM (char c) // definition: undefined behavior
{
    // implementation of FOOPGM
}
```


- The declaration of a function pointer is:

```
extern "OS" {
    typedef void (*fp) (char);
}
fp F00;
```

Note: Function F00() is declared to be a function pointer of type fp.

- The widening rules for all the linkage specifications shown in [Table 27 on page 316](#) are:
 - Any data type that is smaller than int is widened to int
 - float is widened to double
 - Address and Data pointers are not widened
 - struct has the same structure and information as the struct parameter passed
- Data objects can be declared inside the extern linkage declaration:

```
extern "OS" {
    int a1;
}
extern "OS" int a2;
```

Note: Variable a1 is defined while variable a2 is only declared.

- Functions F001(), F002(), and F003() are all declared as OS linkage functions. Functions F001() and F002() are declared by using the declaration-list syntax. F003() is declared by using the simple declaration.

```
extern "OS" {
    void F001 (char, char *);
    void F002 (int, int *);
}
extern "OS" int F003 (double, double *);
```

- In C++ linkage specifications, function identifiers are mangled. In all other linkage specifications, all function identifiers are identical to the exported names unless changed by the #pragma map directive.

Note: For the syntax and description, see the *ILE C/C++ Compiler Reference*.

Using Default Parameter Passing Styles

To pass parameters between ILE C/C++ and other ILE languages, especially ILE C, ILE RPG, or ILE COBOL, you must ensure that the other procedure is set up to accept data by reference.

ILE C++ uses the same calling mechanisms for calling any ILE HLL program or procedure: extern linkage specification.

ILE C++ passes and receives parameters using three passing methods:

By value, directly

The value of the data object is placed directly into the argument list.

By value, indirectly

The value of the data object is copied to a temporary location. The address of the copy, a pointer, is placed into the argument list.

By reference

A pointer to the data object is placed into the argument list. Changes made by the called procedure to the argument are reflected in the calling procedure.

Other ILE languages may have different methods of passing data. See [Table 28 on page 319](#).

ILE HLL	Pass Argument	Receive Argument
ILE C	by value, directly or by reference	by value, directly or by reference

ILE HLL	Pass Argument	Receive Argument
ILE C++	by value, directly or by value, indirectly or by reference	by value, directly or by reference
ILE COBOL	by reference or by value, indirectly	by reference or by value, indirectly
ILE RPG	by reference	by reference
ILE CL	by reference	by reference

Table 29 on page 320 shows the common parameter passing methods for the ILE procedures.

ILE HLL	Pass Argument	Receive Argument
ILE C	by value, directly	by value, directly
ILE C++	by value, directly or by value, indirectly or by reference	by value, directly or by reference
ILE COBOL	by reference or by value, indirectly	by reference
ILE RPG	by reference	by reference
ILE CL	by reference	by reference

Using Operational Descriptors to Pass Parameters of Unknown Data Type

To pass a parameter to a procedure even though the data type is not precisely known to the called procedure you can use operational descriptors. *Operational descriptors* provide descriptive information to the called procedure regarding the form of the argument. This information allows the procedure to properly interpret the passed parameter. **Use operational descriptors only when they are expected by the called procedure.**

Note: For more information on operational descriptors, see:

- *ILE Concepts*
- *ILE C/C++ Compiler Reference*

The C++ compiler supports operational descriptors for describing null-terminated strings. A character string in C++ is defined by: `char string_name[n]`, `char * string_name`, or *string-literal*.

C++ defines a string as a contiguous sequence of characters terminated by and including the first null character. In another language, a string may be defined as consisting of a length specifier and a character sequence. When passing a string from a C++ function to a function written in another language, an operational descriptor can be provided with the argument to allow the called function to determine the length and type of the string being passed.

To use operational descriptors, you specify a `#pragma descriptor` directive in your source to identify functions whose arguments have operational descriptors. Operational descriptors are then built by the calling procedure and passed as hidden arguments to the called procedure. For the syntax, see the *ILE C/C++ Compiler Reference*.

The following examples illustrates the use of operational descriptors in ILE C/C++. They show:

- The `#pragma descriptor` for `func1()` with a `#pragma descriptor` directive for the function in a header file `oper_desc.h`. See [Figure 230 on page 333](#).
- An ILE C program that calls `func1()`. See [Figure 231 on page 333](#).
- The ILE C source code of `func1()` that contains an ILE API that is used to get information from the operational descriptor. See [Figure 215 on page 321](#).

Example: Calling a Function with Operational Descriptors

The following figure shows an ILE C program that calls `func1()`. When the function `func1()` is called, the compiler generates operational descriptors for the three arguments that are specified on the call.

```
#include "oper_desc.h"
...
main()
{
    char a[5] = {'s', 't', 'u', 'v', '\0'};
    char *c;
    c = "EFGH";
    ...
    func1(a, "ABCD", c);
}
```

Figure 215. ILE C Source to Call a Function with Operational Descriptors

Type Casting to Override a Function without Overriding Linkage

To override a function without overriding `extern "OS"` use a type cast, as shown in [Figure 216 on page 321](#).


```
extern "ILE"
{
    typedef void (*ILE) ();
}
extern "C++"
{
    typedef void (*CPP) ();
}
ILE pILE;
CPP pCPP = (CPP) pILE;
```

Figure 216. Type Cast to Override a Function without Overriding Linkage

Passing Arguments from a CL Program to an ILE C++ Program

Table 30 on page 321 shows how arguments are passed from a command line CL call to an ILE C++ program.

Command Line Argument	Argv Array	ILE C++ Arguments
	<code>argv[0]</code>	"LIB/PGMNAME"
	<code>argv[1..255]</code>	normal parameters
'123.4'	<code>argv[1]</code>	"123.4"
123.4	<code>argv[2]</code>	<code>__D("0000000123.40000")</code>
'Hi'	<code>argv[3]</code>	"Hi"
Lo	<code>argv[4]</code>	"L0"
'1'	<code>argv[5]</code>	"1"

A CL character array is not null-terminated when it is passed to another program.  A C++ program that receives such an argument from a CL program should not expect the strings to be null-terminated. You can use the QCAPEXC API to ensure that all the arguments are null-terminated.

How CL Constants Are Passed to an ILE C++ Program

Table 31 on page 322 shows how CL constants are passed from a compiled CL program to an ILE C++ program.

Table 31. CL Constants Passed from a Compiled CL Program to an ILE C++ Program

Compile CL Program Argument	Argv Array	ILE C++ Arguments
	argv[0]	"LIB/PGMNAME"
	argv[1..255]	normal parameters
'123.4'	argv[1]	"123.4"
123.4	argv[2]	__D("00000000123.40000")
'Hi'	argv[3]	"Hi"
Lo	argv[4]	"LO"
'1'	argv[5]	"1"

A command processing program (CPP) passes CL constants as defined in Table 31 on page 322. You can create your own CL command with the Create Command (CRTCMD) command and define an ILE C++ program as the command processing program.

How CL Variables Are Passed to an ILE C++ Program

Table 32 on page 322 shows how CL variables are passed from a compiled CL program to an ILE C++ program. All arguments are passed by reference from CL to C++.

Table 32. CL Variables Passed from a Compiled CL Program to an ILE C++ Program

CL Variables	C++ Arguments
DCL VAR(&v) TYPE(*CHAR) LEN(10) VALUE('123.4')	"123.4"
DCL VAR(&d) TYPE(*DEC) LEN(10 1) VALUE(123.4)	__D("00000000123.40000")
DCL VAR(&h) TYPE(*CHAR) LEN(10) VALUE('Hi')	"Hi"
DCL VAR(&i) TYPE(*CHAR) LEN(10) VALUE(Lo)	"LO"
DCL VAR(&j) TYPE(*LGL) LEN(1) VALUE('1')	"1"

CL variables and numeric literals are not passed to an ILE C++ program with null-terminated strings. Character literals and logical literals are passed as null-terminated strings but are not padded with blanks. Numeric literals such as packed decimals are passed as 15,5 (8 bytes).

CL Example: a Multi-Language ILE Application

This program shows you some typical steps in creating a program that uses several ILE programming languages.

Program Description

This program is an ILE version of the small transaction-processing program described in the [“ILE-OPM CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C++ Program”](#) on page 348.

Program Structure

The program consists of the following components:

- A CL command T2123CM3 that accepts the user input and passes it to an ILE CL program
- An ILE CL program T2123CL3 that processes the input and passes it to an ILE program
- An ILE program T2123ICB in which the main() function of a C++ module T2123ICB calls a procedure CalcAndFormat in an ILE COBOL module T2123CB2

- A service program T2123SP3, created from a C++ source file t2123icc.cpp, that exports the variable TAXRATE
- A service program T2123SP4, created from an ILE RPG module object T2123RP2, that writes an audit trail of all transactions to a file
- An externally described file T2123DD2 that receives the audit trail data

Figure 217 on page 323 shows the ILE structure.

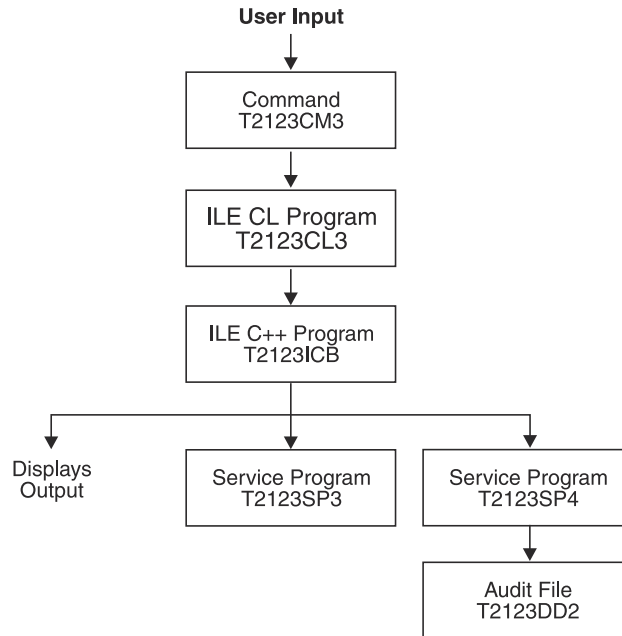


Figure 217. ILE Structure

Program Activation

The programs T2123CL3 and T2123ICB are created with the CRTPGM default for the ACTGRP parameter, ACTGRP(*NEW). When the CL program calls the ILE C++ program, a new activation group is started.

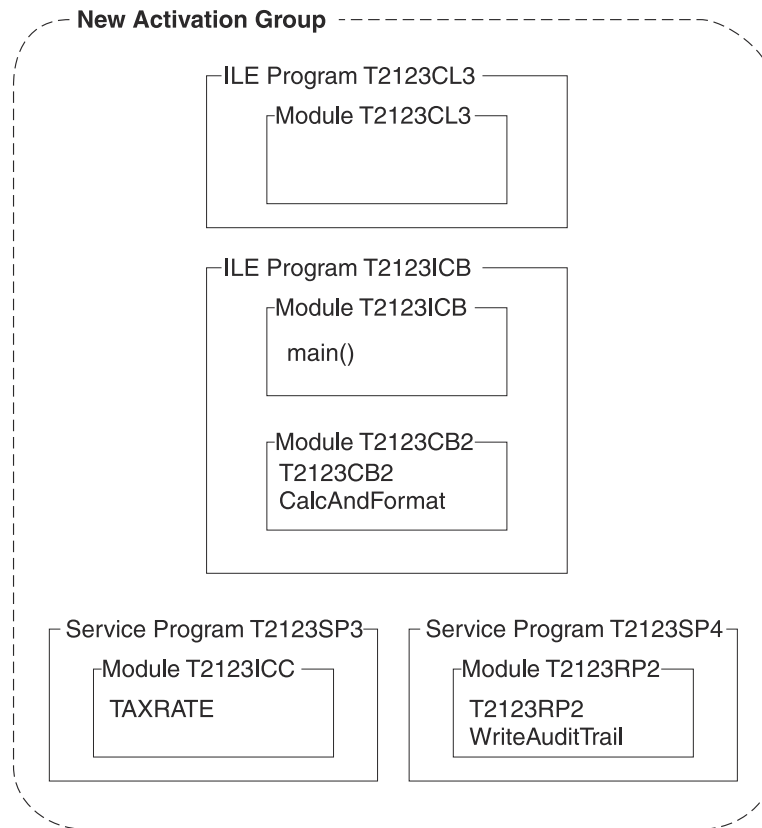


Figure 218. Basic Object Structure

The service programs are created with the CRTSRVPGM default for the *ACTGRP* parameter, *ACTGRP(*CALLER)*. When they are called, they are activated within the activation group of the calling program.

Figure 218 on page 324 shows the basic object structure used in this example.

Application Modules and Files

This program includes an externally described file, a CL program, a command prompt, two C++ source files, and ILE COBOL source file and an ILE RPG source file.

C++ Program T2123ICB.CPP

The source for the ILE C++ program T2123ICB is almost identical to the source shown in [“C++ Source File T2123IC5”](#) on page 350. The difference lies in the linkage specifications used for interlanguage calls. See [“C++ Source File T2123ICB.CPP”](#) on page 325.

C++ Source File T2123ICC

The source for the ILE C++ module T2123ICC shows the variable `TAXRATE` is exported from this module to be used by ILE COBOL and ILE RPG procedures. See [Figure 219](#) on page 326.

Note: The choice of language for `TAXRATE` is C++ because weak definitions (`EXTERNALs` from COBOL) cannot be exported out of a service program to a strong definition language like C or C++, while C or C++ can export to COBOL.

CL Program T2123CL3

The CL program T2123CL3 passes the CL variables `item_name`, `price`, `quantity`, and `user_id` by reference to an ILE C++ program T2123IC5.

CL Command Prompt T2123CM3

You use the CL command prompt T2123CM3 to prompt the user to enter item names, prices, and quantities that will be used by the C++ program T2123ICB.

ILE COBOL Module T2123CB2

The ILE COBOL procedure in T2123CB2 receives pointers to the values of the variables price, quantity and taxrate, and pointers to formatted_cost and success_flag. The CalcAndFormat() function calculates and formats the total cost. Parameters are passed from the ILE C++ program to the ILE COBOL procedure to do the tax calculation. See [Figure 220 on page 327](#).

ILE RPG Module T2123RP2

The ILE RPG module T2123RP2 contains the WriteAuditTrail() function which writes the audit trail for the program. See [Figure 221 on page 327](#).

Service Program T2123SP3

Service program T2123SP3 is created from the C++ module T2123ICC. It exports the variable TAXRATE.

Service Program T2123SP4

Service program T2123SP4 is created from the ILE RPG module T2123RP2. It exports the procedure T2123RP2.

Externally Described File T2123DD2

The file T2123DD2 contains the audit trail for the C++ program T2123ICB. The DDS source defines the fields for the audit file. See [“Externally Described File T2123DD2” on page 349](#) for the DDS source of the audit file T2123DD2.

C++ Source File T2123ICB.CPP

```
// This program demonstrates the interlanguage call capability
// of an ILE C++ program. This program is called by a CL
// program that passes an item name, price, quantity and user ID.
// A COBOL procedure is called to calculate and format total
// cost. An RPG procedure is called to write an audit trail.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// The #pragma map directive maps a function name to the bound
// procedure name so that the purpose of the procedure is clear.
// Tell the compiler that there are bound procedure calls and
// arguments are to be passed by value-reference.

extern "COBOL" void CalcAndFormat(_DecimalT <10,2>,
                                short int, char[],
                                char *);

#pragma map(CalcAndFormat,"T2123CB2")

extern "RPG" void WriteAuditTrail(char[],
                                  char[],
                                  _DecimalT<10,2>,
                                  short int, char[]);

#pragma map(WriteAuditTrail,"T2123RP2")

int main(int argc, char *argv[])
{
// Incoming arguments from a CL program have been verified by
// the *CMD and null-terminated within the CL program.
// Incoming arguments are passed by reference from a CL program.

    char          *user_id;
    char          *item_name;
    short int     quantity;
    _DecimalT<10, 2> price;
    char          formatted_cost[22];

// Remove null terminator for RPG program. Item name is null
// terminated for C++.

    char          rpg_item_name[20];
    char          null_formatted_cost[22];
    char          success_flag = 'N';
    int           i;

//Incoming arguments are all pointers.
```

```

    item_name = argv[1];
    price     = *((_DecimalT<10, 2> *) argv[2]);
    quantity  = *((short *) argv[3]);
    user_id   = argv[4];

// Call the COBOL program to do the calculation, and return a
// Y/N flag, and a formatted result.

    CalcAndFormat(price, quantity, formatted_cost, &success_flag);

    memcpy(null_formatted_cost, formatted_cost, sizeof(formatted_cost));

// Null terminate the result.

    formatted_cost[21] = '\0';
    if (success_flag == 'Y')
    {
        for (i=0; i<20; i++)
        {

// Remove null terminator for the RPG program.

            if (*(item_name+i) == '\0')
            {
                rpg_item_name[i] = ' ';
            }
            else
            {
                rpg_item_name[i] = *(item_name+i);
            }
        }

// Call an RPG program to write audit records.

        WriteAuditTrail(user_id, rpg_item_name, price, quantity,
            formatted_cost);

        cout <<"plus tax =" << quantity << item_name << null_formatted_cost
            <<endl <<endl;
        }
        else
        {
            cout <<"Calculation failed" <<endl;
        }
    }
}

```

C++ Source File T2123ICC

```

// Export the tax rate data.
#include <bcd.h>
const _DecimalT <2,2> TAXRATE = __D(".15");

```

Figure 219. C++ Source Code T2123ICC that Exports a Variable for Use by ILE COBOL and ILE RPG Procedures

Note: Weak definitions (EXTERNALS from COBOL) cannot be exported out of a service program to a strong definition language like C or C++, while C or C++ can export to COBOL. The choice of language for TAXRATE is C++.

ILE COBOL Program T2123CB2

```

IDENTIFICATION DIVISION.
PROGRAM-ID. T1520CB2 INITIAL.
*****
* parameters:
* incoming: PRICE, QUANTITY
* returns : TOTAL-COST (PRICE*QUANTITY*1.TAXRATE)
* SUCCESS-FLAG.
* TAXRATE : An imported value.
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.
DATA DIVISION.
WORKING-STORAGE SECTION.

    01 WS-TOTAL-COST          PIC S9(13)V99          COMP-3.
    01 WS-TAXRATE             PIC S9V99              COMP-3
                                         VALUE 1.
    01 TAXRATE                EXTERNAL PIC SV99       COMP-3.

LINKAGE SECTION.
    01 LS-PRICE               PIC S9(8)V9(2)         COMP-3.
    01 LS-QUANTITY            PIC S9(4)              COMP-4.
    01 LS-TOTAL-COST          PIC $$$,$$$,$$$,$$$,$$.99
                                         DISPLAY.
    01 LS-OPERATION-SUCCESSFUL PIC X                DISPLAY.

PROCEDURE DIVISION USING LS-PRICE
                    LS-QUANTITY
                    LS-TOTAL-COST
                    LS-OPERATION-SUCCESSFUL.

MAINLINE.
    MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
    PERFORM CALCULATE-COST.
    PERFORM FORMAT-COST.
    EXIT PROGRAM.

CALCULATE-COST.
    ADD TAXRATE TO WS-TAXRATE.
    COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                         LS-PRICE *
                                         WS-TAXRATE

    ON SIZE ERROR
        MOVE "N" TO LS-OPERATION-SUCCESSFUL
    END-COMPUTE.

FORMAT-COST.
    MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

Figure 220. T2123CB2

ILE RPG Module T2123RP2

```

FT1520DD2  0  A  E          DISK
D TAXRATE          S          3P 2 IMPORT
D QTYIN           DS
D QTYBIN          1          4B 0
C   *ENTRY        PLIST
C                 PARM          USER          10
C                 PARM          ITEM          20
C                 PARM          PRICE         10 2
C                 PARM          QTYIN
C                 PARM          TOTAL          21
C                 EXSR          ADDR          LR
C   ADDR          BEGSR
C                 MOVE          UDATE          DATE
C                 MOVE          QTYBIN          QTY
C                 MOVE          TAXRATE         TXRATE
C                 WRITE          T1520DD2R
C                 ENDSR

```

Figure 221. ILE RPG Module T2123RP2

Invoking the ILE Program

T2123ICB is considered the main program. It runs in the new activation group that is created when the CL program T2123CL3 is called.

To enter data for the program T2123ICB enter the command: T2123CM2 and press F4 (Prompt). You can enter the sample data in “Invoking the ILE-OPM Program” on page 353.

The output is the same as for the OPM version of this program.

The physical file T2123DD2 contains the same data as shown in the OPM version in “Invoking the ILE-OPM Program” on page 353.

Example: a User-Defined CL Program that Calls an ILE C++ Program

Figure 225 on page 329 shows how to retrieve a return value from main. A CL command called SQUARE calls an ILE C++ program SQITF. The program SQITF calls another ILE C++ program called SQ. The program SQ returns a value to program SQITF.

You use the CL command prompt SQUARE to enter the number you want to determine the square of for the ILE C++ program SQITF:

```
CMD      PROMPT('CALCULATE THE SQUARE')
PARM    KWD(VALUE) TYPE(*INT4) RSTD(*NO) RANGE(1 +
        9999) MIN(1) ALWUNPRT(*YES) PROMPT('Value' 1)
```

Note: Returning an integer value from an ILE C++ program may impact performance.

Programming Tasks

1. To create a CL command prompt SQUARE using the source shown below, type:

```
CRTCMD CMD(MYLIB/SQUARE) PGM(MYLIB/SQITF) SRCFILE(MYLIB/QCMDSRC)
```

```
CMD      PROMPT('CALCULATE THE SQUARE')
PARM    KWD(VALUE) TYPE(*INT4) RSTD(*NO) RANGE(1 +
        9999) MIN(1) ALWUNPRT(*YES) PROMPT('Value' 1)
```

Figure 222. SQUARE – CL Command Source to Receive Input Data

You use the CL command SQUARE to enter the value for the ILE C program SQITF.

2. To create a program SQIFT using the source shown below, type:

```
CRTBNDC PGM(MYLIB/SQIFT) SRCFILE(MYLIB/QCSRC)
```

```
/* This program SQITF is called by the command SQUARE. This      */
/* program then calls another ILE C program SQ to perform        */
/* calculations and return a value.                               */
#include <stdio.h>
#include <decimal.h>
#pragma linkage(SQ, OS) /* Tell compiler this is external call, */
/* do not pass by value.                                         */
int SQ(int);
int main(int argc, char *argv[])
{
    int *x;
    int result;
    x = (int *) argv[1];
    result = SQ(*x);
    /* Note that although the argument is passed by value, the compiler */
    /* copies the argument to a temporary variable, and the pointer to */
    /* the temporary variable is passed to the called program SQ.      */
    printf("The SQUARE of %d is %d\n", *x, result);
}
```

Figure 223. SQITF – ILE C Source to Pass an Argument by Value

3. To create the program SQ using the source shown below, type:

```
CRTBND C PGM(MYLIB/SQ) SRCFILE(MYLIB/QCSRC) OUTPUT(*PRINT)
```

```
/* This program is called by another ILE C program called SQITF. */
/* It performs the square calculations and returns a value to SQITF. */
#include <stdio.h>
#include <decimal.h>
int main(int argc, char *argv[])
{
    int    *vin;
    int    vout;
    vin = (int *) argv[1];
    vout = (*vin)*(*vin);
    return(vout);
}
```

Figure 224. SQ – ILE C Source to Perform Calculations and Return a Value

The program SQ calculates an integer value and returns the value to the calling program SQITF.

4. To enter data for the program SQITF, type:

```
SQUARE
```

and press F4 (Prompt).

5. Type 10, and press Enter. The output is as follows:

```
The SQUARE of 10 is 100
Press ENTER to end terminal session.
```

Source Code

```
// This program SQITF is called by the command SQUARE. This
// program then calls another ILE C++ program SQ to perform
// calculations and return a value.

#include <iostream.h>

extern "OS" int SQ(int); // Tell compiler this is external call,
                        // do not pass by value.

int main(int argc, char *argv[])
{
    int  *x;
    int  result;

    x = (int *) argv[1];

    result = SQ(*x);

    // Note that although the argument is passed by value, the compiler
    // copies the argument to a temporary variable, and the pointer to
    // the temporary variable is passed to the called program SQ.

    cout <<"The SQUARE of" <<x <<"is" <<result <<endl;
}
}
```

The ILE C++ program SQ calculates an integer value and returns the value to the calling program SQITF:

```
// This program is called by a ILE C++ program called SQITF.
// It performs the square calculations and returns a value to SQITF.

int main(int argc, char *argv[])

{ return (*(int *) argv[1]) * (*(int *) argv[1]);
}
```

Figure 225. User-Defined CL Command SQUARE that Calculates the Square of a Specified Number

Using the CL Command SQUARE to Return the Calculated Value

To enter data for the program SQITF:

1. Enter the command SQUARE and press F4 (Prompt).
2. Type 10, and press Enter.

The output is:

```
The SQUARE of 10 is 100
Press ENTER to end terminal session.
```

Example: CL Program that Passes Parameters to an ILE C++ Program

The CL program CLPROG1 passes the parameters v, d, h, i, j to an ILE C++ program MYPROG1.

Note: To pass parameters to an ILE program when you run it, use the PARM option of the CL Call (CALL) command.

The parameters are null-terminated within the the CL program CLPROG1. They are passed by reference. All incoming arguments to MYPROG1 are pointers.

```
/* CLPROG1
PGM      PARM(&V &D &H &I &J)
DCL      VAR(&V) TYPE(*CHAR) LEN(10)
DCL      VAR(&VOUT) TYPE(*CHAR) LEN(11)
DCL      VAR(&D) TYPE(*DEC) LEN(10 1)
DCL      VAR(&H) TYPE(*CHAR) LEN(10)
DCL      VAR(&HOUT) TYPE(*CHAR) LEN(11)
DCL      VAR(&I) TYPE(*CHAR) LEN(10)
DCL      VAR(&IOUT) TYPE(*CHAR) LEN(11)
DCL      VAR(&J) TYPE(*LGL) LEN(1)
DCL      VAR(&JOUT) TYPE(*LGL) LEN(2)
DCL      VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE C++ PROGRAM */
CHGVAR   VAR(&VOUT) VALUE(&V *TCAT &NULL)
CHGVAR   VAR(&HOUT) VALUE(&V *TCAT &NULL)
CHGVAR   VAR(&IOUT) VALUE(&V *TCAT &NULL)
CHGVAR   VAR(&JOUT) VALUE(&V *TCAT &NULL)
CALL     PGM(MYPROG1) PARM(&VOUT &D &HOUT &IOUT &JOUT)
ENDPGM
```

Figure 226. Example of CL Program that Passes Arguments to an ILE C++ Program

The CL program CLPROG1 receives its input values from a CL Command Prompt MYCMD1 which prompts the user to input the desired values. The source code for MYCMD1 is:

```
CMD      PROMPT('ENTER VALUES')
PARM     KWD(V) TYPE(*CHAR) LEN(10) +
         PROMPT('1ST VALUE')
PARM     KWD(D) TYPE(*DEC) LEN(10 2) +
         PROMPT('2ND VALUE')
PARM     KWD(H) TYPE(*CHAR) LEN(10) +
         PROMPT('3RD VALUE')
PARM     KWD(I) TYPE(*CHAR) LEN(1) +
         PROMPT('4TH VALUE')
PARM     KWD(J) TYPE(*LGL) LEN(10 2) +
         PROMPT('5TH VALUE')
```

Figure 227. Example of Generic CL Command Prompt

After the CL program CLPROG1 has received the user input from the command prompt MYCMD1, it passes the input values on to a C++ program MYPROG1. The source code for this program is contained in myprog1.cpp:

```
// myprog1.cpp

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// Arguments are received by reference from CL program CLPROG1
// Incoming arguments are all pointers

int main(int argc, char *argv[])
{
    char          *v;
    char          *h;
    char          *i;
    char          *j;
    _DecimalT <10, 1> d;

    v = argv[1];
    d = *((_DecimalT <10,1> *) argv[2]);
    h = argv[3];
    i = argv[4];
    j = argv[5];
    cout << " v= " << v
         << " d= " << d
         << " h= " << h
         << " i= " << i
         << " j= " << j
         << endl;
}
```

Figure 228. Example of C++ Program that Receives Arguments (Pointers) by Reference

If the CL program CLPROG1 passes the following parameters to the C++ program MYPROG1:

```
'123.4', 123.4, 'Hi', L0, and '1'
```

the output from program MYPROG1 is:

```
v= 123.4      HI          L0          1 d= 123.4 h= HI          L0          1
i= L0        1 j= 1
Press ENTER to end terminal session.
```

Accessing ILE C Procedures from Any ILE Program

ILE C programs are called by dynamic program calls, but the procedures within an activated ILE C program are accessed by means of either of the following:

- [Static procedure calls](#)
- [Procedure pointer calls](#)

Note: ILE C programs that have *not* been activated must be called dynamically.

Static Procedure Calls

A *static procedure call* can call any of the following:

- A procedure within the same module
- A procedure in a separate module within the same ILE C program or service program
- A procedure in a separate ILE C service program

Note: The term procedure in ILE is equivalent to the term function in ILE C.

Static procedure calls use fewer system resources at runtime (specifically, when the program is activated) than dynamic program calls, because static procedure calls are resolved and bound at compile time. Symbols for dynamic program calls are resolved to addresses whenever the call is performed.

Note: The term static procedure call does not refer to static storage class but refers to a bound procedure call within a bound module or service program. Operational descriptors can be used to resolve the differences in the representation of character strings, if:

- The static call is to a procedure written in a language other than C or C++
- Values of this data type are passed as arguments

Procedure Pointer Calls

Procedure pointer calls provide a way to call a procedure dynamically. For example, by manipulating arrays or tables of procedure names or addresses, you can dynamically route a procedure call to different procedures.

Called Procedures and Operational Descriptors

Operational descriptors provide descriptive information to the called procedure in cases where the called procedure cannot precisely anticipate the form of the argument, for example, different types of strings. The additional information allows the procedure to properly interpret the string. You should use operational descriptors only when they are expected by the called procedure, usually an ILE bindable API.

Example: Calling an ILE API from ILE C

The following figure shows the ILE C source code of `func1()` that contains the call to the ILE API. The API is used to determine the string type, length, and maximum length of the string arguments declared in `func1()`. The values for `typeCharZ` and `typeCharV2` are found in the ILE API header file `<leod.h>`.

```
#include <string.h>
#include <stdio.h>
#include <leawi.h>
#include <leod.h>
#include "oper_desc.h"
int func1(char a[5], char b[5], char *c)
{
    int      posn      = 1;
    int      datatype;
    int      currlen;
    int      maxlen;
    _FEEDBACK fc;
    char     *string1;
    /* Call to ILE API CEEGSI to determine string type, length */
    /* and the maximum length. */
    CEEGSI(&posn, &datatype, &currlen, &maxlen, &fc);

    switch(datatype)
    {
        case typeCharZ:
            string1 = a;
            break;
        case typeCharV2:
            string1 = a + 2;
            break;
    }
    /* Use string1. */
    if (!memcmp(string1, "stuv", currlen))
        printf("First 4 characters are the same.\n");
    else
        printf("First 4 characters are not the same.\n");
}
```

Figure 229. ILE C Source to Determine the String Arguments in a Function

Operational Descriptors and the #pragma descriptor Directive

ILE C/C++ provides a #pragma descriptor directive to identify functions whose arguments have operational descriptors. You can retrieve the information from an operational descriptor using the ILE bindable APIs Retrieve Operational Descriptor Information (CEEDOD) and Get Descriptive Information About a String Argument (CEESGI). The ILE C/C++ supports operational descriptors for string arguments.

Note: The term static procedure call does not refer to static storage class but refers to a bound procedure call within a bound module or service program. Operational descriptors can be used to resolve the differences in the representation of character strings, if:

- The static call is to a procedure written in a language other than C or C++
- Values of this data type are passed as arguments

Example: Declaring a Function that Requires Use of Operational Descriptors

The following figure shows how to declare a function that requires operational descriptors for some of its arguments.

```
int func (char *, int *, int, char *, ...); /* prototype */
#pragma descriptor (void func ("", void, void, ""))
```

Figure 230. ILE C Source that Declares a Function that Requires Operational Descriptors

A function that is named `func()` is declared. The #pragma descriptor for `func()` specifies that the ILE C compiler must generate string operational descriptors for the first and fourth arguments of `func()` whenever `func()` is called.

Example: Generating Operational Descriptors

The following figure shows that the #pragma descriptor for `func1` with a #pragma descriptor directive for the function in a header file `oper_desc.h`.

```
/* Function prototype in oper_desc.h */
int func1( char a[5], char b[5], char *c );
#pragma descriptor(void func1( "", "", "" ))
```

Figure 231. ILE C Source to Generate Operational Descriptors

A function that is named `func1()` is declared. The #pragma descriptor for `func1()` specifies that the ILE C compiler must generate string operational descriptors for the three arguments.

OPM CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C Program

This application uses session input to:

- Calculate tax
- Format output
- Write to an audit file

Note: For the ILE version of this example, see [“ILE CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C Program”](#) on page 340.

As shown in [Basic Program Structure](#), the application is a small transaction processing program that takes name, price, and quantity as input items. As output, the application displays the total cost of the items that are specified on the screen and writes an audit trail of transactions to a file.

Basic Program Structure

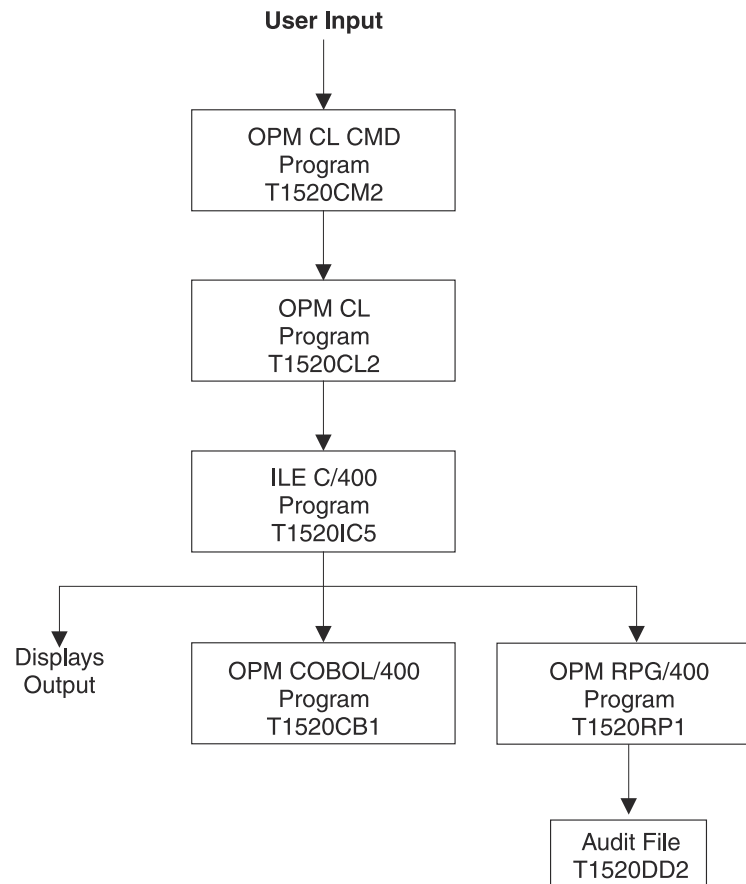


Figure 232. OPM CL Example: Basic Program Structure

As shown in [Basic Program Structure](#), this example consists of:

- A CL command (T1520CM2) that accepts the user's input and passes it to a CL program (T1520CL2).
- A CL program (T1520CL2) that processes the input and passes it to an ILE C program (T1520IC5).
- An ILE C (T1520IC5) program that calls an OPM COBOL program (T1520CB1) to process the input, and an OPM RPG program (T1520RP1) to write the audit trail to an externally described file.
- An OPM COBOL program (T1520CB1) that completes the calculation and formats the cost.
- An OPM RPG program (T1520RP1) that updates the audit file (T1520DD2) with each transaction.

Note: In addition to the source for CMD, CL, ILE C, OPM RPG, and OPM COBOL, you need DDS source for the output file. The DDS source defines the fields for the audit file.

Program Modules and Activation Groups

As shown in [Figure 233](#) on [page 335](#), the CL, COBOL, and RPG programs are activated within the user default activation groups. A new activation group is started when the CL programs call the ILE C program because the ILE C program is created with the CRTPGM default of *NEW for the ACTGRP keyword.

Note: When a CRTPGM parameter does not appear in the CRTBNDC command, the CRTPGM parameter default is used.

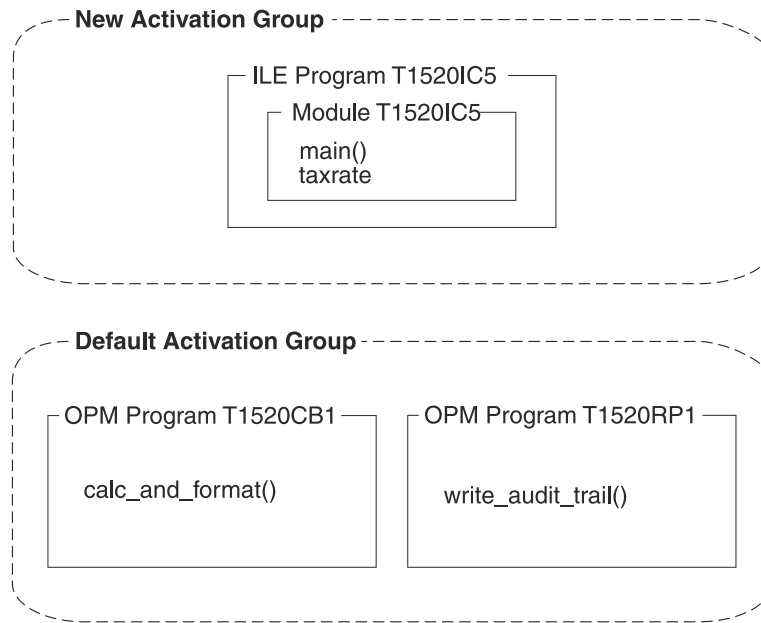


Figure 233. Structure of the Program in ILE C

These programming steps show you how to:

- Create a physical file to contain an audit trail for the ILE C program.
- Create a CL program that passes the parameters item name, price, quantity, and user ID to an ILE C program.
- Create a CL command prompt to enter data for item name, price, and quantity. The OPM CL command prompt passes the data to the CL program which in turn calls an ILE C program.
- Create one program with a main() function that receives incoming arguments from a CL program, calls an OPM COBOL program to complete the tax calculation and format the total cost. It calls an OPM RPG program to write audit records.
- Create an OPM COBOL program that completes the tax calculation and formats the total cost.
- Create an OPM RPG program that writes the audit trail for the application.

Programming Tasks

1. Create the physical file T1520DD2 using the source shown in [Figure 234 on page 335](#). On a command line, enter:

```
CRTPF FILE(MYLIB/T1520DD2) SRCFILE(QCPPLE/QADSSRC) MAXMBS(*NOMAX)
```

This source file contains the audit trail for the ILE C program T1520IC5.

```
R T1520DD2R
      USER          10          COLHDG('User')
      ITEM           20          COLHDG('Item name')
      PRICE          10S 2      COLHDG('Unit price')
      QTY            4S          COLHDG('Number of items')
      TXRATE         2S 2      COLHDG('Current tax rate')
      TOTAL          21          COLHDG('Total cost')
      DATE           6          COLHDG('Transaction date')
K USER
```

Figure 234. T1520DD2 – DDS Source for an Audit File

Note: The DDS source defines the fields for the audit file.

2. Create the CL program T1520CL2 using the source shown in [Figure 235 on page 336](#). On a command line, enter:

```

CRTCLPGM PGM(MYLIB/T1520CL2) SRCFILE(QCPPL/QACL SRC)

PGM      PARM(&ITEMIN &PRICE &QUANTITY)
        DCL      VAR(&USER) TYPE(*CHAR) LEN(10)
        DCL      VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
        DCL      VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
        DCL      VAR(&PRICE) TYPE(*DEC) LEN(10 2)
        DCL      VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
        DCL      VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
CHGVAR   VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
/* GET THE USERID FOR THE AUDIT FILE */
RTVJOBA  USER(&USER)
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF   FILE(T1520DD2) TOFILE(*LIBL/T1520DD2) +
        MBR(T1520DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL     PGM(T1520IC5) PARM(&ITEMOUT &PRICE &QUANTITY +
        &USER)
DLTOVR   FILE(*ALL)
ENDPGM

```

Figure 235. T1520CL2 – CL Source to Pass Variables to an ILE C Program

Note:

- CL variables *item name*, *price*, *quantity*, and *user ID* are passed by reference to the ILE C program T1520IC5.
 - The Retrieve Job Attributes (RTVJOBA) command obtains the user ID for the audit trail.
 - Arguments are passed by reference. They can be changed by the receiving ILE C program.
 - The variable *item_name* is null-ended in the CL program.
 - CL variables and numeric constants are not passed to an ILE C program with null-ended strings.
 - Character constants and logical literals are passed as null-ended strings, but are not widened with blanks.
 - Numeric constants such as packed decimals are passed as 15,5 (8 bytes).
 - Floating point constants are passed as double precision floating point values, for example 1.2E+15.
 - To pass parameters to an ILE program when you run it, use the PARM option of the CL Call (CALL) command.
3. Create a CL command prompt T1520CM2 to enter the item name, price, and quantity for the ILE C program T1520IC5. Use the source shown in [Figure 236 on page 336](#). On a command line, enter:

```

CRTCMD CMD(MYLIB/T1520CM2) PGM(MYLIB/T1520CL2) SRCFILE(QCPPL/QACMSRC)

CMD      PROMPT('CALCULATE TOTAL COST')
        PARM      KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
        MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
        PARM      KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
        RANGE(0.01 99999999.99) MIN(1) +
        ALWUNPRT(*YES) PROMPT('Unit price' 2)
        PARM      KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
        9999) MIN(1) ALWUNPRT(*YES) +
        PROMPT('Number of items' 3)

```

Figure 236. T1520CM2 – CL Command Source to Receive Input Data

4. Create the program T1520IC5 using the source shown. On a command line, enter:

```

CRTBNDC PGM(MYLIB/T1520IC5) SRCFILE(QCPPL/QACSRC) OUTPUT(*PRINT) FLAG(30)
        OPTION(*SHOWINC *NOLOGMSG) MSGLMT(10) CHECKOUT(*PARM) DBGVIEW(*ALL)

```

```

/* This program is called by a CL program that passes an item      */
/* name, price, quantity and user ID.                               */
/* A COBOL program is called to calculate and format the total cost.*/
/* An RPG program is called to write an audit trail.                */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>
/* The #pragma map directive maps a new program name to the existing */
/* program name so that the purpose of the program is clear.        */
#pragma map(calc_and_format,"T1520CB1")
#pragma map(write_audit_trail,"T1520RP1")
/* Tell the compiler that there are dynamic program calls so      */
/* arguments are passed by value-reference.                         */
#pragma linkage(calc_and_format, OS, nowiden)
#pragma linkage(write_audit_trail, OS)
void calc_and_format(decimal (10,2),
                    short int,
                    decimal(2,2),
                    char[],
                    char *);
void write_audit_trail(char[],
                      char[],
                      decimal(10,2),
                      short int,
                      decimal(2,2),
                      char[]);
int main(int argc, char *argv[])
{
/* Incoming arguments from a CL program have been verified by      */
/* the *CMD and null end within the CL program.                    */
/* Incoming arguments are passed by reference from a CL program.   */
char          *user_id;
char          *item_name;
short int     quantity;
decimal (10, 2) price;
decimal (2,2) taxrate = .15D;
char          formatted_cost[22];
/* Remove null end for RPG program. Item name is null ended for C. */
char          rpg_item_name[20];
char          null_formatted_cost[22];
char          success_flag = 'N';
int           i;

/* Incoming arguments are all pointers.                             */
item_name = argv[1];
price = *((decimal (10, 2) *) argv[2]);
quantity = *((short *) argv[3]);
user_id = argv[4];

/* Call the COBOL program to do the calculation, and return a     */
/* Y/N flag, and a formatted result.                                */
calc_and_format(price,
                quantity,
                taxrate,
                formatted_cost,
                &success_flag);
memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

/* Null end the result.                                           */
formatted_cost[21] = '\0';
if (success_flag == 'Y')
{
for (i=0; i<20; i++)
{
/* Remove null end for the RPG program.                            */
if (*(item_name+i) == '\0')
{
rpg_item_name[i] = ' ';
}
else
{
rpg_item_name[i] = *(item_name+i);
}
}
}

/* Call an RPG program to write audit records.                     */
write_audit_trail(user_id,

```

```

        rpg_item_name,
        price,
        quantity,
        taxrate,
        formatted_cost);

    printf("\n%d %s plus tax = %-s\n", quantity,
           item_name,
           null_formatted_cost);
}
else
{
    printf("Calculation failed\n");
}
}

```

Note:

- a. The main() function in this program receives incoming arguments from CL program T1520CL2 that are verified by CL command prompt T1520CM2 and null ended within CL program T1520CL2. All incoming arguments are pointers.
 - b. The main() function also calls calc_and_format(), which is mapped to a COBOL name. It passes by OS-linkage convention the price, quantity, taxrate, formatted cost, and a success flag.
 - c. Because the OPM COBOL program is not expecting widened parameters (the default for ILE C), nowiden is used in the #pragma linkage directive. The formatted cost and the success flag values are updated in program T1520IC5.
 - d. If calc_and_format() returns successfully a record is written to the audit trail by write_audit_trail() in the OPM RPG program. The main() function in this program (T1520IC5) calls write_audit_trail() which is mapped to an RPG program name. It passes by OS-linkage convention the user ID, item name, price, quantity, taxrate, and formatted cost.
 - e. The ILE Compiler by default converts a short integer to an integer unless the nowiden parameter is specified on the #pragma linkage directive. For example, the short integer in the ILE C program is converted to an integer, and then passed to the OPM RPG program. The RPG program is expecting a 4-byte integer for the quantity variable.
 - f. OUTPUT(*PRINT) specifies that you want a compiler listing. OPTION(*SHOWINC *NOLOGMSG) specifies that you want to expand include files in a compiler listing, and not log CHECKOUT option messages.
 - g. FLAG(30) specifies that you want severity level 30 messages to appear in the listing. MSGLMT(10) specifies that you want compilation to stop after 10 messages at severity level 30. CHECKOUT(*PARM) shows a list of function parameters that are not used. DBGVIEW(*ALL) specifies that you want all three views and debug data to debug this program.
5. Create an OPM COBOL program using the source shown in [Figure 237 on page 339](#). On a command line, enter:

```
CRTCBPLPGM PGM(MYLIB/T1520CB1) SRCFILE(QCPPLE/QALBLSRC)
```

```

IDENTIFICATION DIVISION.
PROGRAM-ID. T1520CB1.
*****
* parameters:
*   incoming:  PRICE, QUANTITY
*   returns :  TOTAL-COST (PRICE*QUANTITY*1.TAXRATE)
*             SUCCESS-FLAG.
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-TOTAL-COST          PIC S9(13)V99          COMP-3.
01 WS-TAXRATE            PIC S9V99              COMP-3.

LINKAGE SECTION.
01 LS-PRICE              PIC S9(8)V9(2)          COMP-3.
01 LS-QUANTITY           PIC S9(4)              COMP-4.
01 LS-TAXRATE            PIC S9V99              COMP-3.
01 LS-TOTAL-COST         PIC $$$,$$$,$$$,$$$,$$.99
01 LS-OPERATION-SUCCESSFUL PIC X
PROCEDURE DIVISION USING LS-PRICE
                          LS-QUANTITY
                          LS-TAXRATE
                          LS-TOTAL-COST
                          LS-OPERATION-SUCCESSFUL.
                          DISPLAY.
                          DISPLAY.
                          0

MAINLINE.
MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
PERFORM CALCULATE-COST.
PERFORM FORMAT-COST.
EXIT PROGRAM.
CALCULATE-COST.
MOVE LS-TAXRATE TO WS-TAXRATE.
ADD 1 TO WS-TAXRATE.
COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                LS-PRICE *
                                WS-TAXRATE

ON SIZE ERROR
MOVE "N" TO LS-OPERATION-SUCCESSFUL
END-COMPUTE.
FORMAT-COST.
MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

Figure 237. T1520CB1 – OPM COBOL Source to Calculate Tax and Format Cost

Note:

- a. This program receives pointers to the values of the variables price, quantity, and taxrate, and pointers to formatted_cost and success_flag.
 - b. The calc_and_format() function in program T1520CB1 calculates and formats the total cost. Parameters are passed from the ILE C program to the OPM COBOL program to do the tax calculation.
 - c. The *OPM COBOL User's Guide* contains information on how to compile an OPM COBOL program.
6. Create an OPM RPG program using the source shown in [Figure 238 on page 340](#). On a command line, enter:

```
CRTRPGPGM PGM(MYLIB/T1520RP1) SRCFILE(QCPPL/QARPGSRC) OPTION(*SOURCE *SECLVL)
```

```

FT1520DD20  E          DISK          A
F          T1520DD2R          KRENAMEDD2R
IQTYIN      DS
I          B  1  40QTYBIN
C          *ENTRY  PLIST
C          PARM          USER  10
C          PARM          ITEM  20
C          PARM          PRICE 102
C          PARM          QTYIN
C          PARM          TXRATE 22
C          PARM          TOTAL 21
C          EXSR  ADDRCD
C          SETON          LR
C          ADDRCD  BEGSR
C          MOVELUDATE  DATE
C          MOVE  QTYBIN  QTY
C          WRITEDD2R
C          ENDSR

```

Figure 238. T1520RP1 – OPM RPG Source to Write the Audit Trail

Note:

- a. The write_audit_trail() function in the program T1520RP1 writes the audit trail for the program.
 - b. The *RPG/400® User's Guide* contains information on how to compile an OPM RPG program.
7. Enter data for the program T1520IC5.
- a. On a command line, type:

```
T1520CM2
```

and press F4 (Prompt).

- b. Type the following data into T1520CM2:

```

Hammers
1.98
5000
Nails
0.25
2000

```

The output is as follows:

```

5000 HAMMERS plus tax =          $11,385.00
Press ENTER to end terminal session.
>
2000 NAILS plus tax =           $575.00
Press ENTER to end terminal session.

```

The physical file T1520DD2 contains the data as follows:

```

SMITHE  HAMMERS          00000000198500015          $11,385.00072893
SMITHE  NAILS           00000000025200015          $575.00072893

```

ILE CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C Program

This example is an ILE version of the small transaction processing program [“OPM CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C Program”](#) on page 333.

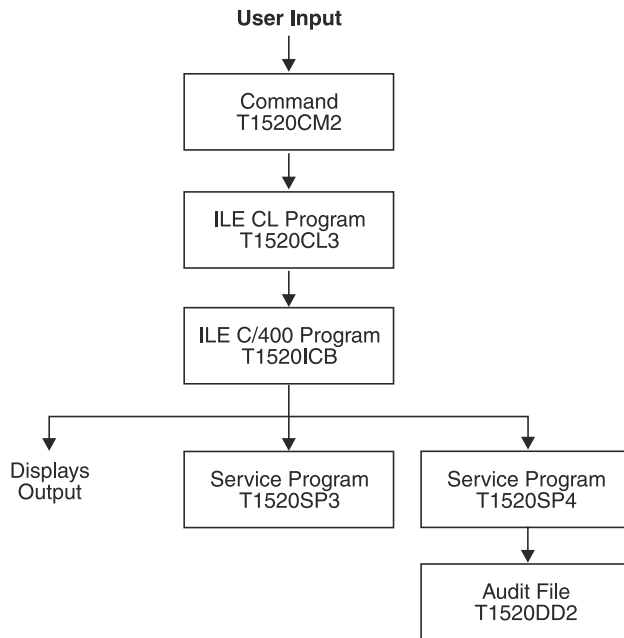


Figure 239. Basic Object Structure

This example consists of:

- A CL command that accepts user's input and passes it to an ILE CL program.
- A ILE CL program that processes the input and passes it to an ILE C program.
- The ILE C program calls ILE procedures to process the input. Output is then written to the user's terminal.
- An ILE COBOL procedure that completes the calculation and formats the cost.
- An ILE RPG procedure that updates the audit file with each transaction.

Note: In addition to the source for CMD, ILE CL, ILE C, ILE RPG and ILE COBOL you need DDS source for the output file, which is the same as the previous example.

Program Modules and Activation Groups

As shown in [Figure 240](#) on page 342, the CL and C programs are activated within a new activation group. The ILE CL program is created with the CRTPGM default for the ACTGRP parameter, ACTGRP(*NEW). The ILE C program is created with ACTGRP(*CALLER).

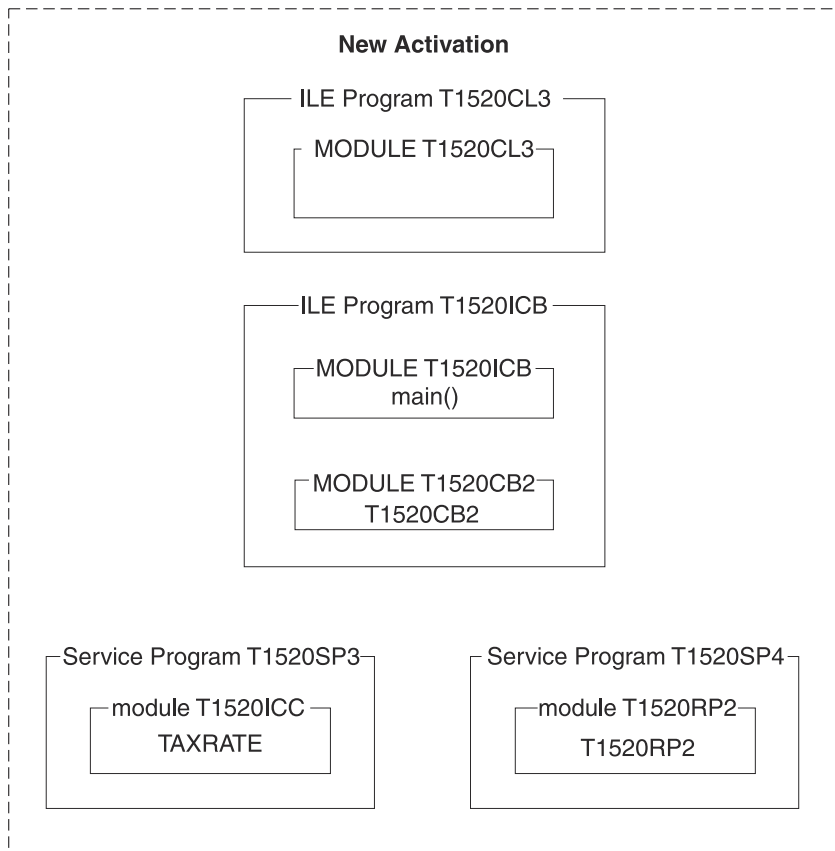


Figure 240. Integrated Language Environment Structure

Programming Tasks

The following steps show you how to:

- Create a physical file to contain an audit trail for the ILE C program.
- Create an ILE CL program that passes the parameters item name, price, quantity, and user ID to an ILE C program.
- Create a CL command prompt to enter data for item name, price, and quantity. The command passes the data to the ILE CL program which in turn calls an ILE C program.
- Create an ILE COBOL module that completes the tax calculation and formats the total cost.
- Create an ILE RPG module that writes the audit trail for the application.
- Create a service program that exports a data item.
- Create a service program that exports an RPG procedure.
- Create one program with a `main()` function that receives incoming arguments from a CL program. The program calls an ILE COBOL procedure to complete the tax calculation and format the total cost, and calls an ILE RPG procedure to write audit records.

1. To create a physical file T1520DD2 using the source shown in [Figure 241 on page 343](#), enter:

```
CRTPF FILE(MYLIB/T1520DD2) SRCFILE(QCPPLE/QADDSSRC) MAXMBRS(*NOMAX)
```



```

R T1520DD2R
      USER          10          COLHDG('User')
      ITEM           20          COLHDG('Item name')
      PRICE          10S 2       COLHDG('Unit price')
      QTY            4S          COLHDG('Number of items')
      TXRATE         2S 2       COLHDG('Current tax rate')
      TOTAL          21          COLHDG('Total cost')
      DATE           6           COLHDG('Transaction date')
K  USER

```

Figure 241. T1520DD2 – Source to Create Physical Files

This file contains the audit trail for the ILE C program T1520ICB.

- To create a CL program T1520CL3 that the source shown in Figure 242 on page 343, enter:

```

CRTCLMOD MODULE(MYLIB/T1520CL3) SRCFILE(QCPPLE/QACL SRC)
CRTPGM PGM(MYLIB/T1520CL3) MODULE(MYLIB/T1520CL3) ACTGRP(*NEW)

```

```

/* ILE version of T1520CL2                                     */
PGM          PARM(&ITEMIN &PRICE &QUANTITY)
DCL          VAR(&USER) TYPE(*CHAR) LEN(10)
DCL          VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
DCL          VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
DCL          VAR(&PRICE) TYPE(*DEC) LEN(10 2)
DCL          VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
DCL          VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM                 */
CHGVAR      VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
/* GET THE USERID FOR THE AUDIT FILE                         */
RTVJOBA     USER(&USER)
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF      FILE(T1520DD2) TOFILE(*LIBL/T1520DD2) +
            MBR(T1520DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL        PGM(T1520ICB) PARM(&ITEMOUT &PRICE &QUANTITY +
            &USER)
DLTOVR      FILE(*ALL)
ENDPGM

```

Note: To pass parameters to an ILE program when you run it, use the PARM option of the CL Call (CALL) command.

Figure 242. T1520CL3 – ILE CL Source to Pass Variables to an ILE C Program

This program passes the CL variables item name, price, quantity, and user ID by reference to an ILE C program T1520ICB. The Retrieve Job Attributes (RTVJOBA) command obtains the user ID for the audit trail. Arguments are passed by reference. They can be changed by the receiving ILE C program. The variable item_name is null ended in the CL program.

- To create a CL command prompt T1520CM2 using the source in Figure 243 on page 343, enter:

```

CRTCMD CMD(MYLIB/T1520CM2) PGM(MYLIB/T1520CL3) SRCFILE(QCPPLE/QACMSRC)

```

```

CMD          PROMPT('CALCULATE TOTAL COST')
PARM         KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
            MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
PARM         KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
            RANGE(0.01 99999999.99) MIN(1) +
            ALWUNPRT(*YES) PROMPT('Unit price' 2)
PARM         KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
            9999) MIN(1) ALWUNPRT(*YES) +
            PROMPT('Number of items' 3)

```

Figure 243. T1520CM2 – Source to Create a CL Command Prompt

You use this CL command to enter the item name, price, and quantity for the ILE C program T1520ICB.

- To create the module T1520ICB using the source shown, enter:

```

CRTCMOD MODULE(MYLIB/T1520ICB) SRCFILE(QCPPLE/QACSRC)

```

```

/* This program demonstrates the interlanguage call capability */
/* of an ILE C program. This program is called by a CL */
/* program that passes an item name, price, quantity and user ID. */
/* A COBOL procedure is called to calculate and format total cost.*/
/* An RPG procedure is called to write an audit trail. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <decimal.h>
/* The #pragma map directive maps a function name to the bound */
/* procedure name so that the purpose of the procedure is clear. */
#pragma map(calc_and_format,"T1520CB2")
#pragma map(write_audit_trail,"T1520RP2")
/* Tell the compiler that there are bound procedure calls and */
/* arguments are to be passed by value-reference. */
#pragma argument(calc_and_format, OS, nowiden)
#pragma argument(write_audit_trail, OS)
void calc_and_format(decimal (10,2),
                    short int,
                    char[],
                    char *);
void write_audit_trail(char[],
                      char[],
                      decimal(10,2),
                      short int,
                      char[]);
extern decimal (2,2) TAXRATE; /* TAXRATE is in *SRVPGM T1520SP3 */
int main(int argc, char *argv[])
{
/* Incoming arguments from a CL program have been verified by */
/* the *CMD and null ended within the CL program. */
/* Incoming arguments are passed by reference from a CL program. */
char *user_id;
char *item_name;
short int quantity;
decimal (10, 2) price;
char formatted_cost[22];
/* Remove null terminator for RPG for iSeries program. Item name is null */
/* ended for C. */
char rpg_item_name[20];
char null_formatted_cost[22];
char success_flag = 'N';
int i;
/* Incoming arguments are all pointers. */
item_name = argv[1];
price = *((decimal (10, 2) *) argv[2]);
quantity = *((short *) argv[3]);
user_id = argv[4];
/* Call the COBOL program to do the calculation, and return a */
/* Y/N flag, and a formatted result. */
calc_and_format(price,
                quantity,
                formatted_cost,
                &success_flag);

memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

/* Null end the result. */

formatted_cost[21] = '\0';
if (success_flag == 'Y')
{
for (i=0; i<20; i++)
{
/* Remove the null end for the RPG for iSeries program. */
if (*(item_name+i) == '\0')
{
rpg_item_name[i] = ' ';
}
else
{
rpg_item_name[i] = *(item_name+i);
}
}
}

/* Call an RPG program to write audit records. */

write_audit_trail(user_id,
                  rpg_item_name,
                  price,
                  quantity,

```

```

        formatted_cost);

    printf("\n%d %s plus tax = %-s\n", quantity,
           item_name,
    }
else
{
    printf("Calculation failed\n");
}
}
}

```

Note:

- a. The main() function in this module receives the incoming arguments from the ILE CL program T1520CL3 that are verified by the CL command T1520CM2 and null ended within the CL program T1520CL3. All the incoming arguments are pointers.
 - b. The main() function in this program calls calc_and_format() which is mapped to a ILE COBOL procedure name. It passes by OS-linkage convention the price, quantity, formatted cost, and a success flag. The ILE OPM COBOL procedure is not expecting widened parameters, the default for ILE C. This is why nowiden is used in the #pragma argument directive. The formatted cost and the success flag values are updated in the procedure T1520CB2.
 - c. If calc_and_format() return successfully the main() function in this program (T1520ICB) calls write_audit_trail() which is mapped to an ILE RPG procedure name. It passes by OS-linkage convention (also called by value-reference) the user ID, item name, price, quantity, and formatted cost. The ILE C compiler by default converts a short integer to an integer unless the nowiden parameter is specified on the #pragma argument directive. For example, the short integer in the ILE C program is converted to an integer, and then passed to the ILE RPG procedure. The RPG procedure is expecting a 4 byte integer for the quantity variable.
5. To create module T1520ICC using the source shown in [Figure 244 on page 345](#), enter:

```

CRTCMOD MODULE(MYLIB/T1520ICC) SRCFILE(QCPPLE/QACSRC)

/* Export the tax rate data.                                     */
#include <decimal.h>
const decimal (2,2) TAXRATE = .15D;

```

Figure 244. T1520ICC – Source Code to Export Tax Rate Data

TAXRATE is exported from this module to ILE C, COBOL, and RPG procedures.

Note: Weak definitions (EXTERNALs from COBOL) cannot be exported out of a service program to a strong definition language like C. C can export to COBOL, hence the choice of language for TAXRATE.

6. To create an ILE COBOL procedure using the source shown in [Figure 245 on page 346](#), enter:

```

CRTCBMOD MODULE(MYLIB/T1520CB2) SRCFILE(QCPPLE/QALBLSRC)

```

```

IDENTIFICATION DIVISION.
PROGRAM-ID. T1520CB2 INITIAL.
*****
* parameters:
*   incoming:  PRICE, QUANTITY
*   returns :  TOTAL-COST (PRICE*QUANTITY*1.TAXRATE)
*             SUCCESS-FLAG.
*   TAXRATE :  An imported value.
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-TOTAL-COST          PIC S9(13)V99          COMP-3.
01 WS-TAXRATE            PIC S9V99              COMP-3
                                VALUE 1.
                                COMP-3.
01 TAXRATE               EXTERNAL PIC SV99      COMP-3.
LINKAGE SECTION.
01 LS-PRICE              PIC S9(8)V9(2)        COMP-3.
01 LS-QUANTITY           PIC S9(4)             COMP-4.
01 LS-TOTAL-COST         PIC $$$,$$$,$$$,$$$,$$.99
                                DISPLAY.
01 LS-OPERATION-SUCCESSFUL PIC X              DISPLAY.

```

```

PROCEDURE DIVISION USING LS-PRICE
                      LS-QUANTITY
                      LS-TOTAL-COST
                      LS-OPERATION-SUCCESSFUL.
MAINLINE.
  MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
  PERFORM CALCULATE-COST.
  PERFORM FORMAT-COST.
  EXIT PROGRAM.
CALCULATE-COST.
  ADD TAXRATE TO WS-TAXRATE.
  COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                LS-PRICE *
                                WS-TAXRATE
  ON SIZE ERROR
    MOVE "N" TO LS-OPERATION-SUCCESSFUL
  END-COMPUTE.
FORMAT-COST.
  MOVE WS-TOTAL-COST TO LS-TOTAL-COST.

```

Figure 245. T1520CB2 – ILE COBOL Source to Calculate Tax and Format Cost

Note:

- a. This program receives pointers to the values of the variables price and quantity, and pointers to formatted_cost and success_flag.
 - b. The calc_and_format() function is procedure T1520CB2. It calculates and formats the total cost.
 - c. The *ILE COBOL Programmer's Guide* contains information on how compile an ILE COBOL source program.
7. To create an ILE RPG procedure using the source shown in [Figure 246](#) on page 347, enter:

```
CRTRPGMOD MODULE(MYLIB/T1520RP2) SRCFILE(QCPPLE/QARPGSRC)
```

```

FT1520DD2  0  A  E          DISK
D  TAXRATE          S          3P 2  IMPORT
D  QTYIN            DS
D  QTYBIN          1          4B 0
C          *ENTRY      PLIST
C          PARM          USER          10
C          PARM          ITEM          20
C          PARM          PRICE         10 2
C          PARM          QTYIN
C          PARM          TOTAL         21
C          EXSR          ADDRREC
C          SETON
C          ADDRREC      BEGSR          LR
C          MOVE          UDATE          DATE
C          MOVE          QTYBIN         QTY
C          MOVE          TAXRATE        TXRATE
C          WRITE        T1520DD2R
C          ENDSR

```

Figure 246. T1520RP2 – ILE RPG Source to Write the Audit Trail

Note:

- a. The `write_audit_trail()` function is the procedure T1520RP2. It writes the audit trail for the program.
 - b. The *ILE RPG Programmer's Guide* contains information on how to compile an ILE RPG source program.
8. To create the service program T1520SP3 from the module T1520ICC, enter:

```

CRTSRVPGM SRVPGM(MYLIB/T1520SP3) MODULE(MYLIB/T1520ICC) +
EXPORT(*SRCFILE) SRCFILE(QCPPLE/QASRVSRC)

```

The T1520SP3 service program exports `taxrate`. The export list is specified in T1520SP3 in QASRVSRC.

9. To create the service program T1520SP4 from the module T1520RP2, enter:

```

CRTSRVPGM SRVPGM(MYLIB/T1520SP4) MODULE(MYLIB/T1520RP2) +
EXPORT(*SRCFILE) SRCFILE(QCPPLE/QASRVSRC)

```

The T1520SP4 service program exports procedure T1520RP2. The export list is specified in T1520SP4 in QASRVSRC.

10. To create the program T1520ICB enter:

```

CRTPGM PGM(MYLIB/T1520ICB) MODULE(MYLIB/T1520ICB MYLIB/T1520CB2) +
BNDSRVPGM(MYLIB/T1520SP3 MYLIB/T1520SP4) ACTGRP(*CALLER)

```

T1520ICB is considered the application's main program. It will run in the new activation group that was created when T1520CL3 was called.

11. To enter data for the program T1520ICB, type T1520CM2 and press F4 (Prompt):

Type the following data into T1520CM2:

```

Hammers
1.98
5000
Nails
0.25
2000

```

The output is as follows:

```

5000 HAMMERS plus tax =          $11,385.00
Press ENTER to end terminal session.
>
2000 NAILS plus tax =           $575.00
Press ENTER to end terminal session.

```

The physical file T1520DD2 contains the following data:

SMITHE	HAMMERS	0000000198500015	\$11,385.00072893
SMITHE	NAILS	0000000025200015	\$575.00072893

ILE-OPM CL Example: Calling OPM, COBOL, and RPG Programs from an ILE C++ Program

This program demonstrates some typical steps in creating a program that uses several ILE and OPM programming languages.

Program Description

The program is a small transaction-processing program that takes as input the item name, price, and quantity for one or more products. As output, the program displays the total cost of the items specified on the display and writes an audit trail of the transactions to a file.

Figure 247 on page 348 shows the basic flow of the program.

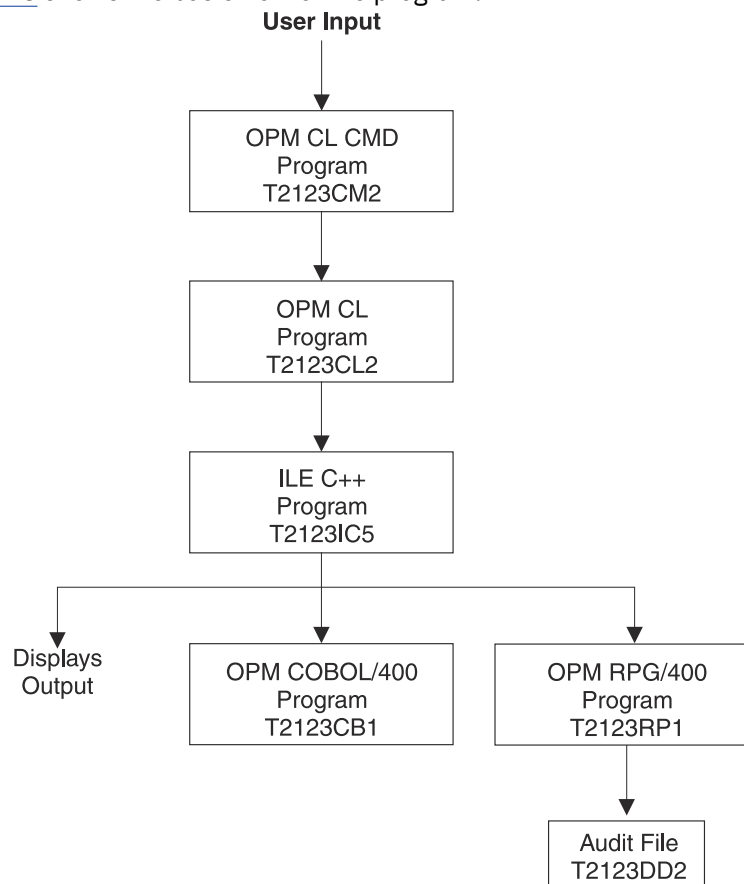


Figure 247. ILE-OPM CL Example: Basic Program Structure

Program Structure

The program consists of these components:

- A CL command T2123CM2 that accepts the users input and passes it to an OPM CL program
- An OPM CL program T2123CL2 that processes the input and passes it to an ILE C++ program
- An ILE C++ program T2123IC5 that calls an OPM COBOL program to process the input, and an OPM RPG program to write the audit trail to an externally described file
- An OPM COBOL program T2123CB1 that completes the calculation and formats the cost
- An OPM RPG program T2123RP1 that updates the audit file with each transaction

- An externally described file T2123DD2 that receives the audit trail

Program Activation

The ILE C++ program T2123IC5 is created with the CRTPGM default for the *ACTGRP* parameter, *ACTGRP(*NEW)*. When the CL program calls the ILE C++ program, a new activation group is started.

The OPM CL, COBOL, and RPG programs are activated within the OPM default activation group.

Figure 248 on page 349 shows the structure of this program in ILE.

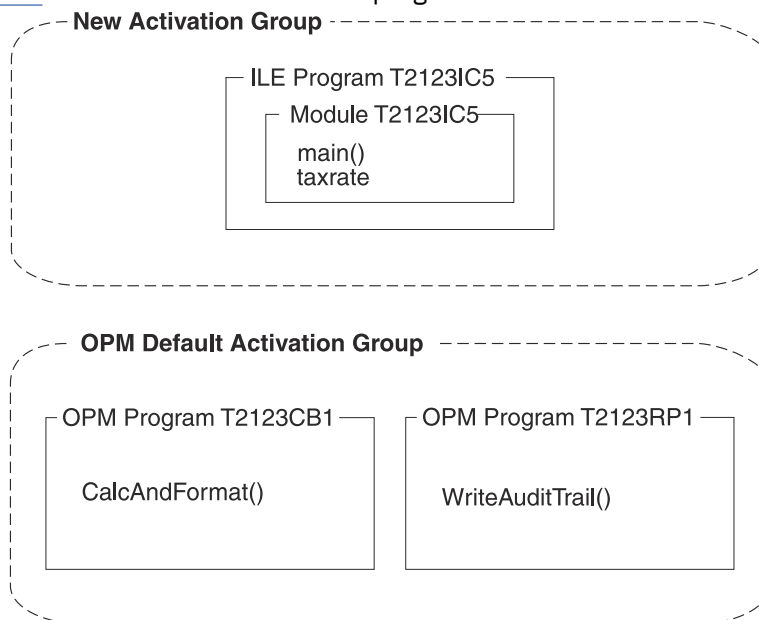


Figure 248. Structure of the Program in ILE C++

Program Files

The source code for each of the files that compose this program are an externally described file, a CL program, a CL command prompt, a C++ source file, and OPM COBOL program and an OPM RPG program.

Externally Described File T2123DD2

The file T2123DD2 contains the audit trail for the C++ program T2123IC5. The DDS source defines the fields for the audit file:

```
R T2123DD2R
  USER          10          COLHDG('User')
  ITEM          20          COLHDG('Item name')
  PRICE        10S 2        COLHDG('Unit price')
  QTY           4S          COLHDG('Number of items')
  TXRATE       2S 2        COLHDG('Current tax rate')
  TOTAL        21          COLHDG('Total cost')
  DATE         6           COLHDG('Transaction date')
K  USER
```

CL Program T2123CL2

The CL program T2123CL2 passes the CL variables *item_name*, *price*, *quantity* and *user_id* by reference to an ILE C++ program T2123IC5.

```
PGM          PARM(&ITEMIN &PRICE &QUANTITY)
DCL          VAR(&USER) TYPE(*CHAR) LEN(10)
DCL          VAR(&USEROUT) TYPE(*CHAR) LEN(11)
DCL          VAR(&ITEMIN) TYPE(*CHAR) LEN(20)
DCL          VAR(&ITEMOUT) TYPE(*CHAR) LEN(21)
DCL          VAR(&PRICE) TYPE(*DEC) LEN(10 2)
DCL          VAR(&QUANTITY) TYPE(*DEC) LEN(2 0)
DCL          VAR(&NULL) TYPE(*CHAR) LEN(1) VALUE(X'00')
/* ADD NULL TERMINATOR FOR THE ILE C PROGRAM */
```

```

CHGVAR      VAR(&ITEMOUT) VALUE(&ITEMIN *TCAT &NULL)
CHGVAR      VAR(&USEROUT) VALUE(&USER *TCAT &NULL)
/* GET THE USERID FOR THE AUDIT FILE */
RTVJOBA     USER(&USER)
/* ENSURE AUDIT RECORDS WRITTEN TO CORRECT AUDIT FILE MEMBER */
OVRDBF      FILE(T2123DD2) TOFILE(*LIBL/T2123DD2) +
            MBR(T2123DD2) OVRSCOPE(*CALLLVL) SHARE(*NO)
CALL        PGM(T2123IC5) PARM(&ITEMOUT &PRICE &QUANTITY +
            &USEROUT)
DLTOVR      FILE(*ALL)
ENDPGM

```

The Retrieve Job Attributes (RTVJOBA) command obtains the user ID for the audit trail. Arguments are passed by reference. They can be changed by the receiving ILE C++ program. The variables containing the user and item names are explicitly null-terminated in the CL program.

Note:

1. CL variables and numeric literals are not passed to an ILE C++ program with null-terminated strings.
2. Character literals and logical literals are passed as null-terminated strings but are not widened with blanks.
3. Numeric literals such as packed decimals are passed as 15,5 (8 bytes). Floating point constants are passed as double precision floating point values (1.2E+15).
4. To pass parameters to an ILE program when you run it, use the PARM option of the CL Call (CALL) command.

CL Command Prompt T2123CM2

You use the CL command prompt T2123CM2 to prompt the user to enter item names, prices, and quantities that will be used by the C++ program T2123IC5.

```

CMD          PROMPT('CALCULATE TOTAL COST')
PARM         KWD(ITEM) TYPE(*CHAR) LEN(20) RSTD(*NO) +
            MIN(1) ALWUNPRT(*NO) PROMPT('Item name' 1)
PARM         KWD(PRICE) TYPE(*DEC) LEN(10 2) RSTD(*NO) +
            RANGE(0.01 99999999.99) MIN(1) +
            ALWUNPRT(*YES) PROMPT('Unit price' 2)
PARM         KWD(QUANTITY) TYPE(*INT2) RSTD(*NO) RANGE(1 +
            9999) MIN(1) ALWUNPRT(*YES) +
            PROMPT('Number of items' 3)

```

C++ Source File T2123IC5

The C++ source file T2123IC5 contains a main() function which receives the incoming arguments from the CL program T2123CL2. These arguments have been verified by the CL command prompt T2123CM2 and null-terminated within the CL program T2123CL2. All the incoming arguments are pointers.

The main() function calls the function CalcAndFormat() which is mapped to a COBOL name. It passes the price, quantity, taxrate, formatted_cost, and a success_flag to the OPM COBOL program T2123CB1 using the extern "OS nowiden" linkage specification, because the OPM COBOL program is not expecting widened parameters.

The formatted_cost and the success_flag values are updated in the C++ program T2123IC5.

If CalcAndFormat() returns successfully, a record is written to the audit trail by WriteAuditTrail() in the OPM RPG program.

The main() function in program T2123IC5 calls WriteAuditTrail() which is mapped to an RPG program name, and passes the user_id, item_name, price, quantity, taxrate, and formatted_cost, using the extern "OS" linkage specification.

Note: By default, the compiler converts a short integer to an integer unless the nowiden parameter is specified on the extern linkage specification. The short integer in the C++ program is converted to an integer, and then passed to the OPM RPG program. The RPG program is expecting a 4-byte integer for

the quantity variable. See [“Interlanguage Data-Type Compatibilities”](#) on page 425 for information on data-type compatibility.

```
// This program is called by a CL program that passes an item
// name, price, quantity and user ID.
// COBOL is called to calculate and format the total cost.
// RPG is called to write an audit trail.

#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <bcd.h>

// The #pragma map directive maps a new program name to the existing
// program name so that the purpose of the program is clear.
// Tell the compiler that there are dynamic program calls so
// arguments are passed by value-reference.

extern "OS nowiden" void CalcAndFormat(_DecimalT <10,2>,
                                     short int, _DecimalT<2,2>, char[],
                                     char *);

#pragma map(CalcAndFormat,"T2123CB1")

extern "OS" void WriteAuditTrail(char[], char[],
                                 _DecimalT<10,2>, short int,
                                 _DecimalT<2,2>, char[]);

#pragma map(WriteAuditTrail,"T2123RP1")

int main(int argc, char *argv[])
{
// Incoming arguments from a CL program have been verified by
// the *CMD and null-terminated within the CL program.
// Incoming arguments are passed by reference from a CL program.

    char          *user_id;
    char          *item_name;
    short int     quantity;
    _DecimalT <10, 2> price;
    _DecimalT <2,2> taxrate = __D(".15");
    char          formatted_cost[22];

// Remove null terminator for RPG program. Item name is null
// terminated for C++.

    char          rpg_item_name[20];
    char          null_formatted_cost[22];
    char          success_flag = 'N';
    int           i;

// Incoming arguments are all pointers.

    item_name =          argv[1];
    price     = *((_DecimalT<10, 2> *) argv[2]);
    quantity  = *((short *)      argv[3]);
    user_id   =          argv[4];

// Call the COBOL program to do the calculation, and return a
// Y/N flag, and a formatted result.

    CalcAndFormat(price, quantity, taxrate, formatted_cost,
                  &success_flag);

    memcpy(null_formatted_cost,formatted_cost,sizeof(formatted_cost));

// Null terminate the result.

    formatted_cost[21] = '\0';
    if (success_flag == 'Y')
    {
        for (i=0; i<20; i++)
        {
// Remove null terminator for the RPG program.

            if (*(item_name+i) == '\0')
            {
                rpg_item_name[i] = ' ';
            }
        }
    }
}
```

```

    else
    {
        rpg_item_name[i] = *(item_name+i);
    }
}

// Call an RPG program to write audit records.

WriteAuditTrail(user_id, rpg_item_name, price, quantity,
                taxrate, formatted_cost);

cout <<quantity <<item_name << "plus tax ="
    <<null_formatted_cost <<endl;
}
else
{
    cout <<"Calculation failed" <<endl;
}
}
}

```

OPM COBOL Program T2123CB1

The OPM COBOL program T2123CB1 receives pointers to the values of the variables price, quantity and taxrate, and pointers to formatted_cost and success_flag.

The CalcAndFormat() function in program T2123CB1 calculates and formats the total cost. Parameters are passed from the ILE C++ program to the OPM COBOL program to do the tax calculation.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. T2123CB1.
*****
* parameters:                                     *
* incoming:  PRICE, QUANTITY                       *
* returns :  TOTAL-COST (PRICE*QUANTITY*1.TAXRATE) *
*           SUCCESS-FLAG.                         *
*****
ENVIRONMENT DIVISION.                                0
CONFIGURATION SECTION.                              0
SOURCE-COMPUTER. IBM-I.                             0
OBJECT-COMPUTER. IBM-I.                             0
DATA DIVISION.                                     0
WORKING-STORAGE SECTION.

    01 WS-TOTAL-COST          PIC S9(13)V99          COMP-3.
    01 WS-TAXRATE            PIC S9V99              COMP-3.

LINKAGE SECTION.

    01 LS-PRICE              PIC S9(8)V9(2)         COMP-3.
    01 LS-QUANTITY           PIC S9(4)              COMP-4.
    01 LS-TAXRATE            PIC SV99               COMP-3.
    01 LS-TOTAL-COST         PIC $$$,$$$,$$$,$$$,$$$$.99
    01 LS-OPERATION-SUCCESSFUL PIC X                DISPLAY.
                                                DISPLAY.

PROCEDURE DIVISION USING LS-PRICE                 0
                        LS-QUANTITY
                        LS-TAXRATE
                        LS-TOTAL-COST
                        LS-OPERATION-SUCCESSFUL.

MAINLINE.
MOVE "Y" TO LS-OPERATION-SUCCESSFUL.
PERFORM CALCULATE-COST.
PERFORM FORMAT-COST.
EXIT PROGRAM.

CALCULATE-COST.
MOVE LS-TAXRATE TO WS-TAXRATE.
ADD 1 TO WS-TAXRATE.
COMPUTE WS-TOTAL-COST ROUNDED = LS-QUANTITY *
                                LS-PRICE *
                                WS-TAXRATE

    ON SIZE ERROR
        MOVE "N" TO LS-OPERATION-SUCCESSFUL
END-COMPUTE.

```

```
FORMAT-COST.  
MOVE WS-TOTAL-COST TO LS-TOTAL-COST.
```

OPM RPG Program T2123RP1

The OPM RPG program T2123RP1 contains the `WriteAuditTrail()` function which writes the audit trail for the program.

```
FT2123DD20  E          DISK          A  
F          T2123DD2R          KRENAMEDD2R  
IQTYIN     DS  
I          B  1  40QTYBIN  
C          *ENTRY  PLIST  
C          PARM      USER  10  
C          PARM      ITEM  20  
C          PARM      PRICE 102  
C          PARM      QTYIN  
C          PARM      TXRATE 22  
C          PARM      TOTAL 21  
C          EXSR  ADDR  
C          SETON          LR  
C          ADDR  
C          BEGSR  
C          MOVE  LDATE  DATE  
C          MOVE  QTYBIN QTY  
C          WRITEDD2R  
C          ENDSR
```

Invoking the ILE-OPM Program

To enter data for the program T2123IC5 enter the command T2123CM2 and press F4 (Prompt).

You can enter this data into T2123CM2:

```
Hammers  
1.98  
5000  
Nails  
0.25  
2000
```

The output is:

```
5000 HAMMERS plus tax =          $11,385.00  
Press ENTER to end terminal session.  
>  
2000 NAILS plus tax =          $575.00  
Press ENTER to end terminal session.
```

The physical file T2123DD2 contains this data:

```
SMITHE  HAMMERS          00000000198500015          $11,385.0007          2893  
SMITHE  NAILS           00000000025200015          $575.0007           2893
```

Using a Linkage Specification to Call an ILE Procedure

C++ provides a linkage specification to enable procedure calls and the sharing of data between the C++ caller and the called procedure. See [“Using a Linkage Specification in a C++ Dynamic Program Call”](#) on page 354 for the syntax.

The valid string literals for the linkage specification to call ILE procedures are:

Linkage Specification

Type of Procedure Called

"C++"

ILE C++ procedure (default)

"C"

ILE C procedure

"C nowiden"

ILE C procedure without widened parameters

"RPG"

ILE RPG procedure

"COBOL"

ILE COBOL procedure

"CL"

ILE CL procedure

"ILE"

General ILE function call

"ILE nowiden"

ILE function call without widened parameters

"VREF"

ILE function call with pointers in temporary storage. (Behaves the same as a regular call although parameters are passed to the function as if they were by reference.)

"VREF nowiden"

Same as "VREF" without widened parameters

Using a Linkage Specification in a C++ Dynamic Program Call

You can call OPM, ILE, or EPM programs from a C++ program. OPM, ILE or EPM programs can also call a C++ program.

C++

C++ provides a linkage specification to enable dynamic program calls and sharing of data between them. For a syntax diagram and additional information, see the *ILE C/C++ Language Reference*.

Valid String Literals

The *"string-literal"* is used to specify the linkage associated with a particular function. The string literals used in linkage specifications are case-insensitive. The valid string literals for the linkage specification to call programs are:

"OS"

OS linkage call

"OS nowiden"

OS linkage call without widened parameters. See ["Specifying that a Function Has External \(OS\) Linkage"](#) on page 317 for details.

Linkage Specification

C++

If you want a C++ program to call an ILE, OPM, or EPM program (*PGM), use the `extern "OS"` linkage specification in your C++ source to tell the compiler that the called program is an external program, not a bound ILE procedure. For example, if you want a C++ program to call an OPM COBOL program (*PGM) this `extern "OS"` linkage specification in your C++ source tells the compiler that `COBOL_PGM` is an external program, not a bound ILE procedure.

```
extern "OS" void COBOL_PGM(void);
```

If you want an ILE, OPM or EPM program to call a C++ program, use the ILE, OPM, or EPM language-specific call statement.

Using Packed Decimal Data in a C Program

C

This topic describes how to:

- [Convert from packed decimal data types](#)
- [Pass a pointer to packed decimal data to a function](#)
- [Call another program that contains packed decimal data](#)
- [Use library functions with packed decimal data](#)
- [Understand packed decimal data type errors](#)

The packed decimal data type representation includes integral and fractional parts. The ILE C compiler supports the packed decimal data type as an extension to ISO C.

Note: This is strictly a C data type. C++ decimal support is provided in the `bcd` class. The header file is `bcd.h`. For more information, refer to the *ILE C/C++ Language Reference*.

You can use the packed decimal data type to:

- Represent large numeric quantities accurately, especially in business and commercial applications for financial calculations. For example, the fractional part of a dollar can be represented accurately by two digits that follow the decimal point.

Note: You do not have to use floating point arithmetic. Floating point is more suitable for scientific and engineering computations, which often use numbers that:

- Are much larger than the largest packed decimal variable can store
 - Are much smaller than the smallest packed decimal, but do not have enough precision for commercial use
- Declare type definitions, arrays, structures, and unions that have packed decimal members. You can apply operators (unary operators) on packed decimal variables. Bitwise operators do not apply to packed decimal data. The packed decimal data type in ILE C is compatible with packed decimal representations in RPG and COBOL. You can also define macros and call library functions with packed decimal arguments. The *ILE C/C++ Language Reference* contains information on the packed decimal data type.

Note: To use the `decimal`, `digitsof`, and `precisionof` macros in your code, you must specify the `<decimal.h>` header file in your ILE C source.

Converting from Packed Decimal Data Types

If the value of the packed decimal type to be converted is within the range of values that can be represented exactly, the value of the packed decimal type is not changed. Packed decimal types are compatible if their types are the same. For example, `decimal(n1, p1)` and `decimal(n2, p2)` have compatible types if and only if $(n_1 = n_2)$ and $(p_1 = p_2)$.

Converting from a Packed Decimal Type to a Packed Decimal Type

The following example illustrates different conversions from packed decimal types to packed decimal types that have different sizes. If the value of the packed decimal type to be converted is not within the range of values that can be represented exactly, the value of the packed decimal type is truncated. If truncation occurs in the fractional part, the result is truncated, and there is no runtime error.

```

#include <decimal.h>
int main (void)
{
    decimal(4,2) targ_1, targ_2;
    decimal(6,2) op_1=1234.56d, op_2=12.34d;
    targ_1=op_1;      /* A runtime error is generated because the integral
                       part is truncated; targ_1=34.56d.          */
    targ_2=op_2;      /* No runtime error is generated because neither the
                       integral nor the fractional part is truncated;
                       targ_2=12.34d.                            */
}

```

Figure 249. ILE C Source to Convert Packed Decimals

If assignment causes truncation in the integral part, then there is a runtime error. A runtime exception occurs when an integral value is lost during conversion to a different type, regardless of what operation requires the conversion. See [“Understanding Packed Decimal Data Type Errors”](#) on page 364 for an example of runtime exceptions.

Examples:

There is no warning or error during compilation on assignment to a smaller target. See [“Understanding Packed Decimal Data Type Errors”](#) on page 364 for information on compile time and runtime errors during conversion.

The following example shows conversion from one packed decimal type to another with a smaller precision. Truncation on the fractional part results.

```

#include <decimal.h>
int main(void)
{
    decimal(7,4) x = 123.4567D;
    decimal(7,1) y;
    y = x;      /* y = 123.4D */
}

```

Figure 250. ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Precision

The next example shows conversion from one packed decimal type to another with a smaller integral part. Truncation on the integral part results. The `#pragma nosigtrunc` directive turns off exceptions generated because of overflow.

```

#pragma nosigtrunc
#include <decimal.h>
int main (void)
{
    decimal(8,2) x = 123456.78D;
    decimal(5,2) y;
    y = x;      /* y = 456.78D */
}

```

Figure 251. ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Integral Part

The next example shows conversion from one packed decimal type to another with a smaller integral part and smaller precision. Truncation on both integral and fractional parts results. The `#pragma nosigtrunc` directive turns off exceptions generated because of overflow.

```

#pragma nosigtrunc
#include <decimal.h>
int main (void)
{
    decimal(8,2) x = 123456.78D;
    decimal(4,1) y;
    y = x;      /* y = 456.7D */
}

```

Figure 252. ILE C Source to Convert a Packed Decimal to a Packed Decimal with Smaller Integral Part and Smaller Precision

Converting from a Packed Decimal Type to an Integer Type

When you convert a value of a packed decimal type to an integer type, the value becomes a packed decimal (20,0), which then becomes an integer type. High-order bits will be truncated depending on the size of the integer type. No runtime exception occurs when assigning a packed decimal to an integer type that results in truncation of the integral part.

Examples:

The following example shows the conversion from a packed decimal type that has a fractional part to an integer type.

```
#include <decimal.h>
int main (void)
{
    int op;
    decimal(7,2) op1 = 12345.67d;
    op = op1;                /* Truncation on the fractional */
                           /* part. op=12345                */
}
```

Figure 253. ILE C Source to Convert a Packed Decimal with a Fractional Part to an Integer

The following example shows the conversion from a packed decimal type that has less than 10 digits in the integral part to an integer type.

```
#include <decimal.h>
int main(void)
{
    int op;
    decimal(3) op2=123d;
    op = op2;                /* No truncation and op=123 */
}
```

Figure 254. ILE C Source to Convert a Packed Decimal with Less than 10 Digits in the Integral Part to an Integer

The following example shows the conversion from a packed decimal type that has more than 10 digits in the integral part to an integer type.

```
#include <decimal.h>
int main (void)
{
    int op2;
    decimal(12) op3;
    op3 = 123456789012d;
    op2 = op3;                /* High-order bits will be truncated.*/
                           /* op2 = 0xBE991A14                */
}
```

Figure 255. ILE C Source to Convert a Packed Decimal with More than 10 Digits in the Integral Part to an Integer

The following example shows conversion from a packed decimal type that has a fractional part, and an integral part having more than 10 digits to an integer type.

```
#include <decimal.h>
int main (void)
{
    int op;
    long long op_2;
    decimal(15,2) op_1 = 1234567890123.12d;
    op = op_1;                /* High-order bits will be truncated. */
    op_2 = op_1;                /* op_2 = 1234567890123, op = 0x71FB04CB */
}
```

Figure 256. ILE C Source to Convert a Packed Decimal with More than 10 Digits in Both Parts to an Integer

Converting from a Packed Decimal Type to a Floating Point Type

When a value of packed decimal type is converted to floating type, if the value being converted is outside the range of values that can be represented, then the behavior is undefined. If the value being converted is within the range of values that can be represented, but cannot be represented exactly, the result is truncated. When a float or a double is converted to a packed decimal with smaller precision, the fractional part of the float or the double will be truncated.

The following example shows the conversion from a packed decimal type to a floating point type.

```
#include <decimal.h>
#include <stdio.h>
int main(void)
{
    decimal(5,2) dec_1=123.45d;
    decimal(11,5) dec_2=-123456.12345d;
    float f1,f2;
    f1=dec_1;
    f2=dec_2;
    printf("f1=%f\nf2=%f\n\n", f1,f2);    /* f1=123.449997    */
                                          /* f2=-123456.125000 */
}

```

Figure 257. ILE C Source to Convert a Packed Decimal to a Floating Point

The output is as follows:

```
f1=123.449997
f2=-123456.125000
Press ENTER to end terminal session.
```

Overflow Behavior

The following table describes the overflow behavior when a packed decimal number is assigned to a smaller target. An exception is not generated when:

- A packed decimal is assigned to a smaller target with integral type.
- A packed decimal is assigned to a smaller target of floating point type.

An exception is generated when a packed decimal is assigned to a smaller packed decimal target. You can suppress runtime errors by using the `#pragma nosigtrunc` directive in your ILE C source code.

From Field	To Field	Runtime Error
Packed Decimal	char, int, short, long, long long, bit	No
Packed Decimal	Packed Decimal	Yes
Packed Decimal	Float	No ²
Packed Decimal	Double	No ^{1, 2}

Note:

1. There is no packed decimal number large enough to cause overflow when the packed decimal is assigned to a double.
2. If you use the MI instruction `setca` to unmask a floating point exception, you receive an error message MCH1213 for a floating point inexact result.

Passing Packed Decimal Data to a Function

There are no default argument promotions on arguments that have packed decimal type when the called function does not include a prototype. This means that any function definition that contains packed decimal arguments has to be prototyped. Otherwise the behavior is undefined. The boundary alignment of an argument with packed decimal type depends on the size of the packed decimal type. To be specific:

- $1 \leq n \leq 7$ aligns on a 4-byte boundary
- $8 \leq n \leq 15$ aligns on a 8-byte boundary
- $16 \leq n \leq 31$ aligns on a 16-byte boundary
- $32 \leq n \leq 63$ aligns on a 32-byte boundary

The following example shows how to pass packed decimal variables to a function.

```
#include <decimal.h>
#include <stdio.h>
decimal(3,1)  d1 = 33.3d;
decimal(10,5) d2 = 55555.55555d;
decimal(28)   d3 = 88888888888888888888888888888888d;
void func1( decimal(3,1), decimal(10,5),
            decimal(10,5), decimal(28));

int main(void) {
    func1(d1, d2, 999.99d, d3); /* with prototype */
                                /* The arguments are passed as followed*/
                                /* 333f0000 00000000 05555555 555f0000 */
                                /* 99999f00 00000000 00000000 00000000 */
                                /* 08888888 88888888 88888888 88888f00 */
}
/* func1 is prototyped */
void func1(decimal(3,1) x1, decimal(10,5) x2,
           decimal(10,5) x3, decimal(28) x4) {
    /* no runtime error when referencing x1, x2, x3 or x4 */
    printf("x1 = D(3,1)\n", x1);
    printf("x2 = D(10,5)\n", x2);
    printf("x3 = D(10,5)\n", x3);
    printf("x4 = D(28)\n", x4);
}
```

Figure 258. ILE C Source to Pass Packed Decimal Variable to a Function

The output is as follows:

```
x1 = 33.3
x2 = 55555.55555
x3 = 999.99
x4 = 88888888888888888888888888888888
```

Passing a Pointer to a Packed Decimal Variable to a Function

The following example shows you how to pass a pointer to a packed decimal variable to a function.

```

/* This program shows how to pass a pointer to a packed decimal      */
/* variable to a function.                                          */
#include <decimal.h>
#include <stdio.h>
    decimal(5,2) var=123.45d;
    decimal(5,2) *p=&var;
    decimal(5,2) *func_1(decimal(5,2) *);

int main(void)
{
    /* Call function with pointer to packed decimal argument. The    */
    /* value that it returns is also a pointer to a packed decimal.  */
    if(func_1(p)!=p)
    {
        printf("Function call not successful\n\n");
    }
    else
    {
        printf("The packed decimal number is: %D(5,2)\n",*func_1(p));
    }
}
decimal(5,2) *func_1(decimal(5,2) *q)
{
    return q;
}

```

Figure 259. ILE C Source to Pass a Pointer to a Packed Decimal Value to a Function

The packed decimal argument in the function call has to be the same type as the packed decimal in the function prototype. If overflow occurs in a function call with packed decimal arguments, no error or warning is issued during compilation, and a runtime exception is generated.

The output is as follows:

```

The packed decimal number is: 123.45
Press ENTER to end terminal session.

```

Calling Another Program that Contains Packed Decimal Data

You can pass packed decimal arguments with interlanguage calls to RPG or COBOL.

Example:

The following example shows an ILE C program that calls an OPM COBOL program and then passes a packed decimal data.

```

#include<stdio.h>
#include <decimal.h>
void CBLPGM(decimal(9,7));
#pragma datamodel(p128)
#pragma linkage(CBLPGM,OS)
#pragma datamodel(pop)
int main(void)
{
    decimal(9,7) arg=12.1234567d;
    /* Call an OPM COBOL program and pass a packed                    */
    /* decimal argument to it.                                        */
    CBLPGM(arg);
    printf("The COBOL program was called and passed a packed decimal value\n");
}

```

Figure 260. ILE C Source for an ILE C Program that Passes Packed Decimal Data

The following example shows COBOL source for passing a packed decimal variable to an ILE C program.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. CBLPGM.
*****
* Packed decimals: This is going to be called by an ILE C          *
* program to pass packed decimal data.                             *
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
77 PAC-DATA PIC X(30) VALUE "PROGRAM START".
77 PACK-IN-WS PIC 99.9999999.
LINKAGE SECTION.
01 PACK-DATA PIC 9(2)V9(7) PACKED-DECIMAL.
PROCEDURE DIVISION USING PACK-DATA.
MAIN-LINE SECTION.
MOVE PACK-DATA TO PACK-IN-WS.
DISPLAY "**** PACKED DECIMAL RECEIVED IS: " PACK-IN-WS.
GOBACK.

```

Figure 261. COBOL Source that Receives Packed Decimal Data from an ILE C Program

The output is as follows:

```

**** PACKED DECIMAL RECEIVED IS: 12.1234567
Press ENTER to end terminal session.

```

```

The COBOL program was called and passed a packed decimal value.
Press ENTER to end terminal session.

```

Using Library Functions with a Packed Decimal Data Type

You can use the `va_arg` macro to accept a packed decimal data of the form (n,p) . You can write packed decimal constants to a file, and scan them back.

The following figure shows you how to use the `va_arg` macro to accept a packed decimal data of the form $\text{decimal}(n,p)$. The `va_arg` macro returns the current packed decimal argument.

```

/* This program uses the va_arg macro to accept a static decimal    */
/* data type of the form decimal(n,p). The va_arg macro returns    */
/* the current packed decimal argument.                             */
#include <decimal.h>
#include <stdio.h>
#include <stdarg.h>
#define N1 3
#define N2 6
int vargf(FILE *,char *,...);
int main(void)
{
    int num1 = -1, num2 = -1;
    char fmt_1[]="%D(3,2)%D(3,2)%D(3,2)";
    char fmt_2[]="%D(3,2)%D(3,2)%D(3,2)%D(3,2)%D(3,2)%D(3,2)";
    decimal(3,2) arr_1[]={ 1.11d, -2.22d, 3.33d };
    decimal(3,2) arr_2[]={ -1.11d, 2.22d, -3.33d, 4.44d, -5.55d, 6.66d};
    FILE *stream_1;
    FILE *stream_2;
    stream_1=fopen("file_1.dat", "wb+");
    num1=vargf(stream_1,fmt_1,arr_1[0],
               arr_1[1],
               arr_1[2]);

    if (num1<0)
    {
        printf("An error occurred when calling function vargf first time\n");
    }
    else
    {
        printf("Number of char. printed when vargf is called first time is:%d\n",
              num1);
    }
}

```

```

stream_2=fopen("file_2.dat", "wb+");

num2=vargf(stream_2,fmt_2,arr_2[0],
           arr_2[1],
           arr_2[2],
           arr_2[3],
           arr_2[4],
           arr_2[5]);

```

```

if (num2<0)
{
    printf("An error occurred when calling function vargf second time\n");
}
else
{
    printf("Number of char. printed when vargf is called a second time is:%d\n",
          num2);
}
fclose(stream_1);
fclose(stream_2);
}
int vargf(FILE *str, char *fmt,...)
{
    int result;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    result = vfprintf(str, fmt, arg_ptr);
    va_end(arg_ptr);
    return result;
}

```

The output is as follows:

```

Number of char. printed when vargf is called first time is: 13
Number of char. printed when vargf is called second time is : 27
Press ENTER to end terminal session.

```

The following example shows you how to write packed decimal constants to a file, and how to scan them back. In addition, the example shows you how to pass a packed decimal array to a function.

```

/* This program shows how to write packed decimal constants to a file */
/* and scan them back again. Also shows how to pass a packed decimal */
/* array to a function. */
#include <decimal.h>
#include <stdio.h>
#include <stdlib.h>
#define N      3                /* Array size */
                                /* for decimal declaration. */
FILE *stream;                 /* File pointer declaration. */
                                /* Declare valid packed decimal */
                                /* array. */
decimal(4,2) arr_1[] = {12.35d, 25.00d, -19.58d};
decimal(4,2) arr_2[N];
void write_num(decimal(4,2) a[N]); /*Declare function to */
                                /*write to a file. */
void read_num(decimal(4,2) b[N]); /*Declare function to */
                                /*read from a file. */

int main(void)
{
    int reposition=0;
                                /* Open the file. */
    if ((stream = fopen("CURLIB/OUTFILE","w+")) == NULL)
    {
        printf("Can not open file");
        exit(EXIT_FAILURE);
    }
    write_num(arr_1);           /* Call function to write */
                                /* values of packed decimal */
                                /* array to outfile with fprintf*/
                                /* library function. */
    reposition=fseek(stream, 0L, SEEK_SET);
    if (reposition!=0)
    {
        printf("FSEEK failed to position file pointer\n");
    }
}

```

```

        exit(EXIT_FAILURE);
    }
    read_num(arr_2);                /* Call function to read      */
                                   /* values of packed decimal  */
                                   /* array from file using     */
                                   /* fscanff() function.       */
    fclose(stream);                /* Close the file.          */
}
/* write_num is passed a packed decimal array. These values are
/* written to a text file with the fprintf library function.
/* If the function is successful a 0 is returned, otherwise a
/* negative value is returned (indicating an error).
*/

void write_num(decimal(4,2) a[N])
{
    int i, j;

    for (i=0;i < N;i++)
    {
        j = fprintf(stream,"%D(4,2)\n",a[i]);
        if (j < 0)
            printf("Number not written to file %D(4,2)\n",a[i]);
    }
}
/* read_num is passed a packed decimal array. The values are
/* read from a text file with the fscanff library function.
/* If the function is successful a 0 is returned, otherwise a
/* negative value is returned (indicating an error).
*/
void read_num(decimal(4,2) b[N])
{
    int i, j;
    for (i=0;i < sizeof(b)/sizeof(b[0]);i++)
    {
        j = fscanff(stream,"%D(4,2)\n",&b[i]);
        if (j < 0)
            printf("Error when reading from file\n");
    }
    printf("b[0]=%D(4,2)\nb[1]=%D(4,2)\n\
        b[2]=%D(4,2)\n", b[0], b[1], b[2]);
}

```

The output is as follows:

```

b[0]=12.35
b[1]=25.00
b[2]=-19.58
Press ENTER to end terminal session.

```

The following example shows how to use the %D(*,*) specifier with the printf() function. If n and p of the variable to be printed do not match with the n and p in the conversion specifier %D(n,p), the behavior is undefined. Use the unary operators digitsof (expression) and precisionof (expression) in the argument list to replace the * in D(*,*) whenever the size of the resulting type of a packed decimal expression is not known.

```

#include <decimal.h>
#include <stdio.h>
int main(void)
{
    decimal(6,2) op_1=1234.12d;
    decimal(10,2) op_2=-12345678.12d;
    printf("op_1 = %*. *D(*,*)\n", 6, 2, digitsof(op_1),
        precisionof(op_1), op_1);
    printf("op_2 = %*. *D(*,*)\n", 10, 2, digitsof(op_2),
        precisionof(op_2), op_2);
}

```

Figure 262. ILE C Source to Print Packed Decimal Constants

The output is as follows:

```

op_1 = 1234.12
op_2 = -12345678.12
Press ENTER to end terminal session.

```

Understanding Packed Decimal Data Type Errors

C

All the warning and messages for packed decimal data errors are issued by the ILE C compiler during compilation, unless explicitly stated as occurring at runtime. If you receive a warning during compilation for overflow, an exception may be generated at runtime; SIGFPE is raised.

Runtime errors occur during the following packed decimal operations:

- Assignment
- Casting
- Initialization
- Arithmetic operations
- Function calls

Note: Some of the overflow situations are signaled during compilation through a warning; loss of digits may occur.

Packed Decimal Warnings and Error Conditions

C

The following figure shows all the warnings and error conditions that are issued by the ILE C compiler for packed decimal expressions.

```
#include <decimal.h>
decimal (999,99) s1 = 1234.56d; /* Generates a severe error because */
/* the decimal size is greater than 63. */
decimal (10, 64) s2 = 1234.56d; /* Generates a sever error because */
/* the precisionsize exceeds 63. */
decimal (1.2, 3) s3 = 1.2d; /* Generates a severe error because */
/* the decimal size is an invalid number. */
decimal (2,1.2) s4 = 1.3d; /* Generates a severe error because the */
/* precision size is an invalid number. */
decimal (1,3) s5 = 1.345d; /* Generates a severe error because the */
/* precision size exceeds the decimal size. */
decimal (63,62) s6
= 123456789012345678901234567890123456789012345678901234567890123456789d;
/* Generates a severe error because decimal */
/* constant is out of range of a valid */
/* packed decimal constants. */
decimal(10,2) s7 = 123456789012345678901234567890123456789012345678901234567890123456789.12345d
+ 12345.1234567891d;
/* s7 = (63,5) + (15,10) */
/* Generates a severe error because of */
/* truncation of precision values in */
/* intermediate result. */
decimal (10,2) s8 = 123456789012345678901234567890.12345d
* 12345678901234567890123456.12d;
/* s8 = (35,5) + (28,2) */
/* Generates a severe error because of */
/* truncation of precision values in */
/* intermediate result. */
decimal (10,2) s9 = 1234567890123456789012345678901234567890d
/ 1234567890.123456789012345678901234567891d;
/* Generates a warning for possible loss of */
/* whole-digit data in intermediate value. */
/* Generates a severe error for loss of */
/* whole-digit data loss in result. */
```

Figure 263. Packed Decimal Warnings and Error Conditions

Note:

- For assignments to a target field that is too small to hold the packed decimal number, the ILE C compiler issues no warning or error for static, external or automatic initialization.

- For expressions requiring decimal point alignment (namely addition, subtraction or comparison), the ILE C compiler issues errors for static, external, and automatic initialization if, during alignment, the maximum number of allowed digits is exceeded.
- For multiplication, if the evaluation of the expression results in a value larger than the maximum number of allowed digits:
 - A compile time error is issued for static or external initialization; no module is created.
 - A compile time warning is issued if the expression is within a function; a runtime exception is generated.
 - Truncation occurs on the fractional part, preserving as much of the integral part as possible.
- For division, a compile time error is generated when $((n_1 - p_1) + p_2) > 63$.

The *ILE C/C++ Language Reference* contains information on the multiplication and division operators for packed decimals.

Suppressing a Runtime Overflow Exception

C

You can use the `#pragma nosigtrunc` directive to suppress a runtime exception that occurs as a result of overflow. The *ILE C/C++ Compiler Reference* contains information on the `#pragma nosigtrunc` directive.

The following figure shows how to suppress the runtime exception created when a packed decimal variable overflows on assignment, in a function call, and in an arithmetic operation.

```

/* This program shows how to suppress a runtime exception when a      */
/* packed decimal variable overflows on assignment, in a function call */
/* and in an arithmetic operation.                                     */
#include <decimal.h>
#pragma nosigtrunc                                                    /* The directive turns off */
                                                                    /* SIGFPE which is raised  */
                                                                    /* in the following overflow */
                                                                    /* situations; no exception */
                                                                    /* occurs at runtime.      */

void f(decimal(4,2) a)
{
}
int main(void)
{
    decimal(8,4)  arg=1234.1234d;
    decimal(5,2) op_1=1234567.1234567d; /* Overflow in initialization.*/
    decimal(2)   op_2;
    decimal(20,5) op_3=12.34d;
    decimal(15,2) op_4=1234567890.12d;
    decimal(6,2) op_5=1234.12d, cast;
    decimal(31,2) res;
    cast=(decimal(2))op_5; /* Overflow in casting. */
    op_2=arg; /* Overflow in assignment. */
    f(arg); /* Overflow in function call. */
    res=op_3*op_4; /* Overflow in arithmetic */
                /* operation. */
}

```

Figure 264. ILE C Source to Suppress a Runtime Exception

Note: No runtime exception is logged in the job log.

Using Packed Decimal Data in a C++ Program

C++

The packed decimal data type representation includes integral and fractional parts. For ILE C++, the largest packed decimal representation is 63 digits.

Note: In IBM i 7.1 and earlier the largest packed decimal representation is 31 digits.

The ILE C++ compiler supports the packed decimal data type in the `bcd` class. The header file is `<bcd.h>`.

Note: The ILE C compiler supports the packed decimal data type as an extension to ISO C. For more information, refer to [“Using Packed Decimal Data in a C Program”](#) on page 354.

You can use the packed decimal data type to:

- Represent large numeric quantities accurately, especially in business and commercial applications for financial calculations.

Note: You do not have to use floating point arithmetic. Floating point is more suitable for scientific and engineering computations, which often use numbers that:

- Are much larger than the largest packed decimal variable can store
 - Are much smaller than the smallest packed decimal, but do not have enough precision for commercial use
- Declare type definitions, arrays, structures, and unions that have packed decimal members. You can apply unary operators on packed decimal variables. Bitwise operators do not apply to packed decimal data. For more information on the packed decimal data type, see the *ILE C/C++ Language Reference*.

This topic describes:

- [The IBM i Binary Coded Decimal \(BCD\) header file](#)
- [Using the `_DecimalT` class template](#)
- [Conversions to other data types](#)
- [C++ exception handling with the `_DecimalT` class template](#)
- [Passing a `_DecimalT` class template object to a function](#)
- [Passing a pointer to a `_DecimalT` class template object](#)
- [Calling another program containing a `_DecimalT` class template](#)
- [Writing `_DecimalT` class template constants to a file](#)

The IBM i Binary Coded Decimal (BCD) Header File

The class and function template definitions for the C++ `_DecimalT` class template and the numerical limits of a `_DecimalT` class template are defined inside the header file `<bcd.h>`. See [Table 34 on page 366](#).

Any C++ source file that uses the `_DecimalT` class template must include the `bcd.h` header file. The `#include <bcd.h>` statement must appear before any use of the `_DecimalT` class template.

Constant Name	Description
<code>DEC_DIG</code>	The maximum number of significant digits that the <code>_DecimalT</code> class template can hold in IBM i 7.1 and earlier.
<code>DEC_PRECISION</code>	The maximum number of decimal places that the <code>_DecimalT</code> class template can hold in IBM i 7.1 and earlier.
<code>DEC63_DIG</code>	The maximum number of significant digits that the <code>_DecimalT</code> class template can hold as of IBM i 7.2.

Table 34. Constants Defined in `bcd.h` (continued)

Constant Name	Description
DEC64_PRECISION	The maximum number of decimal places that the <code>_DecimalT</code> class template can hold as of IBM i 7.2.
DFT_DEC_DIG	Set to DEC_DIG when IBM i 7.1 or earlier is targeted, DEC63_DIG otherwise.
DFT_DEC_PRECISION	Set to DEC_PRECISION when IBM i 7.1 or earlier is targeted, DEC63_PRECISION otherwise.
DEC_INT_DIG	The number of significant digits of a binary coded decimal object when you convert it to an integer type. The value 10 is stored in DEC_INT_DIG.
DEC_INT_PREC	The number of decimal places of a binary coded decimal object when you convert it from an integer type. The value 0 is stored in DEC_INT_PREC.

Using the `_DecimalT` Class Template

When you are working with ILE C++, the `_DecimalT` class template allows representation of up to 63 significant digits, including integral and fractional parts. The fractional part of a dollar can be represented accurately by two digits following the decimal point.

Note: In IBM i 7.1 and earlier the largest packed decimal representation is 31 digits.

Note: You do not have to use floating point arithmetic. Floating point is more suitable for scientific and engineering computations, which often use numbers that:

- Are much larger than the largest packed decimal variable can store
- Are much smaller than the smallest packed decimal, but do not have enough precision for commercial use

The same declarations and operators that you use on other data types, such as float, can be applied to `_DecimalT` class templates, with the exception of unions and bitwise operators that do not apply to `_DecimalT` class templates.

You can:

- Declare type definitions, arrays, and structures that have `_DecimalT` class templates.
- Apply arithmetic, relational, assignment, comma, conditional, equality, logical, and unary operators on the `_DecimalT` class template.
- Pass `_DecimalT` class templates in function calls. The `_DecimalT` class template is compatible with packed decimal representations in ILE languages.
- Define macros, and call library functions with `_DecimalT` class templates.
- View the `_DecimalT` class template when you use the ILE system debugger.

Note: When you view a `_DecimalT` class template, none of the operators are accessible from the debugger. For information on the ILE system debugger, see [“Debugging Programs” on page 89](#).

Declaring `_DecimalT` Class Template Objects

To declare an object as a `_DecimalT` class template:

```
_DecimalT<10,2> x;
_DecimalT<5,0> z;
_DecimalT<18,10> *ptr;
_DecimalT<8,2> arr[100];
```

Note:

1. The variable `x` can have values from `__D("-99999999.99")` to `__D("+99999999.99")`
2. The variable `z` can have values from `__D("-99999")` to `__D("+99999")`
3. `ptr` is a pointer to an object of type `_DecimalT<18,10>`
4. `arr` is an array of 100 elements, where each element is of class `_DecimalT<8,2>`
5. Leading zeros show the size of the number of digits in the `_DecimalT` class templates. You do not need to enter leading zeros in your `_DecimalT` class templates.

Using the `__D` Macro to Simplify Code

Use the `__D` macro to simplify code that requires the frequent use of the `_ConvertDecimal` constructor:

```
_ConvertDecimal(char *);
```

You can initialize a `_DecimalT` class template with a `_ConvertDecimal` object:

```
_DecimalT<5,2> x = __D("123.45");
_DecimalT<DEC_DIG, DEC_PRECISION> a = __D(".000000000000000000000000000005");
_DecimalT<6,2> b[] = {__D("1.2"), __D("2"), __D("1234.56"), __D("-3.3")};
_DecimalT<28,20> a = __D("-12.123456789");
```

Note:

The string must contain only the following characters:

```
0 1 2 3 4 5 6 7 8 9 . - +
```

`_DecimalT` Class Template Input and Output

Both the standard (no extension) and USL (.h extension) headers can be used for input/output of `_DecimalT` objects. To use the C `<stdio>` formatted input/output functions, specify the `%D(n,p)` type format used for the C language internally defined decimal type.

Assuming that you are using header files with a .h extension:

- Use the `cout` pre-defined stream to print a `_DecimalT` class template value.
- Use the `cin` pre-defined stream to read a `_DecimalT` class template value.

Note: If you are using header files without an extension (such as `<stdio>`), use the `stdout` and `stdin` pre-defined streams.

To print the value of a `_DecimalT` class template, you can use the `fprintf()`, `printf()`, `sprintf()`, `vfprintf()`, `vprintf()`, or `vsprintf()` functions.

To read the value of a `_DecimalT` class template, you can use the `fscanf()`, `scanf()`, or `sscanf()` functions.

Using Operators with the `_DecimalT` Class Template

You can use the following operators with the `_DecimalT` class template:

- Arithmetic (see [“Using Arithmetic Operators with the `_DecimalT` Class Template” on page 369](#))
- Assignment
- Comma
- Conditional (see [“Using Conditional Expressions with the `_DecimalT` Class Template” on page 370](#))
- Equality (see [“Using Equality Operators with the `_DecimalT` Class Template” on page 370](#))
- Relational (see [“Using Relational Operators with the `_DecimalT` Class Template” on page 369](#))
- Unary (see [“Using Unary Operators with the `_DecimalT` Class Template” on page 371](#))

Note:

1. Logical operators are not implemented for `_DecimalT` class templates.
2. For information on runtime exceptions during assignments, see [“C++ Packed Decimal Data Conversions”](#) on page 372.

Using Arithmetic Operators with the `_DecimalT` Class Template

Figure 265 on page 369 shows how to:

- Define operands and results
- Perform arithmetic operations on operands
- Direct the output to the standard output stream `cout`

```
#include <bcd.h>           // bcd Class Header File
#include <iostream.h>

int main()
{
    _DecimalT<10,2> op_1 = __D("12");
    _DecimalT<5,5> op_2 = __D("-.12345");
    _DecimalT<24,12> op_3 = __D("12.34");
    _DecimalT<20,5> op_4 = __D("11.01");

    _DecimalT<14,5> res_add;
    _DecimalT<25,2> res_sub;
    _DecimalT<15,7> res_mul;
    _DecimalT<31,14> res_div;

    res_add = op_1 + op_2;
    res_sub = op_3 - op_1;
    res_mul = op_2 * op_1;
    res_div = op_3 / op_4;

    cout <<"res_add =" <<res_add <<endl;
    cout <<"res_sub =" <<res_sub <<endl;
    cout <<"res_mul =" <<res_mul <<endl;
    cout <<"res_div =" <<res_div <<endl;

}
```

Figure 265. Example: Arithmetic Operators for the `_DecimalT` Class Template

The output is:

```
res_add =11.87655
res_sub =0.34
res_mul =-1.4814000
res_dev =1.12079927338782
```

Using Relational Operators with the `_DecimalT` Class Template

When you use relational operators with the `_DecimalT` class template, consider the following:

- You can use the relational expression *less than* (`<`) for `_DecimalT` class templates and compare `_DecimalT` class templates with other arithmetic types (that is, integer, float, double, and long double).
- Implicit conversions are performed using the arithmetic conversion rules.

Figure 266 on page 370 shows how to use relational operators with the `_DecimalT` class template.

```

#include <bcd.h>
#include <iostream.h>

    _DecimalT<10,3> pdval = __D("0000023.423"); // bcd declaration
    int ival = 1233.1; // Integer declaration
    float fval = 1234.34f; // Float declaration
    double dval = 251.5832; // Double declaration
    long double lval = 37486.234; // Long double declaration

main( )
{
    _DecimalT<15,6> value = __D("000485860.085999");
    // Perform relational operation between other data types and
    // bcd class

    if (pdval < ival) cout <<"pdval is the smallest !" <<endl;
    if (pdval < fval) cout <<"pdval is the smallest !" <<endl;
    if (pdval < dval) cout <<"pdval is the smallest !" <<endl;
    if (pdval < lval) cout <<"pdval is the smallest !" <<endl;
    if (pdval < value) cout <<"pdval is the smallest !" <<endl;
}

```

Figure 266. Example: Relational Operators and the `_DecimalT` Class Template

The output is:

```

pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !

```

Using Conditional Expressions with the `_DecimalT` Class Template

Figure 267 on page 370 shows how to:

- Define operands and results
- Use the operands to create conditional expressions
- Direct the output to the standard output stream `cout` only if the result satisfies the specified condition

```

#include <bcd.h>
#include <iostream.h>
int main ( )
{
    _DecimalT<10,2> x, y, z;
    x = __D("1.20");
    y = __D("01.2");
    z = (x==y)? __D("9.9"):__D("2.45");
    if (z== __D("9.9"))
    {
        cout <<"x equals y" <<endl;
    }
}

```

Figure 267. Example: Conditional Operators and the `_DecimalT` Class Template

The output is:

```

x equals y

```

Using Equality Operators with the `_DecimalT` Class Template

Figure 268 on page 371 shows how to:

- Define operands and results
- Perform equality operations on the operands
- Direct the output to the standard output stream `cout`

```

#include <bcd.h>
#include <iostream.h>

_DecimalT<1, 0>   op_1 = __D("+0");      // Declare and initialize
_DecimalT<1, 0>   op_2 = __D("-0");      // valid BCD
_DecimalT<9,4>   op_3 = __D("00012.3400");
_DecimalT<4,2>   op_4 = __D("12.34");

int main(void)    // These statements
{                // perform equality <==> test
                // on the above variable
    if (op_1 == op_2) // declarations
    {
        cout <<"op_1 equals op_2"<<endl;
    }

    if (op_3 != op_4)
    {
        cout <<"op_3 not equals op_4"<<endl;
    }
    else
    {
        cout <<"op_3 equals op_4"<<endl;
    }
}

```

Figure 268. Example: Equality Operators and the `_DecimalT` Class Template

The output is:

```

op_1 equals op_2
op_3 equals op_4

```

Using Unary Operators with the `_DecimalT` Class Template

A *unary expression* contains one operand and a unary operator. All *unary operators* have the same precedence and have right-to-left associativity. For information about overloading unary operators, see the *ILE C/C++ Language Reference*.

Figure 269 on page 371 shows how to:

- Define operands
- Perform unary operations on the operands
- Direct the output to the standard stream `cout`

```

#include <iostream.h>
#include <bcd.h>
int main()
{
    _DecimalT<10,2> op_1 = __D("12");
    _DecimalT<5,5> op_2 = __D("-.12345");
    _DecimalT<24,12> op_3 = __D("12.34");
    _DecimalT<20,5> op_4 = __D("11.01");
    cout << "op_1++ => " << op_1++ << endl;
    cout << "op_1 after increment => " << op_1 << endl;
    cout << "-op_2 => " << -op_2 << endl;
    cout << "--op_3 => " << --op_3 << endl;
    cout << "+op_4 => " << +op_4 << endl;
}

```

Figure 269. Example: Unary Operators and the `_DecimalT` Class Template

The resulting output is:

```

op_1++ => 12.00
op_1 after increment => 13.00
-op_2 => 0.12345
--op_3 => 11.34000000000000
+op_4 => 11.01000

```

C++ Packed Decimal Data Conversions

If the value of the packed decimal type to be converted is within the range of values that can be represented exactly, the value of the packed decimal type is not changed.

Packed decimal values are compatible if their types are the same. For example, `decimal(n1, p1)` and `decimal(n2, p2)` have compatible types if and only if

```
((n1 == n2) && (p1 == p2))
```

Converting Values from One `_DecimalT` Class Template to Another

If the value of a `_DecimalT` class template that is to be converted to another `_DecimalT` class template is not within the range of values that can be represented exactly, the value of the `_DecimalT` class template to be converted is truncated, as shown in the following figure.

```
_DecimalT<4,2> targ_1, targ_2;
_DecimalT<6,2> op_1=__D("1234.56"), op_2=__D("12.34");

targ_1 = op_1;    // A runtime exception is generated because the integral
                // part is truncated; targ_1=__D("34.56").
targ_2 = op_2;    // No runtime exception is generated because neither the
                // integral nor the fractional part is truncated;
                // targ_2=__D("12.34").
```

Figure 270. Example of Converting a Value from One `_DecimalT` Class Template to Another

Note:

1. A runtime exception occurs on assignment to a smaller target only when the integral part is truncated. If assignment causes truncation in the integral part, then there is a runtime exception in which a `_DecErrDigTruncated` object is thrown. This runtime exception occurs when an integral value is lost during conversion to a different type, regardless of what operation requires the conversion. For information on runtime exceptions during conversions, see “[_DecimalT Class Template Runtime Exceptions](#)” on page 375.
2. If truncation occurs in the fractional part, there is no runtime exception.

When one `_DecimalT` class template is assigned to another `_DecimalT` class template with a smaller precision, the result is truncation of the fractional part, as shown in the following figure.

```
_DecimalT<7,4> x = __D("123.4567");
_DecimalT<7,1> y;

y = x;    // y = __D("123.4")
```

Figure 271. Example of Conversion from One `_DecimalT` Class Template to Another with Smaller Precision

Converting Values from a `_DecimalT` Class Template to an Integer Data Type

When a value of a `_DecimalT` class template is converted to an integer type:

- A direct call is made to `CPYNV`.
- If the number of digits exceeds what can be represented in an integer, `CPYNV` first converts the packed decimal value to a large integer, then truncates the hexadecimal value of that integer.

Note: No runtime exception occurs when assigning a `_DecimalT` class template to an integer type that results in truncation of the integral part.

```
int op;
_DecimalT<7,2> op1 = __D("12345.67");
op = op1;           // Truncation on the fractional
                   // part. op=12345
```

Figure 272. Example of Converting to an Integer Type from a `_DecimalT` Class Template with a Fractional Part

```
int op;
_DecimalT<3, 0> op2 = __D("123");
op = op2;          // No truncation; op=123
```

Figure 273. Example of Converting to an Integer Type from a `_DecimalT` Class Template with Less than 10 Digits in the Integral Part

```
int op2;
_DecimalT<12, 0> op3;
op3 = __D("123456789012");
op2 = op3;         // Truncation occurs on the integral
                   // part 123456789012 (0x1CBE991A14).
                   // op2= 3197704724 (0xBE991A14); no runtime
                   // exception.
```

Figure 274. Example of Converting to an Integer Type from a `_DecimalT` Class Template with More than 10 Digits in the Integral Part

```
#include <bcd.h>

int op;
_DecimalT<15,2> op_1 = __D("1234567890123.12");
op = op_1;         // Truncation occurs on the integral and
                   // fractional parts 1234567890123 (0x11F71FB04CB).
                   // op=1912276171 (0x71FB04CB) ; no run-time exception.
```

Figure 275. Example of Converting to an Integer Type from a `_DecimalT` Class Template with More than 10 Digits in the Integral Part and a Fractional Part

Converting Values from a `_DecimalT` Class Template to a Floating Point Data Type

To convert a `_DecimalT` class template class to a floating point data type, use source code similar to that found in the following figure.

```
#include <bcd.h>
#include <iostream.h>

int main(void)
{
    _DecimalT<5,2> dec_1=__D("123.45");
    _DecimalT<11,5> dec_2=__D("-123456.12345");

    float f1,f2;

    f1=dec_1;
    f2=dec_2;

    cout <<"f1=" <<f1 <<endl <<"f2=" <<f2 <<endl <<endl; //f1=123.45
                                                    // f2=-123456
}
}
```

Figure 276. Example of Converting a `_DecimalT` Class Template to a Floating Point Data Type

The output is shown below.

```
f1=123.45
f2=-123456
```

Determining the Size of a `_DecimalT` Class Template

When you use the `sizeof` operator with `_DecimalT`, you can find out the total number of bytes occupied by the `_DecimalT` class template.

Note: Each `_DecimalT` class template digit occupies half a byte. Half a byte is used for the sign. The number of bytes used by `_DecimalT` is the smallest whole number greater than or equal to $(n + 1)/2$ (for example, `sizeof(_DecimalT) = ceil((n + 1)/2)`).

```
int y;
_DecimalT <5, 2> x;
y =sizeof(x);          // This would be calculated to be 3 bytes
                      // (5+1)/2 = 3.
```

Figure 277. Example of Determining the Total Number of Bytes Occupied by a `_DecimalT` Class Template

Determining the Number of Digits in a `_DecimalT` Class Template

When you use the member function `DigitsOf()` with a `_DecimalT` class template, you can find out the total number of digits `n` in a `_DecimalT` class template.

```
int n,n1;
_DecimalT <5, 2> x;
n = x.DigitsOf();      // the result is n=5
```

Figure 278. Example of Determining the Number of Digits in a `_DecimalT` Class Template

Determining the Precision of a `_DecimalT` Class Template

When you use the member function `PrecisionOf()` with a `_DecimalT` class template, you can determine the number of decimal digits `p` in a `_DecimalT` class template, as shown in the following figure.

```
int p,p1;
_DecimalT <5, 2> x;
p=x.PrecisionOf();    // The result is p=2
```

Figure 279. Example of Determining the Number of Decimal Digits `p` of a `_DecimalT` Class Template

How Overflows Are Handled

Table 35 on page 374 describes the overflow behavior when a `_DecimalT` class template is assigned to a smaller target.

An exception is generated when a `_DecimalT` class template is assigned to a `_Decimal` class template with a smaller target.

An exception is not generated when:

- A `_DecimalT` class template is assigned to a `char`, `int`, `short`, `long`, or bit field with a smaller target
- A `_DecimalT` class template is assigned to a floating point with a smaller target

From Type	To Type	Runtime Exception?
<code>_DecimalT</code> class template	<code>char</code> , <code>int</code> , <code>short</code> , <code>long</code> , <code>bit</code>	No
<code>_DecimalT</code> class template	<code>_DecimalT</code> class template	Yes
<code>_DecimalT</code> class template	<code>float</code>	No ¹

Note: ¹There is no `_DecimalT` class template large enough to cause overflow when the `_DecimalT` class template is assigned to a `double`.

Using C++ Exception Handling with the `_DecimalT` Template

This section describes runtime exceptions for `_DecimalT` template classes, error classes, and debug macros when C++ exception handling is used with the `_DecimalT` template class template. If an error is detected during runtime, an error object is thrown.

This section describes:

- [_DecimalT class template runtime exceptions](#)
- [Using a class derived from the `_DecErr` class](#)
- [Using debug macros](#)

`_DecimalT` Class Template Runtime Exceptions

In ILE C, a packed decimal is implemented as a native data type. This approach allows an error, such as a decimal format that is not valid, to be detected at compile time. In ILE C++, detection of a similar error is deferred until runtime.

```
#include <bcd.h>
void main()
{
    _DecimalT<10,20> b = __D("ABC"); // Runtime exception is raised
}

#include <bcd.h>
void main()
{
    _DecimalT<65,2> a;                // Max. dig. allow is 63. Again,
                                     // runtime exception is raised
}
```

Figure 280. `_DecimalT` Class Template Runtime Exceptions

When Runtime Exceptions Occur

Runtime exceptions may occur during the following operations:

- Arithmetic operations
- Assignment
- Casting
- Function calls
- Initialization

Note: Overflow situations that occur during compilation are deferred until runtime; loss of digits may occur.

Runtime Exceptions Issued by the Compiler for `_DecimalT` Class Templates

The runtime exceptions issued by the compiler for `_DecimalT` class templates are shown in the following figure.

```

#include <bcd.h>

static _DecimalT<5,2> s1 = __D("12345678.0");
static _DecimalT<10,2> s2 = __D("1234567891234567891234567.12345")
                          + __D("12345.1234567891");
                          // s2 = (31,5) + (15,10)
static _DecimalT<10,2> s3 = __D("1234567891234562345")
                          * __D("1234567891234.12");
                          // s3 = (19,0) * (15,2)
static _DecimalT<10,2> s4 = __D("12345678912345678912") /
                          __D("12345.123456789123456");
                          // s4 = (20,0) / (20,15)

int main(void)
{
    _DecimalT<5,2> a1 = __D("12345678.0");
    _DecimalT<10,2> a2, a3, a4;
    _DecimalT<5,2> a5 = (_DecimalT<5,2>) __D("123456.78");

    a2 = __D("1234567891234567891234567.12345") + __D("12345.1234567891");
                          // a2 = (31,5) + (15,10)
    a3 = __D("123456789123456.12345") * __D("1234567891234.12");
                          // a3 = (20,5) * (15,2)
                          // expression.
                          // Note: Need (35,7) but
                          // use (31,2), for example,
                          // keep the integral part.
    a4 = __D("12345678912345678912") / __D("12345.123456789123456");
                          // a4 = (20,0) / (20,15)
                          // Note: Need 35 digits to
                          // calculate integral part
                          // and the result becomes
                          // (31,0).
}

```

Figure 281. Runtime Exceptions Issued by the Compiler for `_DecimalT` Class Templates

Note:

1. For assignments to a target field too small to hold the `_DecimalT` class template object, there is a runtime exception issued for static, external, or automatic initialization.
2. For expressions requiring decimal-point alignment, namely addition, subtraction, or comparison, there is a runtime exception issued for static, external, and automatic initialization if, during alignment, the maximum number of allowed digits is exceeded.
3. For multiplication, if the evaluation of the expression results in a value larger than the maximum number of allowed digits (`DEC_DIG`):
 - A runtime exception is issued either for static or external initialization, or if the expression is within a function.
 - Truncation occurs on the fractional part, preserving as much of the integral part as possible.
4. For division, a runtime exception is generated when $((n[1] - p[1]) + p[2]) > 31$.

Defining a C++ `_DecimalT` Class Template Exception Handler

An object instantiated from an error class is thrown if an error occurred during runtime. You can catch the exception by defining your own exception handler.

The following figure illustrates how to use try-catch-throw to handle a `_DecimalT` class template exception.

```
#include <bcd.h>
#include <iostream.h>

void main() {
    try {
        _DecimalT<10,2> = __D("AAA");
    }
    catch (_DecErrInvalidConst Err) {
        cout << "Invalid Decimal constant!" << endl;
    }
}
```

Figure 282. Example of Using the C++ Try Catch Throw Feature to Handle a `_DecimalT` Class Template Exception

Using Debug Macros for `_DecimalT` Class Templates

C++ exception handling is used in the `_DecimalT` class template. If an error is detected during runtime, an error object is thrown.

Table 36 on page 377 defines the three macros that you can use to turn assertion-checking on and off.

Macro	Meaning
<code>_DEBUG</code>	Assertion checking is on.
<code>_DEBUG_DECIMAL</code>	Assertion checking is on for the <code>_DecimalT</code> class template only.
<code>_NODEBUG_DECIMAL</code>	<i>Default:</i> Assertion checking is off. This macro can override the <code>_DEBUG</code> and <code>_DEBUG_DECIMAL</code> macros.

Note:

1. When assertion checking is on, the `_DecimalT` template-class constant and parameters of the `_DecimalT` class template are validated.
2. Checking for divide-by-zero, overflow, and truncation in the `_DecimalT` template-class digit is hard-coded in the C++ runtime libraries and cannot be turned off by the `_NODEBUG_DECIMAL` macro.
3. You can use the `_DEBUG` macro to turn assertion checking on for the `_DecimalT` class template. If you are already using the `_DEBUG` macro in your source, you can use the `_DEBUG_DECIMAL` macro to turn on assertion checking, for the `_DecimalT` class template only.

Enabling and Disabling Error Checking for the `_DecimalT` Class Template

To enable error checking within the `_DecimalT` class template, you can turn on the debug macro by adding either the `DEFINE(_DEBUG_DECIMAL)` option or the `DEFINE(_DEBUG)` option during the invocation of the compiler, as shown in the following figure.

```
CRTCPPMOD DEFINE(_DEBUG)
CRTBND CPP DEFINE(_DEBUG)
CRTCPPMOD DEFINE(_DEBUG_DECIMAL)
CRTBND CPP DEFINE(_DEBUG_DECIMAL)
```

Figure 283. Commands to Enable Error Checking within the `_DecimalT` Class Template at Compile Time

Note:

The C++ program is:

```
// This program calls an ILE COBOL program
// and passes a bcd object.

#include <iostream.h>
#include <bcd.h>

extern "OS" void CBLPGM(_DecimalT<9,7>);

int main(void)
{
    _DecimalT<9,7> arg=_D("12.1234567");

    // Call an ILE COBOL program and pass a bcd object
    // to it.

    CBLPGM(arg);

    cout <<"The COBOL program was called and passed a bcd object"<<endl;
}
```

The ILE COBOL program is:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBLPGM.
*****
*   Packed decimals:  This is going to be called by a C++   *
*   program to pass packed decimal data.                   *
*                                                           *
*****
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.

SOURCE-COMPUTER. IBM-I.
OBJECT-COMPUTER. IBM-I.

INPUT-OUTPUT SECTION.
FILE-CONTROL.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.
77  PAC-DATA                                PIC X(30)
                                         VALUE "PROGRAM START".
77  PACK-IN-WS                             PIC 99.9999999.

LINKAGE SECTION.
01  PACK-DATA                             PIC 9(2)V9(7) PACKED-DECIMAL.

PROCEDURE DIVISION USING PACK-DATA.

MAIN-LINE SECTION.

MOVE PACK-DATA TO PACK-IN-WS.
DISPLAY "**** BCD OBJECT RECEIVED IS: " PACK-IN-WS.
GOBACK.
```

Figure 287. Example of Calling an ILE COBOL Program from an ILE C++ Program and Passing a `_DecimalT` Class Template

The output is:

```
**** BCD OBJECT RECEIVED IS: 12.1234567
The COBOL program was called and passed a bcd object
```

File I/O With `_DecimalT` Class Templates

`_DecimalT` class template objects can be written to a file and read from a file using either the C Runtime file I/O commands (fopen, fprintf, etc.) or the C++ ofstream object.

The following figure provides an example.

```
// This program shows how to write _DecimalT class template
// constants to a file
// and scan them back again. Shows how to pass a _DecimalT
// class template array to a function.

#include <bcd.h>
#include <iostream.h>
#include <stdlib.h>

#define N    3          // Array size for decimal declaration.

FILE *stream;          // File pointer declaration.
                        // Declare valid array.

_DecimalT<4,2> arr_1[] = {__D("12.35"), __D("25.00"),
                        __D("-19.58")};
_DecimalT<4,2> arr_2[N];

void write_num(_DecimalT<4,2> a[N]); //Declare function to
// write to a file.

void read_num(_DecimalT<4,2> b[N]); //Declare function to
//read from a file.

int main(void)
{
int reposition=0;
                        // Open the file. Must use fopen()
                        // to access a physical file.
if ((stream = fopen("*CURLIB/OUTFILE","w+")) == NULL)
{
    cout <<"Can not open file" << endl;
    exit(EXIT_FAILURE);
}
write_num(arr_1);      // Call function to write values of the
// array to outfile with fprintf().

reposition=fseek(stream, 0L, SEEK_SET);

if (reposition!=0)
{
    cout <<"FSEEK failed to position file pointer" <<endl;
    exit(EXIT_FAILURE);
}

read_num(arr_2);      // Call function to read values of the
// array from file using fscanf().

fclose(stream);      // Close the file.
}
// write_num is passed a the array. These values are written to a
// text file with fprintf(). If the function is successful a 0 is
// returned, otherwise a negative value is returned (indicating an
// error.

void write_num(_DecimalT<4,2> a[N])
{
    int i, j;

    for (i=0;i < N;i++)
    {
        j = fprintf(stream,"%D(4,2)\n",a[i]);
        if (j < 0)
            cout <<"Number not written to file" <<a[i] <<endl;
    }
}
// read_num is passed a the array. The values are
// read from a text file with fscanf().
// If the function is successful a 0 is returned, otherwise a
// negative value is returned (indicating an error).

void read_num(_DecimalT<4,2> b[N])
{
    int i, j;

    for (i=0;i < sizeof(b)/sizeof(b[0]);i++)
```

```

    {
        j = fscanf(stream,"%D(4,2)\n",&b[i]);
        if (j < 0)
            cout <<"Error when reading from file" <<endl;
    }
    cout <<"b[0]=" <<b[0] <<endl;
    cout <<"b[1]=" <<b[1] <<endl;
    cout <<"b[2]=" <<b[2] <<endl;
}

```

The output is:

```

b[0]=12.35
b[1]=25.00
b[2]=-19.58

```

You can rewrite this program to use the `ofstream` class, as shown in the following figure:

```

// This program shows how to write _DecimalT template
// constants to a file
// and scan them back again. Shows how to pass a _DecimalT
// class template array to a function.

#include <bcd.h>
#include <iostream.h>
#include <stdlib.h>
#include <fstream.h>

#define N      3                // Array size
                                // for decimal declaration.

                                // Declare valid
                                // array.

_DecimalT<4,2> arr_1[] = {__D("12.35"), __D("25.00"),
                          __D("-19.58")};
_DecimalT<4,2> arr_2[N];

void write_num(_DecimalT<4,2> a[N]); //Declare function to
// write to a file.

void read_num(_DecimalT<4,2> b[N]); //Declare function to
//read from a file.

int main ( void )
{
    write_num(arr_1);           // Call function to write
                                // values of the
                                // array to outf with fprintf
                                // library function.

    read_num(arr_2);           // Call function to read
                                // values of the
                                // array from file using
                                // fscanf() function.
}
// write_num is passed an array. These values are
// written to a text file with the fstream class.
// If the function is successful a 0 is returned, otherwise a
// negative value is returned (indicating an error).

void write_num(_DecimalT<4,2> a[N])
{
    int i;
    ofstream outf("data",ios::trunc || ios::out,
                  filebuf::openprot);
    if (!outf)
    {
        cerr << "Could not open file 'data' " <<endl;
        exit (EXIT_FAILURE);
    }
    for (i=0; i < N; i++) {
        outf << a[i] << endl;
    }
    outf.close()
}
// read_num is passed an array. The values are
// read from a text file with the fstream class.

```



```

// If the function is successful a 0 is returned, otherwise a
// negative value is returned (indicating an error).

void read_num(_DecimalT<4,2> b[N])
{
    int i;
    ifstream file("data");

    if (!file)
    {
        cerr << "Could not open file 'data' " <<endl;
        exit (EXIT_FAILURE);
    }
    for (i=0; i<N; i++)
    {
        file >> b[i];
        cout << "b["<< i <<"]=" <<b[i] <<endl;
    }
    if (file.eof())
    {
        cerr << "Unexpected EOF!" <<endl;
        exit (EXIT_FAILURE);
    }
    file.close();
}

```

The output is:

```

b[0]=12.35
b[1]=25.00
b[2]=-19.58

```

Using Templates in C++ Programs



Templates may be used in C++ to declare and define a related set of:

- Classes
- Functions
- Static data members of template classes

The C++ language describes the syntax and meaning of each kind of template. Each compiler determines the mechanism that controls when and how often a template is expanded.

See the *ILE C/C++ Language Reference* for more information. The "Templates" topic includes descriptions and examples of:

- Template parameters
- Template instantiations and specializations (both explicit and partial)
- Static data members and templates
- Class templates and function templates
- Overloading function templates
- Nested templates

This section includes:

- [“Managing Template Instantiations” on page 384](#)
- [“Template Instantiation Management Options” on page 384](#)
- [“How the ILE C++ Compiler Handles Template Instantiations” on page 385](#)
- [“Example of a Class Template Instantiation” on page 386](#)
- [“Using the Default Template Instantiation Management Option” on page 387](#)
- [“Using the ILE Template Registry Option” on page 388](#)
- [“Using the ILE TEMPINC Option” on page 389](#)

Managing Template Instantiations

You can have only one explicit specialization of any external linkage template instance.

ILE provides several methods for you to handle potential multiple instantiations of templates:

- Use the default instantiation option. See [“Using the Default Template Instantiation Management Option”](#) on page 387 and [“Manually Structuring Code for Single Instantiations”](#) on page 387.
- Manage a repository of all instantiations in a template registry. See [“Using the ILE Template Registry Option”](#) on page 388.
- Use the ILE automatic Template Include facility to ensure single instantiations of templates. See [“Using the ILE TEMPINC Option”](#) on page 389.

Template Instantiation Management Options

The best way to instantiate templates depends on the programming environment and the capabilities of the programmer. [Table 37 on page 384](#) lists the template instantiation options and describes advantages and disadvantages of each.

Note: Any attempt to use both the `TEMPLATE(*TEMPINC)` and `TMPREG` options of the Create C++ Module (`CRTCPMOD`) command at the same time halts compilation.

CRTCPMOD Option	Advantages	Disadvantages
ILE C++ Default: TEMPLATE(*NONE) TMPREG(*NONE)	<p>You do not need to restructure code or file structure</p> <p>In a network environment, you avoid potential file identification problems associated with the ILE C++ TEMPINC option.</p> <p>In a team environment, you avoid file sharing problems associated with the ILE C++ TEMPINC option.</p> <p>In a mixed IFS/QSYS.LIB environment, you avoid file system dependency problems associated with the ILE C++ TEMPINC option.</p>	<p>The time required to compile and link an entire application might be dramatically increased because:</p> <ul style="list-style-type: none"> • The compiler has to instantiate every instance it sees. • Multiple instantiations could be encountered across the application, which forces the linker to throw away all but one. <p>If you make any change to the implementation of a template, you must recompile every unit that uses the template.</p> <p>To avoid multiple template definitions, you must manually structure the code so that a single definition is generated for each template class function or static data member. To do this:</p> <ul style="list-style-type: none"> • You must understand how the ILE C++ compiler reacts to templates. • You must be aware of all the template instantiations that are required by the program. • You might need to reorganize source files and create new compilation units.

Table 37. Template Instantiation Management Options (continued)

CRTCPMOD Option	Advantages	Disadvantages
ILE C++ Template Registry: CRTCPMOD TEMPLATE(*NONE) TMPREG(*DFT) or CRTCPMOD TEMPLATE(*NONE) TMPREG('path-name')	One template instantiation per application is guaranteed. You do not need to structure programs for automatic instantiation. Minimal manual intervention is required.	You might need to manually recompile dependent files whenever a template instantiation is removed.
ILE C++ TEMPINC option: CRTCPMOD TEMPLATE(*TEMPINC) TMPREG(*NONE) or CRTCPMOD TEMPLATE('path-name') TMPREG(*NONE)	One template instantiation per application is guaranteed. You avoid the longer compilation times that result from using the default. You do not have to recompile all units that use a template whenever that template implementation is changed.	You need to split into separate files implementation of the following: <ul style="list-style-type: none"> • function templates • member functions • static data members of class templates This option might not be practical in a team programming environment because: <ul style="list-style-type: none"> • The compiler might update source files while they are being modified by somebody else. • Source file modifications might not be file-system-independent. For example, header files that are locally available might be included rather than header files that are available on a network.

How the ILE C++ Compiler Handles Template Instantiations

When you use templates in your program, the ILE C++ compiler automatically instantiates each C++ template that meets all the following conditions:

- Referenced in the source code
- the definition is visible at the point at which the reference occurs
- Not explicitly specialized by the programmer

Generation of Static Member Definitions

In compliance with the ISO standard, static members of a template class are weakly defined by default. This means that if a weak static member is defined more than once in a program, that static member is initialized only once.

Some programs require strong static data members when they are linked to other modules. To override the default at compilation time, add the WEAKTMPL(*NO) parameter to the CRTCPMOD command.

Note: For detailed information about the WEAKTMPL option, see the *ILE C/C++ Compiler Reference*.

Internal Linkage

If a program consists of several units that are compiled separately, a given template may be expanded in two or more of the compilation units. For templates that define classes, inline functions, or static non-

member functions, this is the desired behavior. These templates need to be defined in each compilation unit in which they are used.

External Linkage

For other functions and for static data members that have external linkage, defining them in more than one compilation unit would normally cause an error when the program is bound. ILE C++ avoids this problem by giving special treatment to template-generated versions of these objects. At bind time, ILE C++ gathers all template-generated functions and static-member definitions, plus any explicit specializations, and resolves all references to them:

- If an explicit specialization of the function or static member exists, it is used for all references. All template-generated definitions of that function or static member are discarded.
- If no explicit specialization exists, one of the template-generated definitions is used for all references. Any other template-generated definitions of that function or static member are discarded.

Note: Multiple template-generated definitions of functions or static members result in larger modules and longer compile times. The duplicated code for the templates is eliminated during binding, so that executable programs are not larger.

Example of a Class Template Instantiation

In the following example, the class template `Stack` implements a stack of items. The overloaded operators `<<` and `>>` are used to push items onto the stack and pop items from the stack.

```
template <class Item, int size> class Stack {
public:
    int operator << (Item item); // push operator
    int operator >> (Item& item); // pop operator
    Stack() { top = 0; } // constructor defined inline
private:
    Item stack[size]; // stack of items
    int top; // index to top of stack
};
```

Figure 288. Example of Class Template Instantiation

Declarations and Definitions

In [Figure 288 on page 386](#), the declaration of the `Stack` class template is contained in the file `stack.h` and the function definitions are contained in the file `stack.c`.

- Whenever you use either the default instantiation management option or the Template Registry option, with the `Stack` template, include both `stack.h` and `stack.c` in all compilation units that use an instance of the `Stack` class.
- Whenever you use the `TEMPINC` option, include only the `stack.h` file in all compilation units that use an instance of the `Stack` class. The `stack.c` file that contains the function template definitions is automatically included in the tempinc file for `stack.c`. All other compilation units that use an instance of the `Stack` class contain a reference to the definitions.
- You can structure your code to work either with or without the `TEMPINC` option by conditionally including the implementation file in the template header file using the macro defined by the compiler when the tempinc option is in effect, as shown in the following figure.

```
#ifndef __TEMPINC__
#include "stack.c"
#endif
```

Figure 289. Example of C++ Code that Works with or without the `TEMPINC` Option

Linkage

As shown in the following figure, the constructor function of the template is defined inline. Assume that the other functions are defined using separate function templates in the file `stack.c`:

```

template <class Item, int size>
int Stack<Item,size>::operator << (Item item) {
    if (top >= size) return 0;
    stack[top++] = item;
    return 1;
}
template <class Item, int size>
int Stack<Item,size>::operator >> (Item& item)
{
    if (top <= 0) return 0;
    item = stack[--top];
    return 1;
}

```

Figure 290. Example of a Constructor Function that Is Defined Inline

Note:

1. The constructor has internal linkage because it is defined inline in the class template declaration. This means that the compiler generates the constructor function body in each compilation unit that uses an instance of the Stack class. In other words, each unit has and uses its own copy of the constructor.
2. For the instance of the Stack class in the compilation unit, the compiler generates definitions for the following functions:

```

Stack<item,size>::operator<<(item)
Stack<item,size>::operator>>(item&)

```

because there is explicit specialization.

If the class template is instantiated in the source file `usr1stack.cpp`, that C++ program contains code similar to that shown in the following figure:

```

#include "stack.h"
#include "stack.c"
void Swap(int i&, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << i;
    i = j;
}

```

Note: The compiler generates the functions `Stack<int,20>::operator<<(int)` and `Stack<int,20>::operator>>(int&)` because both those functions are used in the program, their defining templates are visible, and no explicit specializations are seen.

Figure 291. Example of a Constructor Function that Is Defined Externally

Using the Default Template Instantiation Management Option

Whenever you use the default template instantiation management option, include both the template declaration header (.h file) and the out of line template definitions (.c files) in all compilation units that use an instance of the template class. The compiler generates definitions for each function template. For an example, see [“Example of a Class Template Instantiation”](#) on page 386.

Manually Structuring Code for Single Instantiations

If you do not want to use either of the ILE C++ automatic instantiation methods of generating template definitions, you can structure your program in such a way that you define templates directly in your compilation units. The advantage of this approach is that modules are smaller and compile times are shorter than they are when you include template definitions everywhere.

When you structure your code manually for template instantiation, you avoid the potential problems that are associated with automatic instantiation. See [Table 37](#) on page 384 for a list of potential problems.

Use either or both of the following methods to structure code for single instantiations:

- Use explicit instantiations to force the compiler to generate the necessary definitions for all template classes used in other compilation units. See [“Explicit Instantiations” on page 388](#).
- Use explicit specializations of non-member function templates to force the compiler to generate them. See the *ILE C/C++ Language Reference*. In the index, look up "templates", "explicit specializations".

Note: When you use these methods, you must specify the `TEMPLATE(*NONE)` compiler option to suppress automatic creation of IFS tempinc directories (or `TEMPINC` files). For more information about ILE C/C++ compiler options, see the *ILE C/C++ Compiler Reference*.

Explicit Instantiations

An explicit instantiation of a class forces the definition a class specialization without creating any object of the class. It implies the instantiation of all members of the class that have not already been explicitly specialized. If you do not require all class members in your program, you can explicitly instantiate only those individual members that the program requires.

For more information about explicit instantiations, see the *ILE C/C++ Language Reference*. In the index, look up "templates", and then the subheading "instantiations", "explicit".

In [Figure 292 on page 388](#):

- The header file `stack.h` is included in all compilation units that use an instance of the `Stack` class.
- The template implementation file `stack.c` is used in only one of the files.
- Only two instantiations of the class `Stack` are used in the program. These have the template arguments `< int, 20 >` and `< myClass, 100 >`.

Note: If you know all instances of the `Stack` class that are used in your program, you can define all of the instances in a single compilation unit.

```
#include "stack.h"
#include "stack.c"
#include "myclass.h" // Definition of "myClass" class
template class Stack<int,20>;
template class Stack<myClass,100>;
```

Figure 292. Example of All Instances of a Class Defined in a Single Compilation Unit

Using the ILE Template Registry Option

The ILE Template Registry is available in Version 5, Release 3, or later.

Note: The ILE compiler handles recompilation options differently than the AIX compiler. The ILE C/C++ compiler generates a warning when it is necessary to recompile.

Users must have read/write access to the directory where the template registry file is to be generated. Any attempt to access or create a template registry in a directory where the user does not have read/write access will result in a severe error and will halt compilation.

Note: If there are no templates in the source file, no error will be generated during compilation.

How the ILE Template Registry Option Works

The template registry:

- Maintains a repository of all template instantiations in a program or service program
- Tracks all references to template instantiations
- Ensures that only one definition is provided to the linker

The template registry designates a single module to contain the definition of a template instantiation.

- If a module that formerly contained a template instantiation is recompiled, the compiler will check to make sure that instantiation is still present.

- If there is a dependency on the template instantiation in another module and that definition has been removed, the compiler issues a warning that dependent modules will need to be recompiled to regenerate the missing instantiation. If the dependent modules are not recompiled, the link fails with errors saying that there are missing definitions.

During initial compilation, any template definitions or instantiations that are encountered are expanded and the template registry is updated with the location of the expansion in the module. If a template instantiation is required during subsequent compilations, the compiler checks the template registry to see if an instance already exists in another module. If so, it does not create a new instantiation, but does create a new reference. This ensures that only one definition is provided during the linking phase.

Specifying Values for the TEMPLREG Parameter

The value of the TEMPLREG parameter depends on whether the source file is a stream file or a data member file.

- If *DFT is specified, and:
 - If the source file is a stream file, a 'templaregistry' file is created in the source directory.
 - If the source file is a data member file, a file QTEMPLREG with the member QTEMPLREG is created in the library where the source resides.
- If a path name is specified, and:
 - If the source file is a stream file, that file is used as the template registry.

Note: If you want to explicitly use a file as your template registry, you have to use the "QSYS.LIB..." naming convention.
 - If the source file is a data member file, a file with the same name as the source file is created in the library where the source resides.

Using the ILE TEMPINC Option

Use the ILE TEMPINC option when the code has already been structured for it.

To use the ILE TEMPINC option:

1. List your templates in one or more header files but do not define any arguments for them.
2. Include only the header files in your source code.

Note: For each header file, you can use the #pragma implementation directive instead of a template implementation file to define templates.

3. When you bind your modules, use the same compiler options that you used to compile them. For example:

```
CRTPGM PGM (MYLIB/MYPROG) MODULE(MYLIB/MYFILE)
```

This is especially important for options that control libraries, linkage, and code compatibility.

How the ILE TEMPINC Option Works

For each header file with function templates that need to be defined, the compiler generates an IFS TEMPINC directory (or a tempinc file). The IFS TEMPINC directory (or a tempinc file) generates #include statements for any function templates, class template member functions, or class template static data members that need to be defined. These functions and members are found in:

- The header file with the template declaration
- The corresponding IFS TEMPINC directory (or a tempinc file)
- Any other header files that declare types used in template parameters

Before it invokes the binder, the compiler compiles the tempinc files and generates the necessary template definitions. Only one definition is generated for each template.

The TEMPLATE parameter defaults to *NONE. The other options are *SRCDIR or path-name, where path-name is an IFS directory or file. The compiler creates the specified IFS TEMPINC directory (or tempinc file) if it does not already exist.

Note: The applicable xlc qshell command option is `-qtempinc=dir`, where *dir* is a directory name. You can specify either a fully qualified path name or a path name relative to the current directory.

Structuring a Program for TEMPINC-Managed Instantiations

When you use the TEMPINC instantiation management option, follow these guidelines as you structure a program:

- If you have other declarations that are used inside templates but are not template parameters, you must place or `#include` them in either one of the included header files or in the tempinc file.

- Define any classes that are:

- Used in template arguments
- Required to generate the function template in the header file

Note: If definitions require other header files, include them with the `#include` directive. The definitions are then available when the unit is compiled.

- The function definitions in your template-implementation file can be explicit specializations, template definitions, or both. Any explicit specializations override the definitions generated by the template.
- If you specify a different directory for your tempinc files, ensure that you specify it consistently for all compilations of your program, including the bind step. For example:

```
CRTPGM PGM (MYLIB/MYPROG) MODULE(MYLIB/MYFILE)
```

- If you remove function instantiations or reorganize your program so that the tempinc files become obsolete, delete one or more of these files and recompile your program. If error messages are generated for a file in the IFS TEMPINC directory, you delete the file and recompile. To regenerate all of the tempinc files, delete the TEMPINC directory, the modules, and recompile your program.

Note:

1. After the compiler creates an IFS TEMPINC directory or file, it updates the file as each unit is compiled. The compiler never removes information from an individual file.
2. If you do not delete the modules, MAKEFILE rules prevent the modules from being recompiled, and the IFS TEMPINC files cannot be updated with all the lines needed for all the compilation units used in the program. The bind fails.

- Do *not* put the definitions of any classes used in template arguments in your source code.

Note: This example shows what *not* to do:


```

foo.h
template<class T> void foo(T*);
hoo.h
void hoo(A*);
foo.c
template<class T> void foo(T* t)
{t -> goo(); hoo(t);}
other.h
class A {public: void goo() {} };

main.cpp
#include "foo.h"
#include "other.h"
#include "hoo.h"
int main() { A a; foo(&a); }

```

This requires the expansion of the `foo(T*)` template with `class A` as the template type parameter. The compiler creates an IFS tempinc file `TEMPINC\foo.cpp`. The file contents (simplified below) are:

```

#include "foo.h"           //the template declaration header
#include "other.h"        //file defining template type parameter
#include "foo.c"          //corresponding template implementation

void foo(A*);            //triggers template instantiation

```

Note: This example cannot be properly compiled because the header file `hoo.h` did not satisfy the conditions for inclusion but the header file is required to compile the body of `foo(A*)`.

Figure 293. Example of Class Definitions Used in Template Arguments Also Contained in Source Code (Does Not Compile Properly)

One solution is to move the statement `#include "foo.h"` into the file `foo.c`.

The Template-Implementation File

In the Stack source, the file `stack.c` is a template-implementation file. To create a program using the Stack class template, both `stack.h` and `stack.c` must reside in the same directory. You include `stack.h` in any source files that use an instance of the class. The `stack.c` file does not need to be included in any source files. For example:

```

#include "stack.h"
void Swap(int i&, Stack<int,20>& s)
{
    int j;
    s >> j;
    s << i;
    i = j;
}

```

The compiler automatically generates the functions `Stack<int,20>::operator<<(int)` and `Stack<int,20>::operator>>(int&)`.

Figure 294. Example of Template-Implementation File

Renaming or Relocating the Template-Implementation file

Use the `#pragma implementation` directive to change the name of the template-implementation file or place it in a different directory.

The syntax is:

►► `#pragma — implementation — "path" —` ◄◄

Note:

1. The *path* is used to specify the path name for the template-implementation file. If it is only a partial path name, it must be relative to the directory containing the header file.
2. This path is a quoted string following the normal conventions for writing string literals. Backslashes must be doubled.

Changing the Template-Implementation file

To use the file `stack.def` as the template-implementation file instead of `stack.c`, add the line: `#pragma implementation("stack.def")` anywhere in the `stack.h` file, in the `Stack` class. The compiler then looks for the template-implementation file `stack.def` in the same directory as `stack.h`.

Tempinc Files

If you use the `TEMPINC` template instantiation management option, the `tempinc` file is generated by the compiler.

Note: Do not edit `tempinc` files.

Figure 295 on page 392 shows a typical `tempinc` file generated by the compiler.

```
1 /*0000000000*/ #pragma sourcedir("c:\swearsee\src")
2 /*0698421265*/ #include "\swearsee\src\list.h"
3 /*0000000000*/ #include "\swearsee\src\list.c"
4 /*0698414046*/ #include "\swearsee\src\mytype.h"
5 /*0698414046*/ #include "/QIBM/include/iostream.h"
6 template void List <MyType>::push(MyType);
7 template MyType List<MyType>::pop();
8 ostream& operator<<(ostream&, List<MyType>);
9 #pragma undeclared
10 int count(List<MyType>);
```

Figure 295. A Typical `tempinc` File

Note:

1. This `pragma` ensures that the compiler looks for nested include files in the directory containing the original source file, as required by the ILE C++ file inclusion rules.
2. The header file that corresponds to the `tempinc` file. The number in comments at the start of each `#include` line (for this line `/*0698421265*/`) is a time stamp for the included file. The compiler uses this number to determine if the `tempinc` file is current or should be recompiled. A time stamp containing only zeroes (0) as in line **3** means the compiler is to ignore the time stamp.
3. The template-implementation file that corresponds to the header file in line **2**.
4. Another header file that the compiler requires to compile the `tempinc` file. All other header files that the compiler needs to compile the `tempinc` file are inserted at this point.
5. Another header file required by the compiler. It is referenced in the function declaration in lines **6–7**.
6. The first statement of the explicit instantiation: In this case, the class `List<MyType>` is to be defined.
7. The second statement of the explicit instantiation: In this case, the class `List<MyType>` member functions are to be generated.
8. The `operator<<` function is a nonmember function that matched a template declaration in the `list.h` header file. The compiler inserts this declaration to force the generation of the function definition.
9. The `#pragma undeclared` directive is used only by the compiler and only in `tempinc` files. All function templates that are explicitly declared in at least one compilation unit appear before this line. All function templates that are called, but never declared, appear after this line. This division is necessary because the C++ rules for function overload resolution treat declared and undeclared function templates differently.
10. `count` is a function template that is called but not declared. The template declaration of the function is contained in `list.h`, but the instance `count(List<MyType>)` is never declared.

Using Teraspace in ILE C and C++ Programs

Teraspace is an extension of the IBM i storage model and runtime environment.

The teraspace storage model:

- Supports up to 2 GB, minus 244 bytes, of contiguous address ranges so that you can build high performance applications in a smaller runtime environment
- Assists porting of applications to the IBM i
- Supports the continuous evolution of existing IBM i applications

Note: *ILE Concepts* describes potential problems and usage tips for using teraspace in ILE programs.

Supported Teraspace Environments

ILE C/C++ provides 16-byte pointer libraries. ILE C++ also provides 8-byte pointer libraries (RTBND). These libraries are not binary-compatible. When using teraspace, ensure that bound modules are binary compatible. See [“Binary Compatibility Considerations When Porting Code in a Teraspace Environment” on page 397](#).

Note: For a detailed description of teraspace and other storage models, see "Teraspace and Single-Level Store" in *ILE Concepts*.

C/C++ Pointer Support

To enable programs for teraspace, C and C++ compilers provide the following pointer support:

- Syntax for explicitly declaring 8-byte or 16-byte pointers:
 - Declare a 8-byte pointer as `char * __ptr64`
 - Declare a 16-byte pointer as `char * __ptr128`
- Syntax for specifying and overriding the data model that is unique to the C/C++ programming environment. See [“The 8-Byte Runtime Binding \(RTBND\) Library Extensions” on page 394](#).

C/C++ Pointer Conversions

IBM C and C++ compilers convert pointers with attribute `__ptr128` to a pointer with attribute `__ptr64` (and vice versa) as needed, based on function and variable declarations.

Consider the following:

- A pointer with attribute `__ptr128` that points to single-level store (SLS) storage is arbitrarily converted to a pointer with attribute `__ptr64`.
- Code that is not teraspace-enabled cannot access teraspace.
- Interfaces with pointer-to-pointer parameters require special handling. See [“Maintaining Consistent Argument Declarations” on page 397](#).

The ILE compilers automatically insert pointer conversions to match pointer lengths. In the case of multiple level pointers, the conversion is performed only on the first level of the pointer. For example, conversions are inserted whenever:

- The pointer arguments to a function do not match the length of the pointer parameters in the prototype for the function
- Pointers of different lengths are compared

Note: You can specify explicit pointer conversions. See [“Casting Pointers” on page 280](#).

When casting pointers in a teraspace environment, consider the following:

- A conversion from a 16-byte pointer to an 8-byte pointer works only if the 16-byte pointer contains a teraspace address or a null pointer value. Otherwise, either a MCH0609 exception condition occurs or an arbitrary teraspace offset value is returned.
- 16-byte pointers cannot have types converted from one to another, but a 16-byte Open pointer can contain any pointer type. In contrast, no 8-byte Open pointer exists, but 8-byte pointers can be logically converted between a space pointer and a procedure pointer. Even so, an 8-byte pointer conversion is

just a view of the pointer type, so it does not allow a space pointer to actually be used as a procedure pointer unless the space pointer was set to point to a procedure.

Bindable APIs for Using Teraspace

IBM provides the following bindable APIs for allocating and discarding teraspace.

`_C_TS_malloc()`

Allocates storage within a teraspace.

`_C_TS_free()`

Frees one previous allocation of teraspace.

`_C_TS_realloc()`

Changes the size of a previous teraspace allocation.

`_C_TS_calloc()`

Allocates storage within a teraspace and sets it to 0.

Note: If a teraspace storage model is not specified, these bindable APIs allocate or deallocate single-level storage or teraspace storage according to the storage model of their calling program. For more information about Interprocess Communication APIs and the `shmget()` interface, see the UNIX-Type APIs topic in the *APIs* section in the *Programming* category at the IBM i Information Center web site: <http://www.ibm.com/systems/i/infocenter>.

The 16-Byte Runtime Binding Libraries

Unless you use the RTBND compiler option to specify the runtime binding directory, the C++ runtime environment uses the neutral storage model.

Because it uses 16-byte internal pointers, the neutral storage model:

- Can be used by programs built with either a single-level storage (SLS) model or a teraspace storage model
- Requires more pointer size conversions at runtime

Note: System pointers such as `_SYSPTR` and `_INVPTR` remain 16 bytes in size.

For service programs to use the 16-byte C++ runtime libraries, a default value for RTBND option must be in effect.

Note: For more detailed information about enabling programs for teraspace and rules for binding service programs, see *ILE Concepts*.

The 8-Byte Runtime Binding (RTBND) Library Extensions



To achieve optimal ISO compliance, use the 8-byte pointer Standard Library and object model when you:

- Want to build teraspace high-performance applications using 8-byte pointers
- Are not concerned with the binary incompatibilities with previously built service programs or runtime libraries

Note: The ILE C++ 8-byte runtime libraries are available with Version 5.3 or higher.

The 8-byte runtime library extensions include:

- Binding directory QYPPLR510T in library QSYS
- Service programs QYPPRT510T, QYPPSL510T, and QYPPWL530T (which replace service programs QYPPRT370, QYPPSL510, and QYPPWL530)
- Updated C++ standard library header files

When RTBND(*LLP64) is in effect:

- Binding directory QSYS/QYPPLR510T is used (instead of QSYS/QYPPWL510).

- The C++ Itanium ABI data layout is used. For documentation of the C++ Itanium ABI, see <http://www.codesourcery.com/cxx-abi/abi.html>.
- New POSIX-compliant C signals are available where:
 - The synchronous `signal()` function is mapped to the asynchronous `sigaction()` function
 - The synchronous `raise()` function is mapped to the asynchronous `kill()` function
- Two reserved macros `__LLP64_RTBND__` and `__ASYNC_SIG__` are defined to have the value 1.

Using RTBND to Optimize Performance of a C++ Program

C++ Typically, the size of internal runtime pointers is determined by either the DTAMDLL compiler option or the `#pragma datamodel` directive.

Requirements

To build a C++ program that uses the 8-byte runtime binding library extensions, use the following compiler options:

- Data Model option `DTAMDLL(*LLP64)`
- Storage Model option `STGMDLL(*TERASPACE)`
- `TERASPACE(*YES *TSIFC)`
- Runtime Binding option `RTBND(*LLP64)`

Note: For information specific to using these options, see the *ILE C/C++ Compiler Reference*.

Error Conditions

If incompatible options are used with the `RTBND(*LLP64)` option, diagnostic error CZS2121 is generated and the compilation is stopped.

If the `RTBND(*LLP64)` option is used with any release prior Version 5.3, diagnostic error CZS2120 is generated and the compilation is stopped.

Limitations

When the `RTBND(*LLP64)` option is in effect:

- An 8-byte pointer can point to teraspace only.
- An 8-byte procedure pointer always refers to an active procedure through teraspace.
- The only 8-byte pointer types are space pointers and procedure pointers.
- You *cannot* specify the `*LLP64` parameter of the `RTBND` option if the C++ program:
 - Uses demangling code (8-byte and 16-byte pointer names are mangled differently)
 - Calls objects created with previous versions of the compiler (they are not binary-compatible)

Characteristics of Each Teraspace Storage Model

The following table compares characteristics of the two data models.

<i>Table 38. Characteristics of the Default Teraspace Storage Model Versus the RTBND(LLP64) Teraspace Storage Model</i>	
When RTBND(*DEFAULT) is in effect:	When RTBND(*LLP64) is in effect:
Binding directory QSYS/QYPPLR510 is used.	Binding directory QSYS/QYPPLR510T is used.
Service programs QYPPRT370, QYPPSL510, and QYPPWL530 are used.	Service programs QYPPRT510T, QYPPSL510T, and QYPPWL530T are used.

Table 38. Characteristics of the Default Teraspace Storage Model Versus the RTBND(LLP64) Teraspace Storage Model (continued)

When RTBND(*DEFAULT) is in effect:	When RTBND(*LLP64) is in effect:
The size of the this pointer depends on the effective data model of the class declaration.	The size of this pointer is 8 bytes, regardless of the data model class declaration. All internal compiler-generated structures are assumed to be 8-byte pointers. That is the case even when the class declaration is surrounded by <code>#pragma datamodel (P128)</code> .
16-byte-compatible sections of C++ standard library header files are used.	8-byte-compatible sections of C++ standard library header files are used.
All internal structures (for example virtual function tables) use 16-byte pointers.	All internal structures (for example virtual function tables) use 8-byte pointers.
All C++ programs and C++ service programs can contain only modules that have been built with the RTBND(*DEFAULT) option.	All C++ programs and C++ service programs can contain only modules built that have been built with the RTBND(*LLP64) option.
Standard name mangling is in effect.	Demangling code written for 16-byte library does not work because a new ABI name mangling scheme is used instead. The ABI name mangling scheme prevents binding of modules that were not built with RTBND(*LLP64). Note: If names in one or more modules are explicitly crafted to defeat name mangling, incompatibilities can occur.
A derived class must be of the same data model as the base class.	
Declaration of <code>main()</code> must be consistent with the data model option used when the module is compiled. See “Maintaining Consistent Argument Declarations” on page 397.	
A function declared with a variable argument list is governed by the data model in effect when the argument list is declared. The argument list variables must always be consistent in terms of the size of pointer variables. ILE C/C++ compilers provide one level of pointer conversions of pointers in the variable list.	
A class or structure uses the data model in effect when that class or structure is fully declared. This data model may be different from the data model in effect when that same class or structure is forward-declared. See “Example: Effect of Forward Declarations on the Data Model” on page 398.	
An unprototyped function has its signature inferred by the data model in effect at the time of its first reference. C The C runtime functions might produce unpredictable results if the header files are not included.	
C++ Reference types are 8 bytes in length only when the data model is LLP64.	
C++ A template adopts the data model in effect when it is declared, and applies that data model to future instantiations of the template. All derived classes must be of the same data model as the base class. See “Example: How a Template Adopts a Data Model” on page 399.	
C++ A function signature is affected by the data model governing the class/function declaration. See “Examples: Overloading Functions” on page 400.	
The address of (&) operator returns an 8-byte result.	
Intermixing of pointer sizes in source code is allowed to permit use of the many system APIs that still use 16-byte pointers in structures and function prototypes.	

Table 38. Characteristics of the Default Teraspace Storage Model Versus the RTBND(LLP64) Teraspace Storage Model (continued)

When RTBND(*DEFAULT) is in effect:	When RTBND(*LLP64) is in effect:
Using an array in pointer context is the same as taking the address of the first element in the array. The size of the address is 8-byte if the storage model is teraspace.	
Note: Arrays are not pointers, so they are not affected by pointer modifiers.	

Binary Compatibility Considerations When Porting Code in a Teraspace Environment

Internal representations of classes in all bound modules must be binary-compatible but there is no explicit mechanism to prevent the binding of modules built with RTBND(*LLP64) and those without.

You might experience unexpected runtime behavior whenever:

- C++ programs built with RTBND(*LLP64) option are bound with C++ service programs built without this option
- C++ programs built without the RTBND(*LLP64) option are bound with service programs that have been built with this option.

Specifying the Teraspace Environment

The Storage Model (STGMDL) compiler option determines the teraspace environment for the entire program.

The Data Model (DTAMDLL) compiler option determines the pointer size declaration for the applicable module. To use the 8-byte runtime libraries, specify DTAMDLL(*LLP64).

To override the effect of the default Data Model (DTAMDLL) setting for a section of the source, you can use either DTAMDLL(*LLP64) or the `#pragma datamodel` directive. See [“Example: Effect of Forward Declarations on the Data Model”](#) on page 398 and [“Example: How a Template Adopts a Data Model”](#) on page 399.

Note: The `#pragma datamodel` directive overrides the DTAMDLL option setting.

To override the section setting for a specific variable, specify an attribute for the pointer. See [“Maintaining Consistent Argument Declarations”](#) on page 397 and [“Examples: Overloading Functions”](#) on page 400.

Determining the Size of a Specific Pointer

To determine the size of a specific pointer:

1. Check the pointer attribute.
2. If no pointer attribute exists, check the `#pragma datamodel` setting.
3. If a `#pragma datamodel` directive has not been included in the source section, check the Data Model (DTAMDLL) compiler option specified When the C or C++ module is created or bound.

Maintaining Consistent Argument Declarations

The validity of an argument list declared in `main()` depends on the data model that is in effect when the module is created or bound.

Table 39. Maintaining Consistent Argument Declarations	
Declaration	Valid for Data Models
<code>main(int argc, char * * argv)</code>	P128, LLP64
<code>main(int argc, char * argv[])</code>	P128, LLP64

Declaration	Valid for Data Models
main(int argc, char *__ptr128 *__ptr128 argv)	P128, LLP64
main(int argc, char *__ptr128 argv[])	P128
main(int argc, char *__ptr128 * argv)	P128
main(int argc, char *__ptr64 *__ptr64 argv)	P128, LLP64
main(int argc, char *__ptr64 argv[])	LLP64
main(int argc, char *__ptr64 * argv)	LLP64
main(int argc, char *__ptr64 *__ptr128 argv)	(Invalid for any data model)
main(int argc, char *__ptr128 *__ptr64 argv)	(Invalid for any data model)

Note: The same rule also applies to the third parameter, *envp*, in `main()`. Moreover, the resulting pointer sizes of *argv* and *envp* must also be the same.

Source Code Samples

The following source samples illustrate how to code programs when using *teraspace*.

Example: Effect of Forward Declarations on the Data Model

In the following figure, `struct Foo` does not use the P128 data model in effect at the time of its forward-declaration. Instead, `struct Foo` uses the LLP64 data model in effect at the time `struct Foo` is fully declared.

```
#pragma datamodel(P128)
struct Foo; //forward declaration
#pragma datamodel(LLP64)
struct Foo {
    char* string; // this pointer is 8-byte because LLP64 datamodel
                // was in effect for the struct definition.
};
#pragma datamodel(pop)
#pragma datamodel(pop)
```

Figure 296. Example of a Forward Declaration Based on the LLP64 Data Model

Example: Redefining the new or delete Operator

C++ To create modules that allocate storage dynamically from *teraspace*, specify `TERASPACE(*YES *TSIFC)` on the `CRTCPPMOD` and `CRTBNDCPP` commands. This remaps C runtime functions (such as `malloc`, `calloc`, and `free`) to new *teraspace* equivalents. This also affects storage dynamically allocated using the `new` and `delete` operators. If you want the `new` and `delete` operators to allocate storage in other way, you can override these operators.

The following code redefines the global `new` and `delete` operators to call functions (such as `malloc` or `free`).

Note: To find the declaration of `_set_mt_new_handler`, the following source must be compiled with `DEFINE('__MULTI__')`.

```
#include <stdlib.h>
#include <new>

using namespace std;
void *operator new(size_t size) throw (bad_alloc)
{
    // if size == 0, we must return a valid object! (ARM 5.3.3)
    if (size <= 0)
        size = 1;

    void *ret = (malloc)(size);

    while (ret == NULL)
    {
        // The malloc failed. Call the new_handler to try to free more memory
        void (*temp_new_handler)();

        // First check to see if a thread local new handler is defined.
        temp_new_handler = _set_mt_new_handler(0);
        _set_mt_new_handler(temp_new_handler);

        // If there is no thread local handler try the global handler.
        if (temp_new_handler == NULL)
        {
            // Note that the following code is not thread safe
            // If the application is threaded and a new handler is set then
            // all calls to set_new_handler() in the application must be
            // blocked. Otherwise for the sake of speed, do not use a locking
            // mechanism.

            temp_new_handler = set_new_handler(0);
            set_new_handler(temp_new_handler);
        }

        if (temp_new_handler != NULL)
        {
            temp_new_handler();

            // Try one more time
            ret = (malloc)(size);
        }
        else
            throw bad_alloc();
    }

    // just return the result to the user
    return ret;
}

void operator delete(void *ptr)
{
    // delete of NULL is okay
    if (ptr == NULL)
        return;

    (free)(ptr);
}
```

Figure 297. Example of Source Code that Redefines the Global `new` and `delete` Operators

Example: How a Template Adopts a Data Model

In the following example:

- Any instantiation of `Foot` uses the P128 data model
- Any instantiation of `FootZ` uses the LLP64 data model

```

#pragma datamodel(LLP64)
template <class T>
class FooTZ {
public:
    T bar(const char * a, T x) { return x; }
};
#pragma datamodel(pop)

#pragma datamodel(P128)
template <class T>
class FooT {
public:
    T bar(const char * a, T x) {return x; }
};
#pragma datamodel (pop)

```

Figure 298. Example of a Template that Adopts the Data Model in Effect When the Template Is Declared

Examples: Overloading Functions

C++ A function signature is affected by the data model governing the class/function declaration.

For example, `int Bar::foo(const char *)` is mangled to:

- `foo__3BarFPc` when the data model is P128
- `foo__3BarZFPCc` when the data model is LLP64
- `_ZN3Bar3fooEPKc` when the runtime binding is LLP64

It is possible to create overloaded methods which are identical in every way except the size of a pointer argument.

Example:

```

class Bar {
int foo(const char* __ptr128);
int foo(const char* __ptr64);
};

```

Casting with RunTime Type Information

C++

RunTime Type Information (RTTI) is a major extension to the C++ language made by the ISO standard committee.

This topic describes:

- [The RTTI language extension](#)
- [Using C++ language-defined RTTI](#)
- [Using RTTI in constructors and destructors](#)
- [ILE C++ extensions to RTTI](#)

The RTTI Language Extension

The C++ language supports both the re-use of code and the building of programs from parts. There might be incompatibilities between RTTI mechanisms used internally by various C++ libraries. The RTTI language extension resolves those incompatibilities.

C++ language support for RTTI include:

dynamic_cast operator

This operator combines type-checking and casting in one operation. It checks whether the requested cast is valid, and performs the cast only if it is valid.

typeid operator

This operator returns the runtime type of an object. If the operand provided to the `typeid` operator is the name of a type, the operator returns a `type_info` object that identifies it. If the operand provided is an expression, `typeid` returns the type of the object that the expression denotes.

type_info class

This class describes the RTTI available, and is used to define the type returned by the `typeid` operator. This class provides to users the possibility of shaping and extending RTTI to suit their own needs. This ability is of most interest to implementers of object I/O systems such as debuggers or database systems.

Using C++ Language-Defined RTTI

To use RTTI you need to be familiar with the *dynamic cast* and *typeid* operators.

The dynamic_cast Operator

C++ A *dynamic cast* expression is used to cast a base class pointer to a derived class pointer. This is referred to as *downcasting*.

The `dynamic_cast` operator:

- Makes downcasting much safer than conventional static casting
- Obtains a pointer to an object of a derived class that is given a pointer to a base class of that object
- Returns the pointer only if the specific derived class actually exists

Note: If the specified derived class does not exist zero is returned.

Dynamic casts have the form:

►► `dynamic_cast` — < — *type_name* — > — (— *expression* —) ►◄

The operator converts the expression to the desired type `type_name`. The `type_name` can be a pointer or a reference to a class type. If the cast to `type_name` fails, the value of the expression is zero.

Dynamic Casts with Pointers

C++ A dynamic cast using pointers is used to get a pointer to a derived class in order to use a detail of the derived class that is not otherwise available. For an example, see [Figure 299 on page 401](#):

```
class employee {
public:
    virtual int salary();
};

class manager : public employee {
public:
    int salary();
    virtual int bonus();
};
```

Figure 299. ILE Source to Cast a Pointer to a Derived Class to Use a Detail that Is Otherwise Unavailable

With the class hierarchy used in [Figure 299 on page 401](#), dynamic casts can be used to include the `manager::bonus()` function in the manager's salary calculation but not in the calculation for a regular employee. The `dynamic_cast` operator uses a pointer to the base class `employee`, and gets a pointer to the derived class `manager` in order to use the `bonus()` member function.

In [Figure 300 on page 402](#), dynamic casts are needed only if the base class `employee` and its derived classes are not available to users (as in part of a library where it is undesirable to modify the source code). Otherwise, adding new virtual functions and providing derived classes with specialized definitions for those functions is a better way to solve this problem.

```

void payroll::calc (employee *pe) {
    // employee salary calculation
    if (manager *pm = dynamic_cast<manager*>(pe)) {
        // use manager::bonus()
    }
    else {
        // use employee's member functions
    }
}
}

```

Figure 300. ILE Source to Get a Pointer to a Derived Class to Use a Member Function in Specified Calculations Only

In Figure 300 on page 402:

- If `pe` actually points to a `manager` object at runtime, the dynamic cast is successful, `pm` is initialized to a pointer to a `manager`, and the `bonus()` function is used.
- If `pe` does not point to a `manager` object at runtime, `pm` is initialized to zero and only the functions in the employee base class are used.

Dynamic Casts with References

C++ The `dynamic_cast` operator can be used to cast to reference types. C++ reference casts are similar to pointer casts: they can be used to cast *from* references to base class objects *to* references to derived class objects.

In dynamic casts to reference types, `type_name` represents a type and `expression` represents a reference. The operator converts the expression to the desired type `type_name&`.

You cannot verify the success of a dynamic cast using reference types by comparing the result (the reference that results from the dynamic cast) with zero because there is no such thing as a zero reference. A failing dynamic cast to a reference type throws a `bad_cast` exception.

A dynamic cast with a reference is a good way to test for a coding assumption. In Figure 301 on page 402, the example used in Figure 299 on page 401 is modified to use reference casts.

Note: Figure 301 on page 402 is intended only to show the `dynamic_cast` operator used as a test. *This example does not demonstrate good programming style because it uses exceptions to control execution flow.* Using `dynamic_cast` with pointers, as shown in Figure 300 on page 402, is a better way.

```

void payroll::calc (employee &re) {
    // employee salary calculation
    try {
        manager &rm = dynamic_cast<manager&>(re);
        // use manager::bonus()
    }
    catch (bad_cast) {
        // use employee's member functions
    }
}
}

```

Figure 301. ILE Source to Get a Pointer to a Derived Class Using Reference Casts

The typeid Operator

The `typeid` operator identifies the exact type of an object that is given a pointer to a base class. It is typically used to gain access to information needed to perform some operation where no common interface can be assumed for every object manipulated by the system. Object I/O and database systems often need to perform services on objects where no virtual function is available to do so. The `typeid` operator enables this.

A `typeid` operation has this form:

►► `typeid` — (— `type_name` — `expression` —) ►►

Results of typeid Operations

The result of a typeid operation has type `const type_info&`.

Table 40 on page 403 summarizes the results of various typeid operations.

Operand	typeid Returns
<code>type_name</code>	A reference to a <code>type_info</code> object that represents it.
<code>expression</code>	A reference to a <code>type_info</code> that represents the type of the <code>expression</code> .
Reference to a polymorphic type	The <code>type_info</code> for the complete object referred to.
Pointer to a polymorphic type	The dynamic type of the complete object pointed to. Note: If the pointer is zero, the <code>typeid</code> expression throws <code>bad_typeid</code> exception.
Nonpolymorphic type	The <code>type_info</code> that represents the static type of the <code>expression</code> . Note: The <code>expression</code> is not evaluated.

Using the typeid Operator in Expressions

The examples in Figure 302 on page 403 use the typeid operator in expressions that compare the runtime type of objects in the employee class hierarchy:

```
// ...
employee *pe = new manager;
employee& re = *pe;
// ...
typeid(pe) == typeid(employee*) // 1.True - not a polymorphic type1
typeid(&re) == typeid(employee*) // 2.True - not a polymorphic type2
typeid(*pe) == typeid(manager) // 3.True - *pe represents a polymorphic type3
typeid(re) == typeid(manager) // 4.True - re represents the object manager3

typeid(pe) == typeid(manager*) // 5.False - static type of pe returned4
typeid(pe) == typeid(employee) // 6.False - static type of pe returned4
typeid(pe) == typeid(manager) // 7.False - static type of pe returned4

typeid(*pe) == typeid(employee) // 8.False - dynamic type of expression is manager
typeid(re) == typeid(employee) // 9.False - re actually represents manager
typeid(&re) == typeid(manager*) //10.False - manager* not the static type of re
// ...
```

Figure 302. Examples of typeid operator in Expressions

Note:

1. In the first comparison, `pe` is a pointer (that is, not a polymorphic type); therefore, the expression `typeid(pe)` returns the static type of `pe`, which is equivalent to `employee*`.
2. In the second expression, `re` represents the address of the object referred to (that is, not a polymorphic type); therefore, the expression `typeid(re)` returns the static type of `re`, which is a pointer to `employee`.
3. In the third and fourth comparisons, the type returned by `typeid` represents the dynamic type of the expression only when the expression represents a polymorphic type.
4. Comparisons 5, 6, and 7 are false because it is the type of the *expression* (`pe`) that is examined, not the type of the *object* pointed to by `pe`.

These examples do not directly manipulate `type_info` objects. Using the `typeid` operator with built-in types requires interaction with `type_info` objects:

```

int i;
// ...
typeid(i) == typeid(int)    // True
typeid(8) == typeid(int)   // True
// ...

```

Figure 303. Examples of typeid operators

The type_info Class

To use the typeid operator in RunTime Type Identification (RTTI) you must include the C++ standard header `<typeinfo.h>`. This header defines these classes:

type_info

Describes the RTTI available to the implementation. It is a polymorphic type that supplies comparison and collation operators, and provides a member function that returns the name of the type represented.

The copy constructor and the assignment operator for the class `type_info` are private; objects of this type cannot be copied.

bad_cast

Defines the type of objects thrown as exceptions to report dynamic cast expressions that have failed.

bad_typeid

Defines the type of objects thrown as exceptions to report a null pointer in a typeid expression.

Using RTTI in Constructors and Destructors

The typeid and dynamic_cast operators can be used in constructors or destructors, or in functions called from a constructor or a destructor.

If the operand of dynamic_cast used refers to an object under construction or destruction, typeid returns the type_info representing the class of the constructor or destructor.

If the operand of dynamic_cast refers to an object under construction or destruction, the object is considered to be a complete object that has the type of the constructor's or destructor's class.

The result of the typeid and dynamic_cast operations is undefined if the operand refers to an object under construction or destruction, and if the static type of the operand is not an object of the constructor's or destructor's class or one of its bases.

ILE C++ Extensions to RTTI

The ILE C++ extended_type_info class was designed to provide support for implementing a persistent object store. The basic operations that must be supported are:

- Allocation of memory of an object
- Allocation of memory for an array of objects
- Initial construction of an object
- Initial construction of an array of objects
- Copy construction of an object
- Copy construction of an array of objects

Additional operations for destroy and deallocation of memory are also provided to detect an exception that occurs during construction. These operations are:

- Destruction of an object
- Destruction of an array of objects
- Destruction of memory for an object
- Deallocation of memory for an array of objects

The extended_type_info Classes

Figure 304 on page 406 is a code sample that uses extended_type_info classes.

The extended_type_info class definitions are:

size()

Size of the type represented by the extended_type_info object.

create(void*)

This function is called to create an object of the type represented by the extended_type_info object at the storage location pointed to by `at`.

create(void*, size_t)

This function is called to create an array of objects of the type represented by the extended_type_info object at the storage location pointed to by `at`. If any exceptions are thrown during construction, `create()` destroys the array elements that were already constructed before rethrowing the exception.

copy(void* to, const void* from)

This function is called to copy an object of the type represented by the extended_type_info object into the storage location pointed to by `to`, using the value of the object referred to by `from`.

copy(void* to, const void* from, size_t)

This function is called to copy an array of objects of the type represented by the extended_type_info object into the storage location pointed to by `to`, using the value of the object referred to by `from`. If any exceptions are thrown during construction, `copy()` destroys the array elements that were already constructed before rethrowing the exception.

destroy(void*)

This function is called to destroy an object of the type represented by the extended_type_info object at the storage location pointed to by `at`.

destroy(void*, size_t)

This function is called to destroy an array of objects of the type represented by the extended_type_info object at the storage location pointed to by `at`. If any exceptions are thrown during destruction, `destroy()` destroys the remaining array elements that were already constructed before rethrowing the exception. If another exception is encountered during the destruction of the remaining elements, `destroy()` calls `terminate()`.

allocObject()

This function is called to allocate memory for an object of the type represented by the extended_type_info object. No initialization is performed. The user is expected to use the `create(void*)` or `copy(void*, const void*)` function to initialize the new memory.

allocArray(size_t)

This function is called to allocate memory for an array of objects of the type represented by the extended_type_info object. No initialization is performed. The user is expected to use the `create(void*, size_t)` or `copy(void*, const void*, size_t)` function to initialize the new memory.

deallocObject(void*)

This function is called to deallocate an object of the type represented by the extended_type_info object. No destruction is performed beforehand. The user is expected to use the `destroy(void*)` to terminate the object before deallocating the memory.

deallocArray(void*, size_t)

This function is called to deallocate an array of objects of the type represented by the extended_type_info object. No destruction is performed beforehand. The user is expected to use the `destroy(void*, size_t)` to terminate an array before deallocating the memory.

linkageInfo()

This function returns the mangled name of the class type.

```

class extended_type_info : public type_info {
public:
~extended_type_info();

    virtual size_t size() const=0;

    virtual void* create(void* at) const=0; //object
    virtual void* create(void* at, size_t count) const=0; // array

    virtual void* copy (void* to, const void* from) const=0; //object
    virtual void* copy (void* to, const void* from, size_t count) const=0;
//array

    virtual void* destroy(void* at) const=0; //object
    virtual void* destroy(void* at, size_t count) const=0; //array

    virtual void* allocObject() const=0; //object
    virtual void* allocArray(size_t count) const=0; //array

    virtual void* deallocObject(void* at) const=0; //object
    virtual void* deallocArray(void* at, size_t count) const=0; //array
};

```

Figure 304. ILE Source Showing extended_type_info Class Types

Using International Locales and Coded Character Sets

This topic describes how to:

- Work with alternate coded character sets
- Use international locales

Internationalizing a Program

This topic describes how to:

- Create a source physical file with a specific Coded Character Set Identifier (CCSID)
- Change the CCSID of a member in a source physical file to the CCSID of another member in another source physical file
- Convert the CCSID for specific source statements in a member

The Integrated Language Environment C/C++ compiler recognizes source code that is written in most single-byte EBCDIC CCSID.

Source code written in CCSID 905 and 1026 may not be recognized because the " character varies on these CCSIDs.

The CRTCMOD/CRTCPPMOD and CRTBNDC/CRTBNDCPP commands do not support the SRCSTMF parameter in a mixed-byte environment.

Double-byte character set (DBCS) source code requires special programming considerations.

Note: You should tag the source physical file with a CCSID value number if the CCSID (determined by the primary language) is other than CCSID 037 (US English).

Coded Character Set Identifiers

A *coded character set identifier (CCSID)* comprises a specific set of an encoding scheme (EBCDIC, ASCII, or 8-bit ASCII), character set identifiers, code page identifiers, and additional coding-related information that uniquely identifies the coded graphic character representation used.

A *character set* is a collection of graphic characters.

Graphic characters are symbols, such as letters, numbers, and punctuation marks.

A *code page* is a set of binary identifiers for a group of graphic characters.

Code points are binary values that are assigned to each graphic character, to be used for entering, storing, changing, viewing, or printing information.

Character Data Representation Architecture (CDRA) defines the CCSID values to identify the code points used to represent characters, and to convert the character data as needed to preserve their meanings.

Source File Conversions to CCSID

Your Integrated Language Environment C/C++ source program can be made up of more than one source file. You can have a root source member and multiple secondary source files (such as include files and DDS files).

If any secondary source files are tagged with CCSIDs that are different from the root source member, their contents are automatically converted to the CCSID of the root source member as they are read by the Integrated Language Environment C/C++ compiler.

If the primary source physical file has CCSID 65535, the job CCSID is assumed for the source physical file. If the source physical file has CCSID 65535 and the job is CCSID 65535, and the system has non-65535, the system CCSID value is assumed for the source physical file. If the primary source physical file, job, and system have CCSID 65535, then CCSID 037 is assumed. If the secondary file, job, and system CCSID is 65535, then the CCSID of the primary source physical file is assumed, and no conversion takes place.

Creating a Source Physical File with a Coded Character Set Identifier

You specify the character set you want to use with the CCSID parameter when you create a source physical file. The default for the CCSID parameter is the CCSID of the job.

This figure shows you what happens when you create a program object that has a root source member with CCSID 273 and include files with different CCSIDs. The Integrated Language Environment compiler converts the include files to CCSID 273. The program object is created with the same CCSID as the root source member.

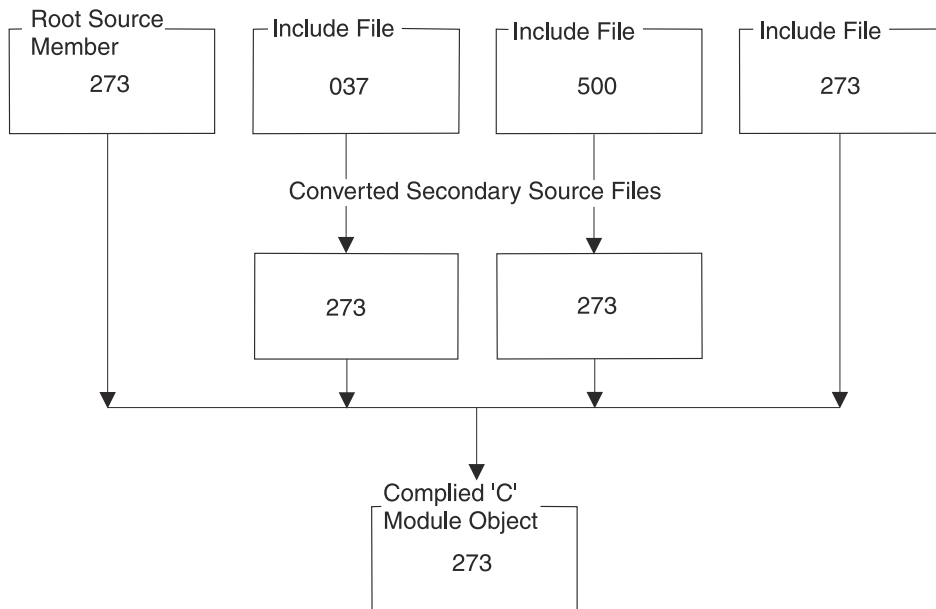


Figure 305. Source File CCSID Conversion

Note: Some combinations of the root source member CCSID and the include file CCSID are not supported.

Example:

The following example shows you how to specify CCSID 273 for the source physical file QCSRC in library MYLIB.

To create a source physical file with CCSID 273, type:

```
CRTSRCPF FILE(MYLIB/QCSRC) CCSID(273)
```

Changing the Coded Character Set Identifier (CCSID)

To change the CCSID of the source physical member from one CCSID to another, use the command CPYF with parameter FMTOPT(*MAP) to obtain the copy of the source physical member in another CCSID.

The following example shows you how to change a member in a source file with CCSID 037 to CCSID 273.

Example:

```
CRTSRCPF FILE(MYLIB/NEWCCSID) CCSID(273)
CPYF FROMFILE(MYLIB/QCPPSRC) TOFILE(MYLIB/NEWCCSID) FROMMBR(HELLO) TOMBR(HELLO)
MBROPT(*ADD) FMTOPT(*MAP)
```

Note:

1. The first command creates CCSID 273.
2. During the copy file operation, the character data in the from-member is converted between the from-file field CCSID and the to-file field CCSID *as long as a valid conversion is defined*.
3. The HELLO member in the file NEWCCSID is copied to QCSRC with CCSID 273. If CCSID 65535 or *HEX is used, it indicates that character data in the fields is treated as bit data and is not converted.

Converting String Literals in a Source File

You can convert the string literals in a source program from the point that the `#pragma convert` directive is specified to the end of the program. The `#pragma convert` directive specifies the CCSID to use for converting the string literals from that point onward in the program. The conversion continues until the end of the source or until another `#pragma convert` directive is specified.

If a CCSID with the value 65535 is specified, the CCSID of the root source member is assumed. If the source file CCSID value is 65535, CCSID 037 is assumed. The CCSID of the string literals before conversion is the same CCSID as the root source member. The CCSID can be either EBCDIC or ASCII.

Example:

The following example shows you how to convert the string literals in T1520CCS to ASCII CCSID 850 even though the CCSID of the source physical file is EBCDIC.

Note: In this example, the TGTCCSID parameter is defaulted to *SOURCE.

1. Type:

```
CRTBNDC PGM(MYLIB/T1520CCS) SRCFILE(QCPPLP/QACSRC)
```

To create the program T1520CCS using the following source:

```
/* This program uses the #pragma convert directive to convert      */
/* string literals.                                              */
#include <stdio.h>
char EBCDIC_str[20] = "Hello World";
#pragma convert(850) /* Use the #pragma convert                  */
/* directive to convert the                                     */
/* string to ASCII, CCSID 850.                                  */
char ASCII_str[20] = "Hello World";
#pragma convert(0) /* Stop string conversion.                  */
int main(void)
{
    int i;
    printf ("EBCDIC_str(hex) = "); /* Print hex value of EBCDIC */
    for (i = 0; i < 11; ++i) /* string. */
        printf ("%X ",EBCDIC_str[i]);
    printf ("\n\n");
    printf ("ASCII_str(hex) = "); /* Print hex value of ASCII */
    for (i = 0; i < 11; ++i) /* string. */
        printf ("%X ",ASCII_str[i]);
}
```

Figure 306. T1520CCS – ILE C Source to Convert Strings and Literals

The CRTBNDC command creates the program T1520CCS in library MYLIB. Program T1520CCS converts the EBCDIC string Hello World to ASCII CCSID 850.

2. To run the program T1520CCS, type:

```
CALL PGM(MYLIB/T1520CCS)
```

The output is as follows:

```
EBCDIC_str(hex) = C8 85 93 93 96 40 E6 96 99 93 84
ASCII_str(hex) = 48 65 6C 6C 6F 20 57 6F 72 6C 64
Press ENTER to end terminal session.
```

Using Unicode Support for Wide-Character Literals

Using Unicode character storage permits processing of characters in multiple character sets without loss of data integrity. Wide-character literals and strings can be stored as UCS-2 characters (Unicode CCSID 13488), which minimizes the need for code page conversions when developing applications for international use. The ILE C/C++ compiler supports Unicode character storage.

Representation of Wide-Character Literals

Wide-character literals can be represented in various ways. These representations and how they are handled are described below:

Trigraphs

Trigraph characters used as literals are converted to their corresponding Unicode characters. For example:

```
wchar_t *wcs = L" ??(";
```

is equivalent to:

```
wchar_t wcs[] = {0x0020, 0x005B, 0x0000};
```

For more information about trigraphs, see the *ILE C/C++ Language Reference*.

Character Escape Codes (\a, \b, \f, \n, \r, \t, \v, \', \", \?)

Character escape codes are converted to their corresponding Unicode escape codes. For example:

```
wchar_t *wcs = L" \t \n";
```

is equivalent to:

```
wchar_t wcs[] = {0x0020, 0x0009, 0x0020, 0x000A, 0x0000};
```

Numeric Escape Codes (\xnnnn, \ooo)

Numeric escape codes are not converted to Unicode. Instead, the numeric portion of the literal is preserved and assumed to be a Unicode character represented in hexadecimal or octal form. For example:

```
wchar_t *wcs = L" \x4145";
```

is equivalent to:

```
wchar_t wcs[] = {0x0020, 0x4145, 0x0000};
```

Specifying `\xnn` in a `wchar_t` string literal is equivalent to specifying `\x00nn`.

Hexadecimal constant values larger than `0xFF` are normally considered invalid. Setting the `*LOCALEUCS2` option changes this to allow 2-byte hexadecimal initialization of `wchar_t` types only. For example:

```
wchar_t wc = L'\x4145'; /* Valid only with *LOCALEUCS2 option, */
                        /* otherwise an out of bounds error */
                        /* will result. */
char      c = '\x4145'; /* Not valid due to size restriction. */
                        /* Error will result with or without */
                        /* specifying *LOCALEUCS2 option. */
```

Note: Numeric hexadecimal escape codes are not validated other than to ensure type size-limits are not exceeded.

DBCS Characters

DBCS characters entered as hexadecimal escape sequences are not converted to Unicode. They are stored as received.

Enabling Unicode Character Set Support

To enable Unicode character set support, specify `LOCALETYPE(*LOCALEUCS2)` on the `CRTCMOD/CRTCPMOD` or `CRTBND/CRTBNDCPP` command line. When this option is selected, the `__UCS2__` macro is defined. Wide-character literals are interpreted using the CCSID of the root source file, then translated to the Unicode CCSID (13488) when stored.

Effect of Unicode on `#pragma convert()` Operations

When `LOCALETYPE(*LOCALEUTF)` is specified, wide-character literals are always represented in UTF-32 format, regardless of the CCSID used by the root source file. In addition, `#pragma convert()` will have no effect on the wide-character literals.

When `LOCALETYPE(*LOCALEUCS2)` is specified, wide-character literals will always represent a UCS-2 character literal regardless of CCSID used by the root source file. In addition, `#pragma convert()` will ignore wide-character literals when converting characters from one codepage to another.

Example:

This example assumes a CCSID 37 source:

```
#include <stdio.h>
#include <wchar.h>
void main () {
#pragma convert (500)
    wchar_t wcs1[] = L"[]";
    char    str1[] = "[]";
#pragma convert (0)
    wchar_t wcs2[] = L"[]";
    char    str2[] = "[]";
    printf("str1 = %x %x\n", str1[0], str1[1]);
    printf("str2 = %x %x\n", str2[0], str2[1]);
    printf("wcs1 = %04x %04x\n", wcs1[0], wcs1[1]);
    printf("wcs2 = %04x %04x\n", wcs2[0], wcs2[1]);
}
```

Running the program would result in output similar to that shown below.

```
str1 = 4a 5a
str2 = ba bb
wcs1 = 005b 005d
wcs2 = 005b 005d
```

GB18030 Code Page Support

The Chinese government requires support of the GB18030 code page in all products sold in China. GB18030 is a Chinese standard that specifies a code page and a mapping table to Unicode.

The key points in this standard are:

- The code page uses multi-byte encoding.
- There is a direct mapping to Unicode (all the code points available in GB18030 are in Unicode).
- There are more than 64 K characters.

This means that GB18030 characters can be represented using Unicode, we use:

- UTF-8 to represent the narrow characters and string literals
- UTF-32 for the wide-characters and string literals

The basic character set that the compiler processes is UTF-8.

The user source does not have to be encoded in UTF-8 because the compiler converts it into UTF-8 for the internal processing.

Generating Wide Characters and String Literals in UTF-32

To generate wide characters and string literals in UTF-32, use the `LOCALETYPE(*LOCALEUTF)` option with:

- **C** Either the Create C Module (CRTCMOD) or the Create Bound C (CRTBNDC) command
- **C++** Either the Create C++ Module (CRTCPPMOD) or the Create Bound C++ (CRTBNDCPP) command

Considerations

When LOCALETYPE(*LOCALEUTF) is specified, wide-character literals are always represented in UTF-32 format regardless of the CCSID used by the root source file. In addition, `#pragma convert()` has no effect on the wide-character literals.

The LOCALETYPE(*LOCALEUTF) option requires that the target CCSID be 1208. When a default or specified target CCSID does not map to 1208:

- An override allows compilation to proceed.
- A diagnostic warning CZS2118 is issued.

A new-line character ('\n') is converted to the value 0x0a regardless of the SYSIFCOPT option (*IFS64IO || *IFSIO || *NOIFSIO).

Translation of narrow characters includes values above the basic character set. An example of such character is '-', which has the 2-byte value 0xC2AC in UTF-8.

C++ When the LOCALETYPE(*LOCALEUTF) option is specified, the compiler predefines 'wchar_t' as an unsigned integer with 4-byte size and alignment (otherwise 'wchar_t' remains an unsigned short integer with 2-byte size and alignment).

C When the LOCALETYPE(*LOCALEUTF) option is specified, the definition for an unsigned integer with 4-byte size and alignment is used. This definition is provided in `<stdlib.h>`.

Targeting a CCSID

The TGTCCSID parameter allows the compiler to:

- Process source files from a variety of CCSIDs or code pages (in the case of a source stream file)
- Target a module CCSID different from that of the root source file, as long as the translation between the source character set and the target module CCSID is installed into the operating system.

Target CCSID (TGTCCSID) is a parameter used with the following ILE C/C++ commands:

- Create C Module (CRTCMOD)
- Create C++ Module (CRTCPPMOD)
- Create Bound C (CRTBNDC)
- Create Bound C++ (CRTBNDCPP)

How the ILE C/C++ Compiler Converts a Source File to a Target CCSID

C When the TGTCCSID differs from the source file's CCSID, the ILE C compiler converts the source files to the TGTCCSID and processes files. This ensures that the target module and all its character data components (for example, listing, string pool) are in the desired TGTCCSID. You can then develop in one character set and target another. The argument defaults to the source file's character set so the default behavior is backward compatible (with the exception of 290, 930 and 5026).

Note: **C++** C++ converts only the string literals (not the source) to the TGTCCSID.

Providing support for more source character sets, increases the NLS usability of the compilers. CCSIDs 290, 930 and 5026 are now supported. The TGTCCSID parameter provides solutions to more complex NLS programming issues. For example, several modules with different module CCSIDs may be compiled from the same source by simply recompiling the source with different TGTCCSID values.

Literals, Comments, and Identifiers

The TGTCCSID parameter allows you to choose the CCSID of the resulting module. The module's CCSID identifies the coded character set identifier in which the module's character data is stored, including character data used to describe literals, comments and identifier names described by the source (with the exception of identifier names for CCSIDs 5026, 930 and 290).

For example, if the root source file has a CCSID of 500 and the compiler parameter TGTCCSID default value is not changed (that is, *SOURCE), the behavior is as before with the resulting module CCSID of 500. All string and character literals, both single and wide-character, are as described in the source file's CCSID. Translations may still occur as before for literals, comments and identifiers.

However, if the TGTCCSID parameter is set to 37 and the same source recompiled, the resulting module CCSID is 37; all literals, comments, and identifiers are translated to 37 where needed and stored as such in the module.

Regardless of what CCSID the root source and included headers are, the resulting module is defined by the TGTCCSID, and all of its literals, comments, and identifiers are stored in this CCSID.

Limitations

Debug Listing View

Introduction of the TGTCCSID parameter removes the limitation preventing the compilation of source with CCSIDs 5026, 930 or 290 without the loss of DBSC characters in literals and comments. However, a lesser limitation is introduced for these CCSIDs; when using listing view to debug a module compiled with TGTCCSID equal to CCSID 5026, 930, or 290, substitution characters appear for all characters not compatible with CCSID 37.

Format Strings

When coding format strings for C runtime I/O functions (for example, *printf("%d\n", 1234);*) the format string must be compatible with CCSID 037. When targeting CCSIDs 290, 930, 5026 which are not CCSID 037 compatible, a `#pragma convert(37)` is required around the format string literal to ensure that the runtime function processes the format string correctly.

Valid Target Encoding Schemes

TGTCCSID values are limited to CCSIDs with encoding schemes 1100 or 1301. An error message is issued by the command if any other value is entered.

1100 = EBCDIC, single-byte, No code extension is allowed, Number of States = 1.

1301 = EBCDIC, mixed single-byte and double-byte, using shift-in (SI) and shift-out (SO) code extension method, Number of States = 2.

International Locale Support

International locale support allows programs to change their behavior according to the user's language environment. This support has three key components:

- Programming tools that create language-specific data
- Programming interfaces (functions) that allow access to this data
- Methods of creating programs that are automatically sensitive to the language environment in which they run

Elements of a Language Environment

The typical elements of a language environment are as follows:

- Native language:
The natural language of the user.

- Character sets and coded character sets:

A coded character set is created by mapping the characters of a character set onto a set of code points (hexadecimal values). See [“Internationalizing a Program” on page 407](#) for more information about coded character sets and CCSIDs.

- Collating and ordering:

The relative order of characters that are used for sorting.

- Character classification:

The type of a character (for example, alphabetic, numeric) in a character set.

- Character case conversion:

The mapping between uppercase and lowercase characters in a character set.

- Date and time format:

The format of date and time data (for example, order of the months, names of the weekdays).

- Format of numbers and monetary quantities:

The format of numbers and monetary quantities. For example, numeric grouping, decimal-point character, and monetary symbols.

- Format of affirmative and negative system responses:

The format of affirmative and negative system responses.

Locales

A locale is a system object that specifies how language-specific data is processed, printed and displayed. A locale is made up of categories that describe the character set, collating sequence, date and time representation and monetary representation of the language environment in which it will be used. Using locales and locale-sensitive functions, applications can be created that are independent of language, cultural data, or character set, yet are sensitive to the language environment of the user.

ILE C/C++ Support for Locales

C The ILE C compiler and runtime environment support locales of the following types: *CLD, *LOCALE, *LOCALEUCS2, and *LOCALEUTF.

C++ The ILE C++ compiler and runtime environment support locales of the following types: *LOCALE, *LOCALEUCS2, and *LOCALEUTF.


Locales of type *LOCALE and the ILE C/C++ support for them is based on the IEEE POSIX P1003.2, ISO/IEC 9899:1990/Amendment 1:1994[E], and X/Open Portability Guide standards for global locales and coded character set conversion.

C The POSIX standard defines a much more comprehensive set of functions and locale data for application internationalization that is compared to that available for *CLD locales. By supporting the POSIX specification for locales in the ILE C/C++ runtime libraries, and introducing new functions which comply with the XPG4, POSIX and ISO/IEC standards, ILE C/C++ programs using locales of type *LOCALE become more portable to and from other operating systems.

Note: CLD is available for use by ILE C only.

ILE C/C++ Support for *CLD and *LOCALE Object Types

Programs that were compiled prior to V3R7 use the *CLD locale support. Programs that are compiled with the option LOCALETYPE(*CLD) on the CRTCMOD or CRTBNDC command uses the locale support that is provided by ILE C/C++ for *CLD objects. Programs compiled with the option LOCALETYPE(*LOCALE) on the CRTCMOD/CRTCPMOD or CRTBNDC/CRTBNDCPP command uses the locale support provided by ILE C/C++ for locales of type *LOCALE.

Note:  C++ does not support *CLD objects.

If you wish to convert your application from using locales of type *CLD to locales of type *LOCALE, the only changes required to your C source code are in calls to `setlocale()`. However, there are many differences between the locale definition source for *CLD and *LOCALE objects. The *LOCALE definition source members for many language environments are provided by the system in the optionally installable library QSYSLOCALE. You may also convert your existing *CLD locale source to the *LOCALE source definition. See [Table 41 on page 415](#) for a mapping of the commands in a source file for creating *CLD objects to the corresponding keywords in a source file for creating *LOCALE objects.

An application may use either locales of type *CLD or *LOCALE, but not both. If an ILE C program attempts to use both types of locales, the results are undefined. ILE C++ does not use *CLD. Also, some locale-sensitive functions are only supported when locales of type *LOCALE are used. See [Table 43 on page 421](#) for a list of locale-sensitive functions.

C Locale Migration Table

The ILE C runtime supports two implementations of the `setlocale()` function and the locale-sensitive functions. The original implementation uses locale objects of type *CLD, while the second implementation uses locale objects of type *LOCALE. The following table summarizes the differences in the locale source keywords between the locales of type *CLD and *LOCALE.

Category	*CLD Command	Description and Format	*LOCALE Keyword
LC_TIME	AM	String of characters used to represent the locale's equivalent of AM.	am_pm
LC_CTYPE	CHARTYP	Character set type.	Determined by CCSID of the locale
LC_COLLATE	CPYSYSCOL	System collating sequence table.	cpysyscol
LC_COLLATE	COLLSTR	String transformation table.	collating-element
LC_COLLATE	COLLTAB	Character weight reassignment for the <code>strcoll</code> function.	collating-element
LC_CTYPE	CTYPE	Set the attributes of a particular character.	upper lower alpha digit space cntrl punct graph print xdigit blank
LC_MONETARY	CURR	A string of characters that represent the currency symbol.	currency_symbol
LC_TIME	DATFMT	A string of characters used to specify the format of the date in this locale.	d_fmt
LC_TIME	DATTIM	A string of characters used to specify the format of the date and time in this locale.	d_t_fmt
LC_NUMERIC	DEC	Decimal point character for formatted non-monetary quantities.	decimal_point
LC_TOD	DSTEND	The instant when Daylight Savings Time ceases to be in effect. (day,time)	dstend

Table 41. C Locale Migration Table (continued)

Category	*CLD Command	Description and Format	*LOCALE Keyword
LC_TOD	DSTNAME	The name of the time zone when Daylight Savings Time is in effect.	dstname
LC_TOD	DSTSHIFT	The number of seconds that the locale's time is shifted when Daylight Savings Time takes effect.	dstshift
LC_TOD	DSTSTART	The instant when Daylight Savings Time comes into effect.	dststart
LC_NUMERIC	GROUP	Digit grouping from processing digits to the left of the decimal point from left to right for formatted non-monetary quantities.	grouping
LC_MONETARY	ICURR	String of characters used to represent the currency symbol in an internationally formatted monetary quantity.	int_curr_symbol
LC_TIME	LDAYS	The long form of each day of the week.	day
LC_TIME	LMONS	The long form of each month of the year.	mon
LC_CTYPE	LOWER	Set the lowercase character to be returned for a given character by the tolower library function.	tolower
LC_MONETARY	MDEC	String of characters used for the decimal point in a formatted monetary quantity.	mon_decimal_point
LC_MONETARY	MFDIGIT	Number of fractional digits to display in a formatted monetary quantity.	frac_digits
LC_MONETARY	MGROUP	Digit grouping from processing digits to the left of the decimal point from left to right for formatted monetary quantities.	mon_grouping
LC_MONETARY	MIFDIGIT	Number of fractional digits to display in an internationally formatted monetary quantity.	int_frac_digits
LC_MONETARY	MMINUS	String of characters used to represent a negative value in a formatted negative monetary quantity.	negative_sign
LC_MONETARY	MMINUSPOS	An encoded value used to represent the position of the negative symbol in a formatted negative monetary quantity.	n_sign_posn

Table 41. C Locale Migration Table (continued)

Category	*CLD Command	Description and Format	*LOCALE Keyword
LC_MONETARY	MNCSP	A true or false value used to determine if the currency symbol precedes the value in a formatted negative monetary quantity. If the value is false, then the symbol succeeds the value.	n_cs_precedes
LC_MONETARY	MNSBYS	A true or false value used to determine if the currency symbol is space-separated in a formatted negative monetary quantity.	n_sep_by_space
LC_MONETARY	MPCSP	A true or false value used to determine if the currency symbol precedes the value in a formatted positive monetary quantity. If the value is false, then the symbol succeeds the value.	p_cs_precedes
LC_MONETARY	MPLUS	String of characters used to represent a positive value in a formatted positive monetary quantity.	positive_sign
LC_MONETARY	MPLUSPOS	An encoded value used to represent the position of the positive symbol in a formatted positive monetary quantity.	p_sign_posn
LC_MONETARY	MPSBYS	A true or false value used to determine if the currency symbol is space-separated in a formatted positive monetary quantity.	p_sep_by_space
LC_MONETARY	MSEP	Character used to separate grouped digits in a formatted monetary quantity.	mon_thousands_sep
LC_TIME	PM	String of characters used to represent the locale's equivalent of PM.	am_pm
LC_TIME	SDAYS	The short form of each day of the week.	abday
LC_NUMERIC	SEP	Character used to separate grouped digits in a formatted non-monetary quantity.	decimal_point
LC_TIME	SMONS	The short form of each month of the year.	abmon
LC_TIME	TIMFMT	A string of characters used to specify the format of the time in this locale.	t_fmt
LC_TOD	TNAME	String of characters used to represent the locale's time zone name.	tname

Table 41. C Locale Migration Table (continued)

Category	*CLD Command	Description and Format	*LOCALE Keyword
LC_TOD	TZDIFF	The number of minutes that the locale's time zone is different from Greenwich Mean Time.	tzdiff
LC_CTYPE	UPPER	Set the uppercase character to be returned for a given character by the toupper library function.	toupper

POSIX Locale Definition and *LOCALE Support

Locale definition source files that conform to the IEEE POSIX P1003.2 standard will be shipped with the system in the optionally installable library QSYSLOCALE. One *LOCALE object, the C locale as defined by the POSIX standard, is provided with the system. Other locales of type *LOCALE can be created with the CRTLOCALE command from the locale source definition members in the QSYSLOCALE library.

LOCALETYPE Compiler Option

The LOCALETYPE option on the CRTCMOD/CRTCPPMOD or CRTBNDC/CRTBNDCPP command allows a program to specify the type of locale object to be used when it is being compiled. The keyword options for the LOCALETYPE option are *CLD, *LOCALE, *LOCALEUCS2, and *LOCALEUTF with the default being *LOCALE. The keyword *CLD enables the *CLD locale support, whereas the other keywords enable the support for locales of type *LOCALE.

The command format for enabling the runtime environment that supports locales of type *LOCALE is:

```
CRTCMOD MODULE(MYLIB/MYMOD) SRCFILE(MYLIB/QCSRC) LOCALETYPE(*LOCALE)
CRTBNDC PGM(MYLIB/MYPGM) SRCFILE(MYLIB/QCSRC) LOCALETYPE(*LOCALE)
```

When the *CLD keyword is not specified for the LOCALETYPE option, the ILE C/C++ compiler defines the macro `__POSIX_LOCALE__`. When `__POSIX_LOCALE__` is defined, the locale-sensitive C runtime functions are remapped to functions that are sensitive to locales that are defined in *LOCALE objects. In addition, certain ILE C/C++ runtime functions can only be used with locales of type *LOCALE and do not work with *CLD locales. These functions are available only in V3R7 and later releases of the ILE C/C++ runtime. The list of locale-sensitive functions in [Table 43 on page 421](#) indicates which functions are sensitive only to locales of type *LOCALE.

Note: The default has changed. Prior to V5R1, *CLD was the default value for ILE C. As of V5R1, the default has been changed to *LOCALE.

Creating Locales

On IBM i platforms, *LOCALE objects are created with the CRTLOCALE command, specifying the name of the file member containing the locale's definition source, and the CCSID to be used for mapping the characters of the locale's character set to their hexadecimal values.

A locale definition source member contains information about a language environment. This information is divided into a number of distinct categories which are described in the next section. One locale definition source member characterizes one language environment.

Characters are represented in a locale definition source member with their symbolic names. The mapping between the symbolic names, the characters they represent, and their associated hexadecimal values is based on the CCSID value that is specified on the CRTLOCALE command.

Here is a model of how a locale of type *LOCALE is created:

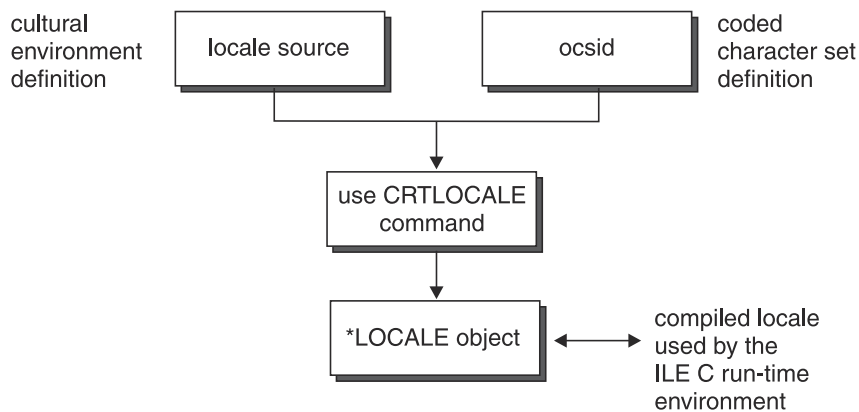


Figure 307. Creation of a locale of type *LOCALE

Note: There is no support for locales created with non-EBCDIC CCSIDs.

Creating Modules Using LOCALETYPE(*LOCALE)

When you create modules with the LOCALETYPE(*LOCALE) option, MB_CUR_MAX have the following values:

- 1 for locales built with a single byte CCSID, such as 00037
- 4 for locales built with a mixed byte CCSID such as 00939

MB_CUR_MAX is dependent on the LC_CTYPE category of the current locale.

Categories Used in a Locale

A locale and its definition source member contain the following categories.

Table 42. Categories Used in a Locale	
Category	Purpose
LC_COLLATE	Defines the collation relations among the characters. Affects the behavior of the collating functions <code>strcoll()</code> , <code>strxfrm()</code> , <code>wscoll()</code> and <code>wcsxfrm()</code> .
LC_CTYPE	Defines character types, such as upper-case, lower-case, space, digit, and punctuation. Affects the behavior of character handling functions.
LC_MESSAGES	Defines the format and values for responses from the application.
LC_MONETARY	Defines the monetary names, symbols, punctuation, and other details. Affects monetary information returned by <code>localeconv()</code> .
LC_NUMERIC	Defines the decimal-point (radix) character for the formatted input/output and string conversion functions, and the non-monetary formatting information returned by <code>localeconv()</code> .
LC_TIME	Defines the date and time conventions, such as calendar used, time zone, and days of the week. Affects the behavior of time display functions.
LC_TOD	Defines time zone difference, time zone name, and Daylight Savings Time start and end.

Setting an Active Locale for an Application

All C and C++ applications using locales of type `*LOCALE` have an active locale which is scoped to the activation group of the program. The active locale determines the behavior of the locale-sensitive functions in the C library. The active locale can be set explicitly with a call to `setlocale()`. See the *ILE C/C++ Runtime Library Functions* for more information on using `setlocale()`.

If the active locale is not set explicitly by a call to `setlocale()`, it is implicitly set by the C runtime environment at program activation time. Here is how the runtime environment sets the active locale when a program is activated:

- If the user profile has a value for the `LOCALE` parameter other than `*NONE` (the default) or `*SYSVAL`, that value is used for the application's active locale.
- If the value of the `LOCALE` parameter in the user profile is `*NONE`, the default "C" locale becomes the active locale.
- If the value of the `LOCALE` parameter in the user profile is `*SYSVAL`, the locale associated with the system value `QLOCALE` will be used for the program's active locale.
- If the value of `QLOCALE` is `*NONE`, the default "C" locale becomes the active locale.

Using Environment Variables to Set the Active Locale

A program's active locale is set either implicitly at program startup, as described above, or explicitly by a call to `setlocale()`. The `setlocale()` function takes two arguments: an integer representing the locale category whose values are needed for the active locale, and the name of the locale from which the values are to be taken. The name of the locale can be any of the following:

- C
- POSIX
- the fully-qualified path name of a locale object of type `*LOCALE`
- a null string("")

When the locale argument of `setlocale()` is specified as a null string(""), `setlocale()` sets the active locale according to the environment variables defined for the job in which the program is running. You can create environment variables that have the same names as the locale categories and specify the locale to be associated with each environment variable. The `LANG` environment variable is automatically created during job initiation when you specify a locale path name for the `LOCALE` parameter in your user profile or for the `QLOCALE` system value.

When a program calls `setlocale(category, "")`, the locale-related environment variables defined in the current job are checked to find the locale name or names to be used for the specified category. The locale name is chosen according to the first of the following conditions that applies:

1. If the environment variable `LC_ALL` is defined and is not null, the value of `LC_ALL` is used for the specified category. If the specified category is `LC_ALL`, that value is applied to all categories.
2. If the environment variable for the category is defined and is not null, then the value that is specified for the environment variable is used. For the `LC_ALL` category, if individual environment variables (for example, `LC_CTYPE`, `LC_MONETARY`, and so on) are defined and are not null, then their values are used for the categories that correspond to the environment variables. This could result in the locale information for each category that is retrieved from a different locale object.
3. If the environment variable `LANG` is defined and is not null, the value of the `LANG` environment variable is used.
4. If no non-null environment variable is present to supply a locale value, the default C locale is used.

If the locale specified for the environment variable is found to be invalid or non-existent, `setlocale()` returns `NULL` and the program's active locale remains unchanged.

For `setlocale(LC_ALL, "")`, if the locale names found identify valid locales on the system, `setlocale()` returns a string naming the locale associated with each locale category. Otherwise, `setlocale()` returns `NULL`, and the program's locale remains unchanged.

SAA and POSIX *Locale Definitions

If an ILE C program is compiled with LOCALETYPE(*LOCALE) and `setlocale()` is not called or if it is called with locale name C or POSIX, the default C environment used is that specified in the POSIX locale definition source in the QSYSLOCALE library. This locale definition is slightly different from the default C locale for type *CLD. Another locale definition source member that is called SAA is provided in the QSYSLOCALE library for compatibility with the default C locale of type *CLD.

If you wish to migrate your application from locales of type *CLD to locales of type *LOCALE, but you want to be compatible with the default C locale of type *CLD, use the SAA locale definition source member in the QSYSLOCALE library to create a locale with the CRTLOCALE command. Then use the name of this locale when you call `setlocale()` in your application.

The differences between the SAA and POSIX locale definitions are as follows:

- For the LC_CTYPE category, the SAA locale has all the EBCDIC control characters defined in the `cntrl` class, whereas the POSIX locale only includes the ASCII control characters. Also, SAA has the cent character and the broken vertical line as punct characters whereas POSIX does not include these two characters in its punct characters.
- For the LC_COLLATE category, the default collation sequence for SAA is the EBCDIC sequence whereas POSIX uses the ASCII sequence. This is independent of the CCSID mapping of the character set. For the POSIX locale, the first 128 ASCII characters are defined in the collation sequence, and the remaining EBCDIC characters are at the end of the collating sequence. Also, in the SAA locale definition, the lowercase letters collate before the uppercase letters, whereas in the POSIX locale definition, the lowercase letters collate after the uppercase letters.
- For the LC_TIME category, the SAA locale specifies the date and time format (`d_t_fmt`) as "%Y/%M/%D %X" whereas the POSIX locale uses "%a %b %d %H %M %S %Y".

Locale-Sensitive Runtime Functions

The following ILE C/C++ runtime functions are sensitive to locales:

<code>asctime()</code>	<code>asctime_r()</code>	<code>btowc()</code> ¹
<code>ctime()</code>	<code>ctime_r()</code>	<code>fprintf()</code>
<code>fgetwc()</code> ¹	<code>fgetws()</code> ¹	<code>fputwc()</code> ¹
<code>fputws()</code> ¹	<code>fwprintf()</code> ¹	<code>fwscanf()</code> ¹
<code>getwc()</code> ¹	<code>getwchar()</code> ¹	<code>gmtime()</code>
<code>gmtime_r()</code>	<code>isalnum()</code> to <code>isxdigit()</code>	<code>isascii()</code>
<code>iswalnum()</code> to <code>iswxdigit()</code> ¹	<code>localeconv()</code>	<code>localtime()</code>
<code>localtime_r()</code>	<code>mblen()</code>	<code>mbrlen()</code> ¹
<code>mbsinit()</code> ¹	<code>mbrtows()</code> ¹	<code>mbsrtowcs()</code> ¹
<code>mbstowcs()</code> ¹	<code>mbtowc()</code> ¹	<code>mktime()</code>
<code>nl_langinfo()</code> ¹	<code>printf()</code>	<code>putwc()</code> ¹
<code>putwchar()</code> ¹	<code>regcomp()</code>	<code>regerror()</code>
<code>regexec()</code>	<code>regfree()</code>	<code>setlocale()</code>
<code>sprintf()</code>	<code>strcoll()</code>	<code>strfmon()</code> ¹
<code>strftime()</code>	<code>strptime()</code> ¹	<code>strxfrm()</code>
<code>swprintf()</code> ¹	<code>swscanf()</code> ¹	<code>time()</code>

Table 43. Locale-Sensitive Runtime Functions (continued)

toascii()	tolower()	toupper()
tolower() ¹	towtrans() ¹	towupper() ¹
vfprintf()	vfwprint() ¹	vprintf()
vsprintf()	vswprintf() ¹	vwprintf() ¹
wcrtomb() ¹	wscoll() ¹	wcsftime()
wcstod()	wcstol()	wcstoul()
wcsrtombs() ¹	wcstombs()	wcswidth() ¹
wcsxfrm() ¹	wctob()	wctomb() ¹
wctrans() ¹	wctype() ¹	wcwidth() ¹
wscanf() ¹	wprintf()	
Note: ¹ sensitive to *LOCALE objects only.		

The GENCSRC Utility and the #pragma mapinc Directive

By using the Generate C/C++ Source (GENCSRC) utility to generate database header files, you can:

- Extract data description specification (DDS) information
- Create a header file with declarations for use in your programs

GENCSRC provides the means to retrieve externally described file information for use in a C/C++ program. The utility creates a C/C++ header file which contains the type definition structure for the include file.

The #pragma mapinc directive uses the GENCSRC command to provide the opportunity to convert DDS files to include files directly.

To compare the GENCSRC utility and the #pragma mapinc directive:

- GENCSRC can produce include files in IFS file systems or database file systems, while #pragma mapinc produces headers in database file systems only.
- GENCSRC include files can be placed permanently anywhere, while #pragma mapinc writes generated include files into the QTEMP library.

The following table shows the comparison of #pragma mapinc options and the keywords for GENCSRC. For more information on any particular option, refer to the description of #pragma mapinc in the *ILE C/C++ Compiler Reference*.

GENCSRC Keyword	#pragma mapinc option	Description
SRCFILE	member_name	The name of the file that you reference on the #include directive in the source program. The output file is generated in the file system.
SRCMBR	member_name	The name of member with the header information. It follows the IBM i naming conventions. The output file is generated in the file system.
OBJ	file_name	The path name of the object to map in QSYS file system.
SRCSTMF	member_name	The output file is generated in the IFS file system.
RCD_FMT	format_name	Indicates the DDS record format to be included in your program. The default is *ALL.
SLTFLD	options	Restricted to a combination of the following values: <ul style="list-style-type: none"> • *INPUT • *OUTPUT • *BOTH (default) • *KEY • *INDICATOR • *LVLCHK¹ • *NULLFLDS
PKDDECLD	d or p	*DECIMAL or *CHAR
STRUCTURE	_P	*PACKED or *NONPACKED

Table 44. Comparison of GENCSRC Keywords and #pragma mapinc Options (continued)

GENCSRC Keyword	#pragma mapinc option	Description
ONEBYTE	1BYTE_CHAR	*ARRAY or *CHAR
UNIONDFN	union_name	*OBJ , NONE , or union_name Note: *OBJ is the default and the default type definition union name is FILE_t and not LIBRARY_FILE_FMT_both_t as described in the #pragma mapinc directive
TYPEDEFPFX	prefix_name	*OBJ or *NONE ¹
Note: 1. For more information, see “Level Checking to Verify Descriptions” on page 184.		

Note: For information about DDS-to-C/C++ data type mapping, see the *ILE C/C++ Compiler Reference*.

Interlanguage Data-Type Compatibilities

Each high-level language (HLL) has different data types. When you want to pass data between programs written in different languages, you must be aware of these differences.

Some data types in the ILE C++ programming language have no direct equivalent in other languages. You can simulate data types in other languages using ILE C++ data types.

Note:

1. No data-type compatibility tables are shown for C. You can use the C++ tables because C and C++ data types are the same *except for the packed decimal data type*. For detailed information on using packed decimal data in ILE C/C++, see:
 - [“Using Packed Decimal Data in a C Program” on page 354](#)
 - [“Using Packed Decimal Data in a C++ Program” on page 365](#)
2. In C++ the packed decimal data type is implemented as the bcd class. The packed decimal data type in ILE C and the binary coded decimal class in C++ are binary-compatible.

The following tables are provided:

- [Table 45 on page 425](#) shows the ILE C++ data-type compatibility with ILE CL.
- [Table 46 on page 426](#) shows the ILE C++ data-type compatibility with CL.
- [Table 47 on page 427](#) shows the ILE C++ data type compatibility with ILE RPG.
- [Table 48 on page 430](#) shows the ILE C++ data-type compatibility with OPM RPG.
- [Table 49 on page 432](#) shows the ILE C++ data-type compatibility with ILE COBOL.
- [Table 50 on page 434](#) shows the ILE C++ data-type compatibility with OPM COBOL.

[Table 45 on page 425](#) shows the ILE C++ data-type compatibility with ILE CL.

ILE C++ declaration in prototype	CL	Length	Comments
char[n] char *	*CHAR LEN(&N)	n	An array of characters where n=1 to 32766. A null-terminated string. CHGVAR &V1 VALUE(&V *TCAT X'00') where &V1 is one byte bigger than &V
char	*LGL	1	Holds '1' or '0'
_Packed struct {short i; char[n]}	Not supported	n+2	A variable length field where i is the intended length and n is the maximum length.
integer types	*INT LEN(&N)	2, 4, 8	A 2-, 4-, or 8- byte signed integer. (CL does not support 1 byte integer type)
	*UINT LEN(&N)	2, 4, 8	A 2-, 4-, or 8- byte unsigned integer. (CL does not support 1 byte unsigned integer type)
float constants	CL constants only	4	A 4 or 8 byte floating point

Table 45. ILE C++ Data-Type Compatibility with ILE CL (continued)

ILE C++ declaration in prototype	CL	Length	Comments
<code>_DecimalT<n,p></code>	*DEC	n/2+1	A packed decimal. The limit of n is 15 and p is 9. In C++, this is a binary coded decimal class and not a data type.
<code>union.element</code>	Not supported	length of longest union member	An element of a union
<code>struct</code> or <code>class</code>	Not supported	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
<code>pointer to function</code>	Not supported	16	A 16-byte pointer

Table 46 on page 426 shows the ILE C++ data-type compatibility with CL.

Table 46. ILE C++ Data-Type Compatibility with CL

ILE C++ declaration in prototype	CL	Length	Comments
<code>char[n]</code> <code>char *</code>	*CHAR LEN(&N)	n	An array of characters where n=1 to 32766. A null-terminated string. CHGVAR &V1 VALUE(&V *TCAT X'00') where &V1 is one byte bigger than &V. The limit of n is 9999.
<code>char</code>	*LGL	1	Holds '1' or '0'
<code>_Packed struct {short i; char[n]}</code>	Not supported	n+2	A variable length field where i is the intended length and n is the maximum length.
integer types	*INT LEN(&N)	2, 4	A 2- or 4- byte signed integer. (CL does not support 1 byte integer type)
	*UINT LEN(&N)	2, 4	A 2- or 4- byte unsigned integer. (CL does not support 1 byte unsigned integer type)
<code>float constants</code>	CL constants only	4	A 4 or 8 byte floating point
<code>_DecimalT<n,p></code>	*DEC	n/2+1	A packed decimal. The limit of n is 15 and p is 9. In C++, this is a binary coded decimal class and not a data type.
<code>union.element</code>	Not supported	length of longest union member	An element of a union

Table 46. ILE C++ Data-Type Compatibility with CL (continued)

ILE C++ declaration in prototype	CL	Length	Comments
struct or class	Not supported	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer

Table 47 on page 427 shows the ILE C++ data type compatibility with ILE RPG.

Table 47. ILE C++ Data-Type Compatibility with ILE RPG

ILE C++ declaration in prototype	Free-form ILE RPG syntax	ILE RPG D spec, columns 33 to 39	Length	Comments
char[n] char *	CHAR(n)	nA	n	An array of characters where n=1 to 16773104. If it is a null-terminated string parameter, code the prototyped parameter as a pointer with the VALUE and OPTIONS(*STRING) keywords.
char	IND	1N	1	An indicator
char[n]	ZONED(n)	nS 0	n	A zoned decimal
_Packed struct {unsigned short i; char[n]}	VARCHAR(n)	nA VARYING	n+2	A variable length field where i is the intended length and n is the maximum length. Note: If n is greater than 65535, the first element of the struct is an unsigned int, and the length is n+4.
_Packed struct {unsigned int i; char[n]}	VARCHAR(n : 4)	nA VARYING(4)	n+4	A variable length field where i is the intended length and n is the maximum length.
wchar_t[n]	UCS2(n)	nC	2n	An array of UCS-2 characters. Note: The RPG UCS-2 type also supports UTF-16 data by specifying CCSID(1200).

Table 47. ILE C++ Data-Type Compatibility with ILE RPG (continued)

ILE C++ declaration in prototype	Free-form ILE RPG syntax	ILE RPG D spec, columns 33 to 39	Length	Comments
<code>_Packed struct {unsigned short i; wchar_t[n]}</code>	VARUCS2(n)	nC VARYING	2n+2	A variable length UCS-2 field where i is the intended length and n is the maximum length. Note: If n is greater than 65535, the first element of the struct is an unsigned int, and the length is 2n+4. Note: The RPG UCS-2 type also supports UTF-16 data by specifying CCSID(1200).
<code>_Packed struct {unsigned int i; wchar_t[n]}</code>	VARUCS2(n : 4)	nC VARYING(4)	2n+4	A variable length UCS-2 field where i is the intended length and n is the maximum length. Note: The RPG UCS-2 type also supports UTF-16 data by specifying CCSID(1200).
<code>wchar_t[n]</code>	GRAPH(n)	nG	2n	An array of graphic characters.
<code>_Packed struct {unsigned short i; wchar_t[n]}</code>	VARGRAPH(n)	nG VARYING	2n+2	A variable length graphic field where i is the intended length and n is the maximum length. Note: If n is greater than 65535, the first element of the struct is an unsigned int, and the length is 2n+4.
<code>_Packed struct {unsigned int i; wchar_t[n]}</code>	VARGRAPH(n : 4)	nG VARYING(4)	2n+4	A variable length graphic field where i is the intended length and n is the maximum length.
<code>char[n]</code>	DATE	D	8, 10	A date field
<code>char[n]</code>	TIME	T	8	A time field
<code>char[n]</code>	TIMESTAMP	Z	20-32	A time stamp field
<code>short int</code>	INT(5)	5I 0	2	An integer field
<code>short unsigned int</code>	UNS(5)	5U 0	2	An unsigned integer field
<code>int</code>	INT(10)	10I 0	4	An integer field
<code>unsigned int</code>	UNS(10)	10U 0	4	An unsigned integer field
<code>long int</code>	INT(10)	10I 0	4	An integer field

Table 47. ILE C++ Data-Type Compatibility with ILE RPG (continued)

ILE C++ declaration in prototype	Free-form ILE RPG syntax	ILE RPG D spec, columns 33 to 39	Length	Comments
long unsigned int	UNS(10)	10I 0	4	An unsigned integer field
long long int	INT(20)	20I 0	8	An 8-byte integer field.
long long unsigned int	UNS(20)	20U 0	8	An 8-byte unsigned integer field.
struct {unsigned int : n}x;	Not supported.	Not supported	4	A 4-byte unsigned integer, a bitfield
float	FLOAT(4)	4F	4	A 4-byte floating point
double	FLOAT(8)	8F	8	An 8-byte floating point
long double	FLOAT(8)	8F	8	An 8-byte floating point
_Decimal32	Not supported.	Not supported.	4	A 4-byte decimal floating point.
_Decimal64	Not supported.	Not supported.	8	An 8-byte decimal floating point.
_Decimal128	Not supported.	Not supported.	16	A 16-byte decimal floating point.
enum	INT(p)	pI	1, 2, 4	Enumeration. For n = 1,2,4: p = 3,5,10 Define ILE RPG named constants with the enum values.
*	POINTER	*	16	A pointer.
_DecimalT<n, p>	PACKED(n:p)	nP p	n/2+1	A packed decimal. n must be less than or equal to 30. In C++, this is a binary coded decimal class and not a data type.
union.element	Keyword POS(1)	<type> with OVERLAY(data structure name)	length of longest union member	An element of a union
data_type[n]	<type> with keyword DIM(n)	<type> with keyword DIM(n)	16	An array to which C++ passes a pointer

Table 47. ILE C++ Data-Type Compatibility with ILE RPG (continued)

ILE C++ declaration in prototype	Free-form ILE RPG syntax	ILE RPG D spec, columns 33 to 39	Length	Comments
struct or class	data structure	data structure	n	A structure. Use the <code>_Packed</code> qualifier on the struct if the RPG data structure does not have the <code>ALIGN</code> keyword. ¹ Use the <code>ALIGN</code> keyword on the RPG data structure if the C struct does not have the <code>_Packed</code> qualifier.
pointer to function	POINTER(*PROC)	* with keyword PROCPTR	16	A 16-byte pointer
<p>Note: ¹ Classes with <code>virtual</code> functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.</p>				

Table 48 on page 430 shows the ILE C++ data-type compatibility with OPM RPG.

Table 48. ILE C++ Data-Type Compatibility with OPM RPG

ILE C++ declaration in prototype	OPM RPG I spec, DS subfield columns spec	Length	Comments
char[n]	1 10	n	An array of characters where n=1 to 32766
char	*INxxxx	1	An indicator which is a variable starting with *IN
char[n]	1 nd (d>=0)	n	A zoned decimal The limit of n is 30
_Packed struct {unsigned short i; char[n]}	Not supported.	n+2	A variable length field where i is the intended length and n is the maximum length.
_Packed struct {unsigned int i; char[n]}	Not supported.	n+4	A variable length field where i is the intended length and n is the maximum length.
wchar_t[n]	Not supported.	2n	An array of UCS-2 characters.
_Packed struct {unsigned short i; wchar_t[n]}	Not supported.	2n+2	A variable length UCS-2 field where i is the intended length and n is the maximum length.
_Packed struct {unsigned int i; wchar_t[n]}	Not supported.	2n+4	A variable length UCS-2 field where i is the intended length and n is the maximum length.
wchar_t[n]	Not supported.	2n	An array of graphic characters.

Table 48. ILE C++ Data-Type Compatibility with OPM RPG (continued)

ILE C++ declaration in prototype	OPM RPG I spec, DS subfield columns spec	Length	Comments
<code>_Packed struct {unsigned short i; wchar_t[n]}</code>	Not supported.	2n+2	A variable length graphic field where i is the intended length and n is the maximum length.
<code>_Packed struct {unsigned int i; wchar_t[n]}</code>	Not supported.	2n+4	A variable length graphic field where i is the intended length and n is the maximum length.
<code>char[n]</code>	char	6, 8, 10	A date field
<code>char[n]</code>	char	8	A time field
<code>char[n]</code>	char	26	A time stamp field.
<code>short int</code>	B 1 20	2	A 2-byte signed integer with a range of -9999 to +9999
<code>int</code>	B 1 40	4	A 4-byte signed integer with a range of -999999999 to +999999999
<code>long int</code>	B 1 40	4	A 4-byte signed integer with a range of -999999999 to +999999999
<code>long long int</code>	Not supported.	8	An 8-byte integer field.
<code>struct {unsigned int : n}x;</code>	Not supported	1, 2, 4	A 4-byte unsigned integer, a bitfield
<code>float</code>	Not supported	4	A 4-byte floating point
<code>double</code>	Not supported	8	An 8-byte floating point
<code>long double</code>	Not supported	8	An 8-byte floating point
<code>_Decimal32</code>	Not supported	4	A 4-byte decimal floating point.
<code>_Decimal64</code>	Not supported	8	An 8-byte decimal floating point.
<code>_Decimal128</code>	Not supported	16	A 16-byte decimal floating point.
<code>enum</code>	Not supported	1, 2, 4	Enumeration
<code>*</code>	Not supported	16	A pointer
<code>_DecimalT<n,p></code>	P 1 n/2+1d	n/2+1	A packed decimal. n must be less than or equal to 30. In C++, this is a binary coded decimal class and not a data type.
<code>union.element</code>	data structure subfield starting at position 1	length of longest union member	An element of a union
<code>data_type[n]</code>	E-SPEC array	16	An array to which C++ passes a pointer

Table 48. ILE C++ Data-Type Compatibility with OPM RPG (continued)

ILE C++ declaration in prototype	OPM RPG I spec, DS subfield columns spec	Length	Comments
struct or class	data structure	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer

Note: ¹All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.

Table 49 on page 432 shows the ILE C++ data-type compatibility with ILE COBOL.

Table 49. ILE C++ Data-Type Compatibility with ILE COBOL

ILE C++ declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
char[n] char *	PIC X(n).	n	An array of characters where n=1 to 3,000,000
char	PIC 1 INDIC ..	1	An indicator
char[n]	PIC S9(n) DISPLAY.	n	A zoned decimal
wchar_t[n]	PIC G(n) DISPLAY.	2n	A graphic data type
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	A variable length field where i is the intended length and n is the maximum length.
char[n]	PIC X(n).	6	A date field
char[n]	PIC X(n).	5	A day field
char	PIC X.	1	A day-of-week-field
char[n]	PIC X(n).	8	A time field
char[n]	PIC X(n).	26	A time stamp field
short int	PIC S9(4) COMP-4.	2	A 2-byte signed integer with a range of -9999 to +9999
short int	PIC S9(4) BINARY.	2	A 2-byte signed integer with a range of -9999 to +9999
int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
int	PIC S9(9) BINARY.	4	A 4-byte signed integer with a range of -999999999 to +999999999
int	USAGE IS INDEX	4	A 4-byte integer

Table 49. ILE C++ Data-Type Compatibility with ILE COBOL (continued)

ILE C++ declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
long int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
long int	PIC S9(9) BINARY.	4	A 4-byte signed integer with a range of -999999999 to +999999999
struct {unsigned int : n}x;	PIC 9(9) COMP-4. PIC X(4).	1, 2, 4	Bitfields can be manipulated using hex literals
float	USAGE IS COMP-1	4	A 4-byte floating point
double	USAGE IS COMP-2	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
*	USAGE IS POINTER	16	A pointer
_DecimalT<n,p>	PIC S9(n-p)V9(p) COMP-3.	n/2+1	A packed decimal. In C++, this is a binary coded decimal class and not a data type.
_DecimalT<n,p>	PIC S9(n-p) 9(p) PACKED-DECIMAL.	n/2+1	A packed decimal. In C++ this is a binary coded decimal class not a data type.
union.element	REDEFINES	length of longest union member	An element of a union
data_type[n]	OCCURS	n times the length of the data type	An array to which C++ passes a pointer
struct or class	OCCURS ...DEPENDING ON	variable	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
struct or class	01 record 05 field1 05 field2	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	PROCEDURE-POINTER	16	A 16-byte pointer to a procedure
Not supported.	PIC S9(18) COMP-4.	8	An 8-byte integer
Not supported.	PIC S9(18) BINARY.	8	An 8-byte integer

Note: ¹All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C, all nested structures are packed.

Table 50 on page 434 shows the ILE C++ data-type compatibility with OPM COBOL.

<i>Table 50. ILE C++ Data-Type Compatibility with OPM COBOL</i>			
ILE C++ declaration in prototype	OPM COBOL LINKAGE SECTION	Length	Comments
char[n] char *	PIC X(n) .	n	An array of characters where n=1 to 3,000,000
char	PIC 1 INDIC ..	1	An indicator
char[n]	PIC S9(n) USAGE IS DISPLAY	n	A zoned decimal The limit of n is 18.
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	A variable length field where i is the intended length and n is the maximum length
char[n]	PIC X(n) .	6, 8, 10	A date field
char[n]	PIC X(n) .	8	A time field
char[n]	PIC X(n) .	26	A time stamp field
short int	PIC S9(4) COMP-4.	2	A 2-byte signed integer with a range of -9999 to +9999.
int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
long int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999
struct {unsigned int : n}x;	PIC 9(9) COMP-4. PIC X(4) .	1, 2, 4	Bitfields can be manipulated using hex literals.
float	Not supported	4	A 4-byte floating point
double	Not supported	8	An 8-byte floating point
long double	Not supported	8	An 8-byte floating point
enum	Not supported	1, 2, 4	Enumeration
*	USAGE IS POINTER	16	A pointer
_DecimalT<n, p >	PIC S9(n-p)V9(p) COMP-3.	n/2+1	A packed decimal The limits of n and p are 18. In C++, this is a binary coded decimal class and not a data type.
union.element	REDEFINES	length of longest union member	An element of a union
data_type[n]	OCCURS	n times the length of the data type	An array to which C++ passes a pointer

Table 50. ILE C++ Data-Type Compatibility with OPM COBOL (continued)

ILE C++ declaration in prototype	OPM COBOL LINKAGE SECTION	Length	Comments
struct or class	OCCURS ...DEPENDING ON	variable	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
struct or class	01 record	n	A structure. The structure must be packed. ¹ Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported	16	A 16-byte pointer
Not supported.	PIC S9(18) COMP-4.	8	An 8-byte integer
<p>Note: ¹All structures must be packed. Classes with virtual functions cannot be directly represented in any other language. Nested structures must be explicitly packed. In C all nested structures are packed.</p>			

Ensuring thread safety (C++)

If you are building multithreaded C++ applications, there are some thread-safety issues which you need to consider when using objects defined in the C++ Standard Template Library or in the stream classes.

Ensuring thread safety of template objects

The following headers in the Standard Template Library are reentrant:

- **algorithm**
- **deque**
- **functional**
- **iterator**
- **list**
- **map**
- **memory**
- **numeric**
- **queue**
- **set**
- **stack**
- **utility**
- **valarray**
- **vector**

ILE C++ supports reentrancy to the extent that you can safely read a single object from multiple threads simultaneously. This level of reentrancy is intrinsic. No locks or other globally allocated resources are used.

However, the headers are not reentrant in these cases:

- A single container object is written by multiple threads simultaneously.
- A single container object is written in one thread, while being read in one or more other threads.

If multiple threads write to a single container, or a single thread writes to a single container while other threads are reading from that container, it is your responsibility to serialize access to this container. If multiple threads read from a single container, and no processes write to the container, no serialization is necessary.

Ensuring thread safety of string objects

The classes declared in the **string** standard library are not thread-safe by default and thus may not work correctly in a multithreaded application.

If thread-safety is desired, the following macro variable should be defined prior to the inclusion of the **string** header.

__MULTI__

The macro can be defined within the source code or with a `DEFINE('__MULTI__')` parameter on the `CRTCPMOD` or `CRTBNDCPP` command.

When an application is built with this macro defined, standard library routines used with string objects are thread safe.

Ensuring thread safety of stream objects

All classes declared in the **iostream** standard library are reentrant, and use a single lock to ensure thread-safety while preventing deadlock from occurring. However, on multiprocessor machines, there is a

chance, although rare, that livelock can occur when two different threads attempt to concurrently access a shared stream object, or when a stream object holds a lock while waiting for input (for example, from the keyboard). If you want to avoid the possibility of livelock, you can disable locking in input stream objects, output stream objects, or both, by using the following macros at compile time:

__NOLOCK_ON_INPUT

Disables input locking.

__NOLOCK_ON_OUTPUT

Disables output locking.

To use one or both of these macros, specify the macro name with the **DEFINE** option on the compilation command line. For example:

```
CRTCPMOD DEFINE('__NOLOCK_ON_INPUT') DEFINE('__NOLOCK_ON_OUTPUT')
```

However, if you disable locking on input or output objects, it is your responsibility to provide the appropriate locking mechanisms in your source code if stream objects are shared between threads. If you do not, the behavior is undefined, with the possibility of data corruption or application crash.

Related information

For additional information about topics related to ILE C programming, refer to the following IBM publications:

- The IBM i Information Center describes creating and managing projects defined for the Application Development Manager/400 feature, as well as using the program to develop applications. View Programmer information, ADTS.
- *ADTS for AS/400: Source Entry Utility, SC09-2605-00*, provides information about using the Application Development ToolSet/400 source entry utility (SEU) to create and edit source members. The manual explains how to start and end an SEU session and how to use the many features of this full-screen text editor. The manual contains examples to help both new and experienced users accomplish various editing tasks, from the simplest line commands to using pre-defined prompts for high-level languages and data formats.
- *Application Display Programming, SC41-5715-02*, provides information about:
 - Using DDS to create and maintain displays for applications;
 - Creating and working with display files on the system;
 - Creating online help information;
 - Using UIM to define panels and dialogs for an application;
 - Using panel groups, records, or documents
- *Recovering your system, SC41-5304-10*, provides performance information about backup and recovery operations. Also includes advanced recovery topics, such as saving and restoring to a different release, and disaster recovery techniques.
- *CICS for iSeries Application Programming Guide, SC41-5454-02*, provides information on application programming for CICS/400. It includes guidance and reference information on the CICS® application programming interface and system programming interface commands, and gives general information about developing new applications and migrating existing applications from other CICS platforms.
- *ILE C/C++ for AS/400 MI Library Reference, SC09-2418-00*, provides information on Machine Interface instructions available in the ILE C compiler that provide system-level programming capabilities.
- *CL Programming, SC41-5721-06*, provides a wide-ranging discussion of programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- *Communications Management, SC41-5406-02*, provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.
- *Experience RPG IV Multimedia Tutorial, SK2T-2700*, is an interactive self-study program explaining the differences between RPG III and RPG IV and how to work within the new ILE environment. An accompanying workbook provides additional exercises and doubles as a reference upon completion of the tutorial. ILE RPG code examples are shipped with the tutorial and run directly on the system.
- *GDDM Programming Guide, SC41-0536-00*, provides information about using graphical data display manager (GDDM) to write graphics application programs. Includes many example programs and information to help users understand how the product fits into data processing systems.
- *GDDM Reference, SC41-3718-00*, provides information about using graphical data display manager (GDDM) to write graphics application programs. This manual provides detailed descriptions of all graphics functions available in GDDM. Also provides information about high-level language interfaces to GDDM.

- *ICF Programming, SC41-5442-00*, provides information needed to write application programs that use system communications and the intersystem communications function (ICF). Also contains information on data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- *IDDU Use, SC41-5704-00*, describes how to use the interactive data definition utility (IDDU) to describe data dictionaries, files, and records to the system. Includes:
 - An introduction to computer file and data definition concepts
 - An introduction to the use of IDDU to describe the data used in queries and documents
 - Representative tasks related to creating, maintaining, and using data dictionaries, files, record formats, and fields
 - Advanced information about using IDDU to work with files created on other systems and information about error recovery and problem prevention.
- *IBM Rational® Development Studio for i: ILE C/C++ Compiler Reference, SC09-4816-05*, provides reference information for the ILE C/C++ compiler. It includes compiler options, ILE C/C++ macros, preprocessor directives, and pragmas.
- *IBM Rational Development Studio for i: ILE C/C++ Language Reference, SC09-7852-02*, provides reference information about the ILE C/C++ compiler, including elements of the language, statements, and preprocessor directives. Examples are provided and considerations for programming are also discussed.
- *Standard C/C++ Library Reference, SC09-4949-01*, provides reference information about the standard C/C++ language, statements, and preprocessor directives.
- *ILE C/C++ Runtime Library Functions, SC41-5607-04*, provides reference information about ILE C library functions, including Standard C library functions and ILE C library extensions. Examples are provided and considerations for programming are also discussed.
- *IBM Rational Development Studio for i: ILE COBOL Programmer's Guide, SC09-2540-07*, provides information about how to write, compile, bind, run, debug, and maintain ILE COBOL programs on the system. It provides programming information on how to call other ILE COBOL and non-ILE COBOL programs, share data with other programs, use pointers, and handle exceptions. It also describes how to perform input/output operations on externally attached devices, database files, display files, and ICF files.
- *IBM Rational Development Studio for i: ILE COBOL Reference, SC09-2539-07*, provides a description of the ILE COBOL programming language. It provides information on the structure of the ILE COBOL programming language and the structure of an ILE COBOL source program. It also provides a description of all Identification Division paragraphs, Environment Division clauses, Data Division clauses, Procedure Division statements, and Compiler-Directing statements.
- *ILE Concepts, SC41-5606-09*, explains concepts and terminology pertaining to the Integrated Language Environment architecture of the operating system. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *IBM Rational Development Studio for i: ILE RPG Programmer's Guide, SC09-2507-08*, provides information about the ILE RPG programming language, which is an implementation of the RPG IV language in the Integrated Language Environment (ILE) on the system. It includes information on creating and running programs, with considerations for procedure calls and interlanguage programming. The guide also covers debugging and exception handling and explains how to use system files and devices in RPG programs. It also includes information on migration to RPG IV and sample compiler listings. It is intended for people with a basic understanding of data processing concepts and of the RPG language.
- *IBM Rational Development Studio for i: ILE RPG Reference, SC09-2508-08*, provides information about the ILE RPG programming language. This manual describes, position by position and keyword by keyword, the valid entries for all RPG IV specifications, and provides a detailed description of all the operation codes and built-in functions. This manual also contains information on the RPG logic cycle, arrays and tables, editing functions, and indicators.

- *Local Device Configuration, SC41-5121-00*, provides information about configuring local devices on the system. This includes information on how to configure the following:
 - Local work station controllers (including twinaxial controllers)
 - Tape controllers
 - Locally attached devices (including twinaxial devices)
- *Machine Interface Functional Reference, SC41-5810-00*, describes the machine interface instruction set. It also describes the functions that can be performed by each instruction and also the necessary information to code each instruction.
- *Printer Device Programming, SC41-5713-06*, provides information to help you understand and control printing. Provides specific information on printing elements and concepts of the system, printer file and print spooling support for printing operations, and printer connectivity. Includes considerations for using personal computers, other printing functions such as Business Graphics Utility (BGU), Advanced Function Printing (AFP), and examples of working with the system printing elements such as how to move spooled output files from one output queue to a different output queue. It also includes a list of control language (CL) commands used to manage printing workload. Fonts available for use with the system are also provided. Font substitution tables provide a cross-reference of substituted fonts if attached printers do not support application-specified fonts.
- *REXX/400 Programmer's Guide, SC41-5728-00*, provides a wide-ranging discussion of programming with REXX on the system. Its primary purpose is to provide useful programming information and examples to those who are new to REXX and to provide those who have used REXX in other computing environments with information about the REXX implementation.
- *IBM Rational Development Studio for i: ILE RPG Programmer's Guide, SC09-2507-08*, provides information needed to design, code, compile, and test RPG programs on the system. The manual provides information on data structures, data formats, file processing, multiple file processing, the automatic report function, RPG command statements, testing and debugging functions, application design techniques, problem analysis, and compiler service information. The differences between the RPG for compiler, the System/38 environment RPG III compiler, and the System/36-compatible RPG II compiler are also discussed.
- *Security reference, SC41-5302-11*, tells how system security support can be used to protect the system and the data from being used by people who do not have the proper authorization, protect the data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.
- *Local Device Configuration, SC41-5121-00*, provides step-by-step procedures for initial installation, installing licensed programs, program temporary fixes (PTFs), and secondary languages from IBM. This manual is also for users who already have a system with an installed release and want to install a new release.
- The IBM i Information Center provides information for the experienced application and system programmers who want to use the IBM i application programming interfaces (APIs), as well as examples to help the programmer use APIs.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

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

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This ILE C/C++ Programmer's Guide publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of the ILE C/C++ compiler.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "[Copyright and trademark information](http://www.ibm.com/legal/copytrade.shtml)" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux[®] is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java[™] and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Index

Special Characters

- [__ptr128](#) [393](#)
- [__ptr64](#) [393](#)
- [_C_TS_malloc\(\)](#) [394](#)
- [_C_TS_free\(\)](#) [394](#)
- [_C_TS_malloc\(\)](#) [394](#)
- [_C_TS_realloc\(\)](#) [394](#)
- [_CTLA_HANDLE](#) control action [253](#)
- [_CTLA_HANDLE_NO_MSG](#) control action [254](#)
- [_CTLA_IGNORE](#) control action [254](#)
- [_CTLA_IGNORE_NO_MSG](#) control action [254](#)
- [_CTLA_INVOKE](#) control action [253](#)
- [_EXCP_MSGID](#) global variable [251](#)
- [_Racquire\(\)](#) function [216](#), [226](#), [228](#)
- [_Rclose\(\)](#) function [226](#)
- [_Rdevatr](#) [212](#)
- [_Rdevatr\(\)](#) function [226](#)
- [_Rfeod\(\)](#) function [226](#), [237](#)
- [_Rfeov\(\)](#) function [237](#)
- [_Rformat\(\)](#) function [226](#)
- [_Rindara\(\)](#) function [226](#)
- [_Riofbk\(\)](#) function [154](#), [220](#), [226](#)
- [_Ropen\(\)](#) function [137](#), [140](#), [141](#), [225](#)
- [_Ropnfbk\(\)](#) function [154](#)
- [_Rpgmdev\(\)](#) function [218](#), [226](#), [228](#)
- [_Rread\(\)](#) function [226](#)
- [_Rreadindv\(\)](#) function [226](#), [228](#)
- [_Rreadn\(\)](#) function [226](#)
- [_Rreadnc\(\)](#) function [226](#)
- [_Rrelease\(\)](#) function [226](#), [228](#)
- [_Rupdate\(\)](#) function [226](#)
- [_Rupfb\(\)](#) function [226](#)
- [_Rwrite\(\)](#) function [226](#)
- [_Rwrited\(\)](#) function [226](#)
- [_Rwriterd\(\)](#) function [226](#)
- [_Rwrread\(\)](#) function [226](#)
- [#include](#) directive [179](#), [390](#)
- [#pragma](#) argument directive [297](#), [311](#), [316](#)
- [#pragma](#) cancel_handler directive [271](#)
- [#pragma](#) convert directive [407](#), [409](#), [411](#), [413](#)
- [#pragma](#) descriptor directive [333](#)
- [#pragma](#) disable_handler directive [252](#)
- [#pragma](#) enum [307](#)
- [#pragma](#) exception_handler directive [59](#), [251](#), [252](#), [254](#)
- [#pragma](#) implementation directive [389](#), [391](#)
- [#pragma](#) linkage (PGMNAME, OS) directive [313](#)
- [#pragma](#) linkage directive [297](#), [298](#)
- [#pragma](#) map directive [285](#), [297](#), [313](#), [317](#)
- [#pragma](#) mapinc directive
 - creating C structure type definitions [179](#)
 - database files [193](#)
 - header description [180](#)
 - long record field names [184](#)
 - multiple record formats [193](#)
 - record format [181](#)
- [#pragma](#) nosigtrunc directive [365](#)

- [#pragma](#) pack directive [304](#)
- [#pragma](#) undeclared directive [392](#)

Numerics

- 64-bit version of IFS [163](#)
- 64-bit version of IFS interface, enabling [177](#)

A

- activation group, description [9](#)
- activation groups
 - *CALLER [52](#)
 - *NEW [51](#)
 - abnormal program end [50](#)
 - creation [42](#)
 - default heap [54](#)
 - deleting named [53](#)
 - description [50](#)
 - dynamic storage [54](#)
 - identifying [18](#)
 - impact on performance [72](#)
 - naming [43](#)
 - process [42](#)
 - reasons to use [21](#)
 - specifying [50](#)
- activation groups, named
 - and call stack [53](#)
 - creating [50](#)
 - deleting [53](#), [54](#)
 - impact on performance [72](#)
 - non-ISO compliant behaviour [52](#)
 - runtime behaviour [51](#)
- activation, description [8](#)
- activations
 - description [21](#), [42](#)
 - grouping [43](#)
 - processes [43](#)
- Add Program (ADDPGM) command [102](#)
- address overflows [69](#)
- ALIAS keyword [181](#)
- alignment problems in C structures, avoiding [189](#)
- Allocate Object(ALCOBJ) command [200](#)
- arguments
 - packed decimal [359](#), [361](#)
 - passed in registers [61](#)
 - passing
 - default styles [319](#)
 - from CL program to ILE C++ program [321](#)
 - ILE, CL, COBOL, or RPG linkage [317](#)
 - matching data type requirements [332](#)
 - operational descriptors [320](#), [332](#)
 - packed decimal [359](#)
 - pointers as arguments [281](#)
- arrays, evaluating [119](#)

B

- [bcd.h file](#) [298](#)
- [Binary Coded Decimal Class Library for IBM i](#) [298](#)
- [binary stream files](#)
 - [binary stream database files](#)
 - [I/O considerations](#) [203](#)
 - [binary stream display files](#)
 - [program devices](#) [225](#)
 - [binary stream ICF files](#)
 - [I/O considerations](#) [227](#)
 - [program devices](#) [227](#)
 - [binary stream save files](#)
 - [I/O considerations](#) [243](#)
 - [binary stream subfiles](#)
 - [I/O considerations](#) [225](#)
 - [binary stream tape files](#)
 - [I/O considerations](#) [236](#)
 - [opening, one character at a time](#) [146](#)
 - [opening, one character at a time \(C++\)](#) [147](#)
 - [opening, one record at a time](#) [151](#)
 - [opening, one record at a time \(C++\)](#) [152](#)
 - [reading, one character at a time](#) [148](#)
 - [reading, one record at a time](#) [153](#)
 - [updating, one character at a time](#) [149](#)
 - [writing, one character at a time](#) [148](#)
 - [writing, one record at a time](#) [152](#)
- [binary streams](#) [166](#)
- [binary streams, described](#) [166](#)
- [bindable APIs](#)
 - [to free or re-allocate storage](#) [55](#)
 - [to manage the default heap](#) [55](#)
- [bindable APIs for using teraspace](#) [394](#)
- [binder language](#)
 - [creating](#) [30, 34](#)
 - [EXPORT keyword](#) [30](#)
 - [EXPORT symbol](#) [78](#)
 - [Export Symbol \(EXPORT\) command](#) [38](#)
 - [reason to use](#) [22](#)
 - [source file](#) [28](#)
 - [source file creation](#) [29](#)
 - [to create a service program](#) [32](#)
- [binder listings, using](#) [19](#)
- [binder, using](#) [17](#)
- [binding directories](#)
 - [creating](#) [17](#)
 - [description](#) [17](#)
 - [reasons for creating](#) [17](#)
- [binding modules into a program](#) [14, 15](#)
- [bit fields](#)
 - [evaluating](#) [119, 125](#)
- [breakpoints](#)
 - [conditional](#) [105](#)
 - [job](#) [106](#)
 - [removing \(clearing\) all](#) [109](#)
 - [setting](#) [105](#)
 - [thread](#) [106](#)
 - [unconditional](#) [105](#)

C

- [C linkage, specifying for a function](#) [318](#)
- [C locale migration table](#) [415](#)
- [C++ calling conventions](#) [284](#)
- [C++ I/O stream classes](#) [65](#)
- [C++ objects](#)
 - [using in a C program](#) [288, 292](#)
- [C++ templates](#) [383](#)
- [Call \(CALL\) command](#)
 - [changes to parameters](#) [46](#)
 - [passing parameters to a program](#) [44, 46](#)
 - [using](#) [44](#)
- [call message queue](#) [247](#)
- [call stack](#) [53, 284](#)
- [call stack entries](#) [284](#)
- [calling](#)
 - [C++ procedures](#) [287](#)
 - [C++ programs](#) [287](#)
 - [call stack entries](#) [284](#)
 - [ILE C++ programs](#) [328](#)
 - [ILE programs](#) [322](#)
 - [message queue](#) [247](#)
 - [OPM programs](#) [348](#)
 - [procedures](#) [155, 331](#)
 - [procedures, using linkage specification](#) [353](#)
 - [programs](#) [44, 46](#)
 - [programs, using library qualifications](#) [286](#)
 - [programs, using linkage specification](#) [314, 354](#)
- [calling conventions \(C++\)](#) [284](#)
- [cancel handlers, using \(C++\)](#) [271](#)
- [casting pointers, description](#) [280](#)
- [Change Command Defaults \(CHGCMDDFT\) command](#) [14, 42](#)
- [Change Debug \(CHGDBG\) command](#) [102, 103](#)
- [Change Service Program \(CHGSRVPGM\) command](#) [24](#)
- [Change Tape File \(CHGTAPF\) command](#) [236](#)
- [character arrays, displaying](#) [124](#)
- [Character Data Representation Architecture \(CDRA\)](#) [407](#)
- [character escape codes](#) [410](#)
- [character sets](#) [407, 414](#)
- [characters](#)
 - [case conversion](#) [414](#)
 - [classification](#) [414](#)
 - [coded character sets](#) [407](#)
 - [collating](#) [414](#)
 - [evaluating arrays](#) [119](#)
 - [graphic](#) [407](#)
 - [ordering](#) [414](#)
- [CHGMOD command](#) [135](#)
- [CL linkage](#) [317](#)
- [classes](#)
 - [accessing a C++ class from a C program](#) [292](#)
 - [creating for use in ILE C++](#) [291](#)
 - [evaluating](#) [119](#)
 - [extended_type_info](#) [405](#)
 - [ILE C++](#) [291](#)
 - [mapping a C++ class to a C structure](#) [291](#)
 - [type_info](#) [404](#)
- [COBOL linkage](#) [317](#)
- [Coded Character Set Identifiers \(CCSID\)](#)
 - [CCSID 037](#) [408, 409](#)
 - [CCSID 1026](#) [407](#)
 - [CCSID 1100](#) [413](#)
 - [CCSID 1301](#) [413](#)

Coded Character Set Identifiers (CCSID) (*continued*)

- CCSID 13488 [410](#)
- CCSID 273 [408](#)
- CCSID 290 [407](#), [413](#)
- CCSID 5026 [413](#)
- CCSID 65535 [408](#), [409](#)
- CCSID 905 [407](#)
- CCSID 930 [413](#)
- changing [408](#)
- Character Data Representation Architecture (CDRA) [407](#)
- code points [407](#)
- codepages [407](#)
- definition [407](#)
- exceptions [13](#)
- graphic characters [407](#)
- ILE C/C++ compiler recognition [407](#)
- mixed-byte CCSIDs [419](#)
- other than CCSID 037 [407](#)
- single-byte CCSIDs [419](#)
- source file conversion [407](#)
- specifying [408](#)
- Target CCSID (TGCCSID) parameter [412](#), [413](#)
- trigraphs in place of C characters [13](#)
- wide-character literals [410](#)

coded character sets [414](#)

codepages [407](#)

coding procedures and data items [78](#)

command line, debug [90](#)

command processing program (CPP) [322](#)

commitment control [208](#)

common mechanism to return function results [313](#)

compile time errors [364](#)

Compress Object (CPROBJ) command [24](#), [72](#)

condition handling

- description [258](#)
- registering [258](#)
- service programs [259](#)
- to promote an exception [262](#)

conditional breakpoints

- adding [107](#), [108](#)
- setting to a statement [108](#)

contiguous address ranges [392](#)

control actions [253](#)

control language (CL) commands

- Add ICF Device (ADDICFDEVE) [228](#)
- Add Program (ADDPGM) [102](#)
- Allocate Object (ALCOBJ) [200](#)
- Break [116](#)
- Call (CALL) [115](#), [208](#)
- Change Command Defaults (CHGCMDDFT) [14](#), [42](#)
- Change Debug (CHGDBG) [102](#), [103](#)
- Change ICF File (CHGICFF) [228](#)
- Change Service Program (CHGSRVPGM) [24](#)
- Change Tape File (CHGTAPF) [236](#)
- checking syntax before running [46](#)
- CHGMOD [135](#)
- Compress Object (CPROBJ) [24](#), [72](#)
- Copy File (CPYF) [13](#)
- Copy to Stream File (CPYTOSTMF) [167](#)
- Create a Display File (CRTDSPF) [225](#)
- Create Binding Directory (CRTBNDDIR) [17](#)
- Create Bound C Program (CRTBNDC) [14](#), [42](#), [100](#)
- Create Bound C++ Program (CRTBNDCPP) [14](#), [42](#)
- Create C Module (CRTCMOD) [15](#), [100](#)

control language (CL) commands (*continued*)

- Create C++ Module (CRTCPPMOD) [15](#), [384](#)
- Create Command (CRTCMD) [48](#)
- Create DDM file (CRTDDMF) [202](#)
- Create Display File (CRTDSPF) command [222](#)
- Create ICF File (CR TICFF) [227](#), [228](#)
- Create Journal (CRTJRN) [208](#)
- Create Journal Receiver (CRTJRNRCV) [208](#)
- Create Locale (CRTLOCALE) [418](#), [421](#)
- Create Printer File (CRTPRTF) [233](#)
- Create Program (CRTPGM)
 - default parameters [14](#)
 - specifying parameters [18](#)
- Create Service Program (CRTSRVPGM) [23](#)
- Create Structured Query Language ILE C Object (CRTSQLCI) [14](#)
- Create Tape File (CRTTAPF) [236](#)
- Decompress Object (DCPOBJ) [72](#)
- Display Debug Watches (DSPDBGWCH) [113](#)
- Display Module Source (DSPMODSRC) [102](#)
- End Commitment Control (ENDCMTCTL) [208](#)
- End Debug (ENDDBG) [100](#)
- End Program Export (ENDPGMEXP) [78](#)
- Initialize Physical File Member (INZPFM) [199](#)
- Monitor Message (MONMSG) [50](#)
- Override Database File (OVRDBF) [200](#), [211](#)
- Override Diskette File (OVRDKTF) [239](#)
- Override ICF Device (OVRICFDEVE) [228](#)
- Override ICF File (OVRICFF) [228](#)
- Override Tape File (OVRTAPF) [236](#)
- Reclaim Resources (RCLRSC) [54](#)
- Remove Program (RMVPGM) [102](#)
- Reorganize Physical File Member (RGZPFM) [199](#)
- Retrieve Binder Source (RTVBNDSRC) [30](#)
- Retrieve Job Attributes (RTVJOBA) [313](#), [336](#), [350](#)
- Start Commitment Control (STRCMTCTL) [208](#)
- Start Debug (STRDBG) [100](#), [102](#), [103](#)
- Start Journal Physical File (STRJRNPF) [208](#)
- Start Program Export (STRPGMEXP) [78](#)
- Start Source Entry Utility (STRSEU) [13](#)
- Transfer Control (TFRCTL) [46](#)
- Update Service Program (UPDSRVPGM) [24](#)
- using with service programs [24](#)
- Work with Module (WRKMOD) [134](#)

conventions, calling [283](#), [311](#)

converting

- *CLD objects types to *LOCALE object types [415](#)
- codepages, wide-character literals [411](#)
- from packed decimal data types [355](#)
- packed decimal data stored in character arrays [197](#)
- packed decimal or xoned data, automatically [197](#)
- packed decimal type to floating point type [358](#)
- packed decimal type to integer type [357](#)
- packed decimal type to packed decimal type [355](#)
- REXX variables [46](#)
- source file CCSID [407](#)
- string literals in a source file [409](#)
- xoned decimal data stored in character arrays [197](#)

Copy File (CPYF) command [13](#)

Copy to Stream File (CPYTOSTMF) command [167](#)

Create Binding Directory (CRTBNDDIR) command [17](#)

Create Bound C Program (CRTBNDC) command [100](#)

Create C Module (CRTCMOD) command [100](#)

Create C++ Module (CRTCPPMOD) command [384](#)

- Create Command (CRTCMD) command [48](#)
- Create DDM file (CRTDDMF) command [202](#)
- Create Display File (CRTDSPF) [225](#)
- Create Display File (CRTDSPF) command [222](#)
- Create ICF File (CRTICFF) command [227](#)
- Create Journal (CRTJRN) command [208](#)
- Create Journal Receiver (CRTJRNRCV) command [208](#)
- Create Locale (CRTLOCALE) command [418](#), [421](#)
- Create Printer File (CRTPRPF) command [233](#)
- Create Service Program (CRTSRVPGM) command [23](#)
- Create Structured Query Language ILE C Object (CRTSQLCI) command [14](#)
- Create Tape File (CRTTAPF) command [236](#)
- creating
 - binding directory [17](#)
 - include source view [100](#)
 - listing view [100](#)
 - listing view for debugging [99](#)
 - locale objects [418](#)
 - new commands [48](#)
 - program in one step
 - compiling and binding [14](#)
 - program in two steps
 - compiling and binding [15](#)
 - programs [11](#), [17](#)
 - root source view [100](#)
 - service programs [17](#)
 - source physical file with a CCSID [408](#)
 - temporary module [14](#)
 - views [100](#)

D

- data description specification (DDS)
 - and keyed sequence files [199](#)
 - avoiding duplicate key values [199](#)
 - GENCSRC utility [423](#)
 - support for [423](#)
- data type compatibility [345](#)
- data types
 - bit fields [57](#)
 - choosing [69](#)
 - compatibility [425](#)
 - to improve performance [57](#)
- database files
 - access paths [198](#)
 - arranging key fields [199](#)
 - arrival sequence access paths [198](#)
 - binary stream functions [203](#)
 - commitment control [208](#)
 - comparison with stream file [164](#)
 - definition [197](#)
 - duplicate key values [199](#)
 - field level description [198](#)
 - input and output buffers [182](#)
 - key fields [182](#)
 - keyed sequence access path
 - arranging key fields [199](#)
 - keyed sequence access paths [199](#)
 - managing [198](#)
 - null-capable fields [200](#)
 - opening as binary stream files [202](#)
 - opening as record files [201](#)
 - preventing corruption of [99](#)

- database files (*continued*)
 - record functions [201](#)
 - record level description [198](#)
 - record-at-a-time processing [203](#)
 - source members [198](#)
 - synchronizing changes [208](#)
 - valid keyword parameters [201](#)
- database records
 - arrival sequence processing [203](#)
 - deleting [199](#)
 - evaluating [119](#)
 - keyed sequence processing [204](#)
 - locking [199](#)
 - record I/O functions [206](#)
 - removing deleted records [199](#)
- DBCS characters [410](#)
- DBGVIEW parameter
 - *ALL option [100](#)
 - *EXPMAC option [100](#)
 - *LIST option [100](#)
 - *NONE option [101](#)
 - *SHOWINC option [100](#)
 - *SHOWSKP option [100](#)
 - *SHOWSYS option [100](#)
 - *SHOWUSR option [100](#)
 - *SOURCE option [100](#)
- CRTBNDC command [100](#)
- CRTBNDCPP command [99](#), [100](#)
- CRTCMOD command [100](#)
- CRTCPPMOD command [100](#)
- ILE options [104](#)
- options [89](#)
- preparing a program for debugging [98](#)
- DDM files
 - binary stream functions [203](#)
 - creation [197](#)
 - I/O considerations [202](#)
 - opening as binary files [202](#)
 - opening as record files [201](#)
 - record functions [201](#)
 - valid keyword parameters [201](#)
- debug command line [90](#)
- debug commands
 - ATTR [90](#)
 - BOTTOM [91](#)
 - BREAK [90](#), [107](#)
 - CLEAR [90](#), [108](#), [109](#), [112](#)
 - DISPLAY [91](#)
 - Display Module [104](#)
 - DOWN [91](#)
 - entering [90](#)
 - EQUATE [91](#), [121](#)
 - EVAL [91](#), [118–120](#), [125](#), [129](#), [130](#)
 - EVAL (C++) [131](#)
 - FIND [91](#)
 - HELP [91](#)
 - LEFT [91](#)
 - NEXT [91](#)
 - PREVIOUS [91](#)
 - QUAL [91](#), [118](#)
 - RIGHT [91](#)
 - SET [91](#), [102](#), [103](#)
 - STEP [91](#), [114](#)
 - STEP INTO [114](#), [117](#)

- debug commands (*continued*)
 - STEP OVER [114, 116](#)
 - TBREAK [91, 108, 109](#)
 - THREAD [91](#)
 - TOP [91](#)
 - UP [91](#)
 - using [90](#)
 - WATCH [91, 111](#)
 - where to find them [90](#)
- debug data
 - creating [98](#)
 - effect on object size [89](#)
 - none [101](#)
 - options [89](#)
- debug options
 - setting [103](#)
- debug session
 - adding OPM program to ILE debug session [103](#)
 - adding programs and service programs to [102](#)
 - before starting the debugger [89, 98, 99](#)
 - expression grammar [90](#)
 - limitations [90](#)
 - removing programs and service programs from [102](#)
 - starting [100](#)
 - starting for OPM programs [102](#)
- debug view
 - description [89](#)
 - synchronizing options with listing view [99](#)
- debugging programs
 - adding OPM program to ILE debug session [103](#)
 - before starting [89, 98, 99](#)
 - limitations [90](#)
 - preparing a program for [98](#)
- decimal types [89](#)
- declaring pointer variables [278](#)
- Decompress Object (DCPOBJ) command [72](#)
- deep copy [69](#)
- default program device
 - acquiring [216](#)
 - changing [218](#)
- demangling names [31](#)
- device files
 - both fields [190](#)
 - IBM i feedback areas [212](#)
 - indicator field [191](#)
 - input fields [189](#)
 - output fields [190](#)
 - separate indicator area
 - INDARA keyword [191](#)
 - part of the file buffer [191](#)
 - using INDARA keyword in DDS [213](#)
- different passing methods [311](#)
- direct monitor handlers
 - scoping [252](#)
 - using [251](#)
- diskette files
 - binary stream functions [240](#)
 - blocking [240](#)
 - I/O considerations [239, 240](#)
 - opening as binary stream files [240](#)
 - opening as record files [240](#)
 - record functions [241](#)
- Display Debug Watches (DSPDBGWCH) command [113](#)
- display files
 - binary stream functions [225](#)
 - changing default program device [226](#)
 - creating [222](#)
 - definition [222](#)
 - I/O considerations [222](#)
 - indicators [212](#)
 - major/minor return codes [213](#)
 - open as record files [225](#)
 - opening as binary stream files [224](#)
 - record functions [226](#)
 - separate indicator areas [212](#)
 - subfiles [222](#)
- Display Module Source (DSPMODSRC) command [102](#)
- displaying
 - C++ constructs, sample source [131](#)
 - character arrays [124](#)
 - characters to a newline [124](#)
 - null-ended character arrays [123](#)
 - structure [122](#)
 - system and space pointers, sample source [130](#)
 - templates, while debugging [128](#)
 - value of variables [118](#)
 - variables as hexadecimal values [122](#)
- distributed (DDM)
 - definition [197](#)
 - opening files as binary stream files [202](#)
- Document Library Services (QDLS) [161](#)
- double-byte character set (DBCS) programming
 - considerations
 - Coded Character Set Identifiers (CCSID)
 - double-byte character set (DBCS) [407](#)
- downcasting [401](#)
- dynamic program call, description [284](#)

E

- elements of a language environment [413](#)
- End Commitment Control (ENDCMTCTL) command [208](#)
- ending
 - abnormal program end [50](#)
 - programs [50](#)
- enumerations, evaluating [119, 126](#)
- environment variables
 - locale [420](#)
- equating a name with a variable expression or command [121](#)
- errno
 - checking the value [249](#)
 - header file [249](#)
 - initialization and re-initialization [249](#)
- error macros
 - EIOERROR [249](#)
 - EIORECERR [249](#)
- errors
 - compile time [364, 365](#)
 - language-specific [246](#)
 - messaging support [21, 22](#)
 - packed decimal data type [364](#)
 - runtime [364](#)
- escape codes [410](#)
- EVAL debug command
 - sample source [129](#)
- examples
 - creating a sample ILE C application [74](#)

- examples (*continued*)
 - debugging C applications [92, 97](#)
 - exception percolation [59](#)
- exception classes [253](#)
- exception handling
 - additional types [246](#)
 - direct monitor handlers [59, 251](#)
 - errno values [249](#)
 - HLL-specific handlers [251](#)
 - ILE condition handlers [251](#)
 - priority [247, 269](#)
 - return values [249](#)
 - to improve performance [58](#)
- exception handling mechanism [258](#)
- exception messages
 - C2M [59](#)
 - global variable `_EXCP_MSGID` [251](#)
 - list [246](#)
 - monitoring [50](#)
 - reducing [58](#)
 - specifying identifiers [254](#)
- exceptions
 - nested [270](#)
 - reducing [58](#)
- explicit instantiations of C++ templates [388](#)
- explicit specializations of C++ templates [388](#)
- Export Symbol (EXPORT) command [38, 78](#)
- exports
 - definition [17](#)
 - determining [28](#)
 - updating list of [30, 38](#)
 - using the Display Module command [28](#)
 - working with [28](#)
- external programs, declaring [317](#)
- externally described device files, using [193](#)
- externally described files
 - `#pragma mapinc` directive [179](#)
 - definition [179](#)
 - field level descriptions [198](#)
 - logical database files [193](#)
 - physical database files [193](#)
 - record format name [181](#)
 - using [193](#)
- externally described physical database files [193](#)

F

- `fclose()` function [225](#)
- feedback information [63](#)
- file control
 - text streams and binary streams [142](#)
- file description [139](#)
- file objects [139](#)
- File Server (QFileSvr.400) [162](#)
- files
 - compilations [169](#)
 - database [197](#)
 - distributed (DDM) [197](#)
 - externally described [198](#)
 - file description [139](#)
 - file naming conventions [140](#)
 - file objects [139](#)
 - linking a file stream with [140](#)
 - logical [197](#)

- files (*continued*)
 - long name support [184](#)
 - names, long and short [184](#)
 - opening once for input and output [64](#)
 - physical [65, 197](#)
 - program-described [198](#)
 - record-at-a-time processing [152](#)
 - shared, reducing use of [64](#)
 - sharing [200](#)
 - tempinc files [392](#)
 - template-implementation [391](#)
- `fopen()` function [138, 140, 141, 143, 146, 147, 151, 152, 165, 166, 224, 225, 233](#)
- format
 - date [414](#)
 - monetary quantities [414](#)
 - numbers [414](#)
 - SYSIFCOPT [163](#)
 - system responses [414](#)
 - time [414](#)
- `fread()` function [153, 154, 225](#)
- Free Storage (CEEFRST) bindable API [55](#)
- `freopen()` function [139, 143, 147, 152](#)
- `fropen()` function [225](#)
- function calls [89](#)
- function pointer [278](#)
- functions
 - `_GetExcData` [264](#)
 - `_Racquire()` [226](#)
 - `_Rclose()` [226](#)
 - `_Rdevatr()` [226](#)
 - `_Rfeod()` [226](#)
 - `_Rformat()` [226](#)
 - `_Rindara` [212](#)
 - `_Rindara()` [226](#)
 - `_Riofbk()` [220, 226](#)
 - `_Ropen()` [225](#)
 - `_Rpgmdev()` [218, 226](#)
 - `_Rread()` [226](#)
 - `_Rreadindn()` [226](#)
 - `_Rreadindnc()` [226](#)
 - `_Rreadindv()` [226](#)
 - `_Rreadnc()` [226](#)
 - `_Rrelease()` [226](#)
 - `_Rupdate()` [226](#)
 - `_Rupfb()` [226](#)
 - `_Rwrite()` [226](#)
 - `_Rwrited()` [226](#)
 - `_Rwriterd()` [226](#)
 - `_Rwrread()` [226](#)
- binary stream, for database and DDM files [203](#)
- checking return value [249](#)
- evaluating [119](#)
- `fclose()` [225](#)
- `fopen()` [138, 147, 165, 166, 224, 225, 233](#)
- `fopen()` format [140](#)
- `fread()` [225](#)
- `freopen()` [139](#)
- `fropen()` [225](#)
- `fwrite()` [225, 233](#)
- library functions with a packed decimal data type [361](#)
- locale-sensitive runtime [421](#)
- `recl` parameter [138](#)
- `open()` member [147](#)

functions (*continued*)

- raise [264](#)
- record I/O [62](#)
- setbuf() [138](#)
- setvbuf() [138](#)
- signal [264](#), [265](#)
- signals raised [265](#)
- specifying ILE linkage for [316](#)
- stepping into [117](#)
- stream [65](#)

fwrite() function [225](#), [233](#)

G

GENCSRC utility [180](#)

GENCSRC utility, described [423](#)

GENCSRC utility, level checking [185](#)

Get Heap Storage (CEEGTST) bindable API [55](#)

global variables

- _EXCP_MSGID [251](#)

graphic characters [407](#)

grouping file operations [208](#)

H

handling errors [273](#)

handling the signal [265](#)

header files

- <bcd.h> [366](#)

- <decimal.h> [355](#)

- <errno.h> [249](#)

- <recio.h> [200](#)

- <signal.h> [265](#)

- <stdio.h> [142](#)

- bcd.h [298](#)

- creating with GENCSRC [423](#)

- lecond.h [259](#)

- milib.h [313](#)

- recio.h [137](#)

- stdio.h [155](#)

- xxfdbk.h [154](#), [155](#)

heap storage

- default [54](#), [55](#)

- managing [54](#)

- types [54](#)

- user-created [54](#)

HLL-specific handlers

- using [263](#)

I

I/O considerations

- binary stream database files [203](#)

- binary stream ICF files [227](#)

- binary stream save files [243](#)

- binary stream subfiles [225](#)

- binary stream tape files [236](#)

- DDM files [203](#)

- record diskette files [241](#)

- record subfiles [222](#)

I/O feedback area [154](#)

ICF files

- binary stream functions [227](#)

ICF files (*continued*)

- change the default program device [228](#)

- creation [227](#)

- I/O considerations [227](#)

- indicators [212](#)

- major/minor return codes [213](#)

- open as binary stream files [227](#)

- opening as record files [228](#)

- record functions [229](#)

- record-at-a-time processing [227](#)

- separate indicator areas [212](#)

- usings [227](#)

ILE C function, described [331](#)

ILE C standard [63](#)

ILE condition handlers [259](#)

ILE linkage [317](#)

ILE linkage for a function [316](#)

ILE source debugger

- commands [90](#)

- enabling to accept OPM programs [103](#)

- limitations

 - decimal types [89](#)

 - function calls [89](#)

 - type casts [89](#)

- starting [100](#)

importing

- definition [17](#)

- requests, resolving [19](#)

- unresolved requests

 - circular references [32](#)

 - handling [35](#)

 - program creation order [36](#)

include source view [100](#)

INDARA keyword [213](#)

indicators

- definition [212](#)

- INDARA [212](#)

- option indicators [212](#)

- response indicators [212](#)

- separate indicator areas [212](#)

- specifying as part of the file buffer [212](#)

- types [212](#)

indicators as part of the file buffer

- INDARA keyword [213](#)

indicators in a separate indicator area [213](#)

Initialize Physical File Member (INZPFM) command [199](#)

inlining [61](#)

inlining (C++) [61](#)

integrated file system [163](#)

Integrated File System (IFS)

- 64-bit version [163](#)

- binary streams [166](#)

- compilations [169](#)

- Document Library Services (QDLS) [161](#)

- File Server (QFileSvr.400) [162](#)

- LAN Server/400 (QLANSrv) [161](#)

- Library (QSYS.LIB) [160](#), [165](#)

- OpenSystems (QOpenSys) [159](#)

- Optical support (QOPT) [162](#)

- root (/) [159](#)

- storing data as streams [166](#)

- stream files [164](#)

- stream files and database files [164](#)

- stream files, editing [167](#)

Integrated File System (IFS) (*continued*)
text streams [165](#)
Integrated Language Environment (ILE)
activating groups [21](#)
activation [42](#)
condition handlers [258](#)
inter-language calls [297](#)
international locale support
ILE C support [414](#)
ISO C standard [62](#)

J

job breakpoints [106](#)
journaling environment [208](#)

L

LAN Server/400 (QLANSrv) [161](#)
language environment [413](#)
language environments [418](#)
LC_ALL locale variable [419](#)
LC_COLLATE locale variable [419](#)
LC_CTYPE locale variable [419](#)
LC_MONETARY locale variable [419](#)
LC_NUMERIC locale variable [419](#)
LC_TIME locale variable [419](#)
LC_TOD locale variable [419](#)
level checking, described [184](#)
level checking, GENCSRC utility [185](#)
Library (QSYS.LIB) [160](#), [165](#)
library names, specifying [66](#)
limitations
debug expression grammar [89](#)
ILE source debugger [90](#)
porting programs to ILE C++ [295](#)
linking a file stream with a file [140](#)
listing [100](#)
listing view [99](#)
listing view, creating [100](#)
locale definition
POSIX locale definition [421](#)
POSIX standard [418](#)
POSIX versus SAA [421](#)
SAA locale definition [421](#)
locales
*CLD object types [414](#)
*LOCALE object types [414](#)
customizing [419](#)
environment variables [420](#)
ILE C for AS/400 support [414](#)
international locale support [413](#)
library function [419](#)
overview of ILE C support [414](#)
runtime functions [421](#)
setting an active locale [420](#)
LOCALETYPE option
CRTBNDC command [418](#)
CRTCMOD command [418](#)
locate mode [63](#)
logical files
multi-format [197](#)

M

macros, expanding [61](#)
major/minor return code, checking [250](#)
major/minor return codes [213](#)
mapping
C++ class to C structure [291](#)
internal identifier to IBM i compliant name [285](#)
match data type requirements
by reference [311](#)
by value directly [311](#)
by value indirectly [311](#)
messaging support [21](#)
mixed-byte environment [407](#)
module object
debug data [16](#)
program entry procedure [16](#)
user entry procedure [16](#)
modules
changing the view [105](#)
different views [105](#)
effect of debug data on size [89](#)
observability, changing [133](#)
observability, removing [135](#)
preparing for debugging [98](#)
setting debug options [104](#)
Monitor Message (MONMSG) command [50](#)
multiple record formats, using in a logical file [193](#)

N

name mangling [28](#)
names, mangled [31](#)
naming conventions
#pragma mapinc [184](#)
#pragma mapinc lname option [181](#)
#pragma mapinc name generation [180](#)
field type lvchk [181](#)
long filenames [184](#)
native language [413](#)
nested exceptions [270](#)
newline, displaying [124](#)
no debug data [101](#)
null ended character arrays, displaying [123](#)
null-capable fields [200](#)
numeric escape codes [410](#)

O

observability of modules
and object size [72](#)
removing [71](#), [135](#)
open data path [200](#)
open feedback area [154](#)
Open Systems (QOpenSys) [159](#)
open() function [152](#)
open() member function [140](#), [144](#), [147](#)
opening
database files as binary stream files [202](#)
DDM files as binary stream files [202](#)
display files as record files [225](#)
opening text stream files
modes [143](#)

- operational descriptors [320](#)
- operational descriptors, description [332](#)
- Optical support (QOPT) [162](#)
- optimization level
 - *FULL [134](#)
 - *NONE [134](#)
 - changing [73](#), [134](#)
- optimization process [73](#)
- option indicators [212](#)
- options
 - for performance [73](#), [74](#)
 - INLINE [61](#)
 - OPTIMIZE [73](#)
 - rtncode=y [58](#)
 - speed versus size [73](#)
 - to enable performance measurement [58](#)
- OS linkage function (C++) [317](#)
- overflow behavior [358](#)
- Override Database File (OVRDBF) [211](#)
- Override Database File (OVRDBF) command [200](#)
- Override Diskette File (OVRDKTF) command [239](#)
- Override Tape File (OVRTAPF) command [236](#)

P

- packed decimal data
 - conversion functions [197](#)
 - using [196](#)
- packed decimal data type representation [355](#), [366](#)
- packed decimal data type, C versus C++ support [355](#)
- packed decimal data types
 - creating compatible binary-coded decimal types [298](#)
- packed decimal datatypes
 - #pragma nosigtrunc directive [365](#)
- page faults [69](#)
- parameters
 - default passing styles [319](#)
 - passing [44](#)
 - passing (C++) [315](#)
 - passing from C++ to a different HLL [315](#)
 - using operational descriptors to pass [320](#)
- passing
 - arguments [311](#)
 - different methods [311](#)
 - packed decimal arguments [360](#)
 - packed decimal data to a function [359](#)
 - packed decimal value to a function [358](#)
 - parameters [46](#), [319](#)
 - parameters (C++) [315](#)
 - parameters by using operational descriptors [320](#)
 - parameters from C++ to a different HLL [315](#)
 - pointer to a packed decimal variable to a function [359](#)
 - pointers as arguments [281](#)
 - styles [311](#)
- percolating exceptions [59](#)
- percolation
 - example [59](#)
- performance
 - compile time
 - improving [73](#)
 - function calls [61](#)
 - hooks [58](#)
 - improving
 - avoiding virtual functions [73](#)

- performance (*continued*)
 - improving (*continued*)
 - choosing data types [69](#)
 - I/O considerations [62](#)
 - inline function calls [61](#)
 - reducing dynamic memory allocation calls [69](#)
 - reducing program startup time [72](#)
 - reducing space requirements [69](#)
 - reducing space used for padding [70](#)
 - improving through data types [57](#)
 - improving with minimal code changes [58](#)
 - measuring [58](#)
 - runtime performance [56](#)
- performance improvement [62](#)
- percol() function [249](#)
- physical files [65](#), [197](#)
- pointer casting, description [280](#)
- pointers
 - _null_map [201](#)
 - 16-byte [393](#)
 - 8-byte [393](#)
 - and data models [393](#)
 - casting [280](#)
 - casting constraints [280](#)
 - comparisons [66](#)
 - constraints
 - casting [280](#)
 - declaring pointer variables [278](#)
 - defining [58](#)
 - dynamic casting with [401](#)
 - evaluating [119](#), [125](#)
 - FILE pointer casting [212](#)
 - function
 - with OS-linkage [279](#)
 - IBM i types [276](#)
 - IBM i uses [276](#)
 - ILE constraints [277](#)
 - incompatible pointer types [279](#)
 - ISO C definition [276](#)
 - label [277](#)
 - modifying for teraspace [393](#)
 - open
 - ILE constraints [277](#)
 - impact on performance [66](#)
 - passing pointers [281](#)
 - pointers other than open pointers [277](#)
 - shallow copy versus deep copy [69](#)
 - teraspace [393](#)
 - to improve performance [68](#)
 - to input, output, and key null field maps [200](#)
 - to reduce indirect access [68](#)
 - types [276](#)
- printer files
 - binary stream functions [233](#)
 - FCFC [233](#)
 - I/O considerations [233](#)
 - indicators [212](#)
 - major/minor return codes [213](#)
 - open as binary stream files [233](#)
 - opening as record files [233](#)
 - record functions [234](#)
 - separate indicator areas [212](#)
- procedure calls
 - call stack entries [284](#)

- procedure calls (*continued*)
 - ILE C++ and ILE COBOL procedure calls [313](#)
 - static [16](#)
- procedure pointer calls, described [332](#)
- procedures
 - calling [155](#), [283](#), [331](#)
 - calling, using linkage specification [353](#)
 - non-recursive call [285](#)
 - recursive call [285](#)
 - renaming [285](#)
 - stepping into [117](#)
 - stepping over [116](#)
- Process Commands (QCAPCMD) API [46](#)
- program entry procedure, description [284](#)
- program entry procedures (PEP)
 - as component of program object [16](#)
 - identifying [16](#)
- program source, viewing [104](#)
- program-described files [198](#)
- programs
 - call stack entries [284](#)
 - calling [44](#), [283](#)
 - calling (TFRCTL) [46](#)
 - calling ILE [284](#), [322](#)
 - calling ILE C++ [328](#)
 - calling OPM [348](#)
 - calling, using library qualifications [286](#)
 - calling, using linkage specification [314](#), [354](#)
 - creating [17](#)
 - debugging [99](#), [105](#), [109](#)
 - devices
 - I/O feedback area [220](#)
 - effect of debug data on size [89](#)
 - end [50](#)
 - porting to ILE C++ [295](#)
 - preparing for debugging [98](#), [133](#)
 - preparing for production [133](#)
 - renaming [285](#)
 - running [41](#)
 - running via auser-created CL command [48](#)
 - stepping into [114](#)
 - stepping over [114](#)
 - stepping through [114](#)
 - updating [21](#)
 - versus service programs [22](#)
- public interface [22](#)

Q

- QINLINE [139](#)
- QTEMP library [138](#)
- QXX functions [309](#)

R

- Reallocate Storage (CEECZST) bindable API [55](#)
- recio.h file [137](#)
- Reclaim Resources (RCLRSC) command [54](#)
- record blocking
 - I/O feedback structure [211](#)
 - to improve I/O performance [211](#)

- record blocking (*continued*)
 - turning off [211](#)
 - turning on [211](#)
- record diskette files
 - blocking [241](#)
 - I/O considerations [241](#)
 - reading and writing [241](#)
- record display files
 - I/O considerations [225](#)
- record field names [181](#)
- record files
 - I/O extensions and C++ [137](#)
- record format
 - _Rformat() function [181](#)
 - definition [181](#)
- record ICF files
 - I/O considerations [228](#)
 - program devices [228](#)
- record save files
 - I/O considerations [244](#)
- record subfiles
 - I/O considerations [226](#)
- record tape files
 - blocking [237](#)
 - I/O considerations [237](#)
 - using _Rfeod [237](#)
 - using _Rfeov [237](#)
- Register Call Stack Entry Termination User Exit Procedure (CEERTX) bindable API [271](#)
- Register ILE Condition Handler (CEEHDLR) bindable API [258](#)
- Register Storage class [57](#)
- Remove Program (RMVPGM) command [102](#)
- removing
 - breakpoints [105](#)
 - module observability [135](#)
- Reorganize Physical File Member (RGZPFM) command [199](#)
- reserving storage [188](#)
- response indicators [212](#)
- Retrieve Binder Source (RTVBNDSRC) command [30](#)
- Retrieve Job Attributes (RTVJOBA) command [313](#), [336](#), [350](#)
- return value of a function, checking [249](#)
- returning function results [313](#)
- RIOFB_T [63](#)
- root source [100](#)
- root source view, creating [100](#)
- RPG linkage [317](#)
- RTBND, using to optimize performance (C++) [395](#)
- runtime
 - errors [364](#)
 - limits [74](#)
 - locale-sensitive functions [421](#)
 - suppressing errors [365](#)
- runtime model
 - ISO C standard [42](#)
- runtime storage
 - dynamically allocating (C++) [55](#)
 - managing [54](#)
- RunTime Type Information (RTTI)

RunTime Type Information (RTTI) *(continued)*

- C++ language-defined [401](#)
- dynamic casting [400](#)
- dynamic casting with pointers [401](#)
- dynamic casting with references [402](#)
- dynamic_cast operator [401](#)
- extended_type_info class [405](#)
- extensions [404](#)
- type_info class [404](#)
- typeid operator [402](#)
- using in constructors and destructors [404](#)

S

save files

- binary stream functions [243](#)
- I/O considerations [243](#)
- open as record files [243](#)
- opening as binary stream files [243](#)
- record functions [244](#)
- using [243](#)

scalar variables

- arrays [118](#)
- changing value while debugging [120](#)
- structures [118](#)

segment faults [69](#)

service programs

- automatically putting into debug mode [117](#)
- changing [24](#)
- condition handlers [259](#)
- creating [23](#)
- exports [28](#)
- public interface [22](#)
- reasons for creating [17](#), [22](#)
- related commands [24](#)
- sample program SEARCH [24](#)
- updating [24](#)
- updating export list [30](#), [38](#)
- using a placeholder [37](#)
- using binder language to create [32](#)
- versus programs [22](#)

session handle [155](#)

session manager [155](#)

setbuf() function [138](#)

setting

- active locale [420](#)
- breakpoints [105](#)
- conditional breakpoint to a statement [108](#)
- debug options [103](#)

setvbuf() function [138](#)

shallow copy [69](#)

shared files, reducing use of [64](#)

SIG_DFL [265](#)

SIG_IGN [265](#)

signal

- SIGABRT [265](#)
- SIGFPE [265](#)
- SIGILL [265](#)
- SIGINT [265](#)
- SIGIO [265](#)
- SIGOTHER [265](#)
- SIGSEGV [265](#)
- SIGTERM [265](#)
- SIGUSR1 [265](#)

signal *(continued)*

SIGUSR2 [265](#)

signal handler

- changing the state of nested [264](#)

signal handling [265](#)

signature [22](#)

SLTFLD keyword [185](#)

Source Entry Utility (SEU) [29](#)

source file conversions [407](#)

source statements

- entering [13](#), [437](#)
- entering, using SEU [29](#)
- viewing [104](#)

space pointers, displaying [130](#)

specifying CCSID [408](#)

Start Commitment Control (STRCMTCTL) command [208](#)

Start Debug (STRDBG) command [102](#)

Start Journal Physical File (STRJRNPFF) command [208](#)

Start Program Export (STRPGMEXP) command [78](#)

Start Source Entry Utility (STRSEU) command [13](#)

starting

- OPM debug session [102](#)
- source debug session [100](#)

static procedure call, description [331](#)

stderr [138](#)

stdin [138](#)

stdio.h file [155](#)

stdout [138](#)

STEP debug command

- into [114](#)
- into, example [115](#)
- over [114](#)

stepping

- into a program [114](#)
- into a program, example [115](#)
- into procedures [117](#)
- over a program [114](#)
- over procedures [116](#)
- through program [114](#)

storage

- dynamically allocating at runtime (C++) [55](#)
- exhausted [53](#)

heap [54](#)

managing

- requirements [72](#)

reclaiming [53](#)

runtime

- dynamically allocating (C++) [55](#)
- managing [54](#)

static

- OPM default activation group [53](#)

stream buffering

fully buffered [137](#)

line buffered [137](#)

unbuffered [137](#)

stream files

dynamic creation [138](#)

record lengths [138](#)

stream files, editing [167](#)

strerror() function [249](#)

string literals [315](#), [354](#)

structure, displaying [122](#)

structures

- evaluating [119](#), [126](#)

- subfiles
 - binary stream functions [225](#)
 - definition [222](#)
 - I/O considerations [222](#)
 - opening as binary stream files [224](#)
 - record functions [226](#)
 - using [222](#), [223](#)
- system buffer [63](#)
- system exceptions
 - record files [250](#)
 - stream files [250](#)
- system pointers, displaying [130](#)
- system resources
 - reclaiming [53](#)

T

- tape files
 - binary stream functions [236](#)
 - blocking [236](#)
 - I/O considerations [236](#)
 - open as binary stream files [236](#)
 - opening as record files [237](#)
 - processing [65](#)
 - record functions [237](#)
- TEMPINC [389](#)
- templates
 - explicit specializations [386](#)
 - function definitions [390](#)
 - implementation files [391](#)
 - include files [392](#)
 - instantiating [384](#)
 - instantiations [387](#)
 - instantiations, automatic [385](#), [387](#)
 - using [383](#)
- templates, displaying [128](#)
- teraspaces (C++)
 - 16-byte pointers [393](#)
 - bindable APIs [394](#)
 - C/C++ pointer support [393](#)
 - C++ limitations when overloading replacement functions [56](#)
 - C++ replacement functions [56](#)
 - definition [392](#)
 - determining the environment [392](#)
 - function overloading [400](#)
 - operator delete [56](#), [398](#)
 - operator delete, overloading [56](#)
 - operator new [56](#), [398](#)
 - operator new, overloading [56](#)
 - overloading functions [400](#)
 - overloading operator new or operator delete [56](#)
 - pointer conversions [393](#)
 - replacement functions (C++) [56](#)
 - supported environments [393](#)
- testing
 - for a coding assumption [402](#)
- TEXT keyword [181](#)
- text stream files
 - opening [143](#)
 - opening (C++) [144](#)
 - reading [145](#)
 - reading, example [145](#)
 - updating [146](#)

- text stream files (*continued*)
 - writing [144](#)
- text streams
 - how they are defined [138](#)
 - opening [138](#)
 - printing [139](#)
- text streams, described [165](#)
- thread breakpoints [106](#)
- Transfer Control (TFRCTL) command [46](#)
- trigraphs [410](#)
- try catch throw (C++), using [377](#)
- try-catch-throw (C++), using [271](#)
- type casts [89](#)
- TYPEDEF_PFX keyword [185](#)
- types of handlers [251](#)

U

- unconditional breakpoints
 - removing [105](#)
 - setting [105](#)
- understanding packed decimal data type errors [364](#)
- unicode
 - effect on codepage conversions [411](#)
 - enabling character set support [411](#)
 - support for multiple character literals [410](#)
 - support for wide-character literals [410](#)
- Unregister Call Stack Entry Termination User Exit Procedure (CEETUTX) bindable API [271](#)
- Unregister ILE Condition Handler (CEEHDLU) bindable API [259](#)
- Update Service Program (UPDSRVPGM) command [24](#)
- updating
 - service program export list [38](#)
- user entry procedures (UEP)
 - as component of program object [16](#)
 - identifying [16](#)
- user-defined data stream (UDDS) [222](#)

V

- va_arg macro [361](#)
- variable, expression or command, equating a name [121](#)
- variables
 - assigning an expression to [120](#)
 - assigning values to [121](#)
 - DDS variable names [181](#)
 - debugging [118](#)
 - defining scope [121](#)
 - determining the value of an expression [119](#)
 - displaying accurately [133](#)
 - displaying as hexadecimal values [122](#)
 - displaying values [118](#)
 - environment variables [420](#)
 - equating a name with [121](#)
 - evaluating [119](#), [125](#)
 - global [57](#), [61](#)
 - handling duplicate names [19](#)
 - optimizing [57](#)
 - Register Storage class use [57](#)
 - REXX [46](#)
 - scalar
 - changing value while debugging [120](#)

variables (*continued*)
 static [57](#)
 using F11 to display [119](#)
 volatile qualifier [57](#)
views
 creating [100](#)
 different module views [105](#)
 include source [100](#)
 listing [100](#)
 program source [104](#)
 root source [100](#)
VREF linkage, specifying [317](#)

W

watch conditions
 and conditional breakpoints [109](#)
 example of setting [112](#)
 removing [112](#)
 setting and removing [110](#), [113](#)
watches
 displaying active [113](#)
wide-character literals, codepage conversions [411](#)
wide-character literals, Unicode support for [410](#)

X

xxfdbk.h file [154](#), [155](#)

Z

zoned decimal data
 conversion functions [197](#)
 using [196](#)



Product Number: 5770-WDS

SC09-2712-08

