SYBASE®

ODL Guide

e-Biz Impact™

5.4.5

DOCUMENT ID: DC10099-01-0545-01

LAST REVISED: July 2005

Sybase, Inc., One Sybase Drive, Dublin, CA 94568.

Contents

About This Book

Audience

This book is for e-Biz Impact™ version 5.4.5 application developers.

How to use this book

The following chapters are included in this book:

- Chapter 1, “Overview,” describes the syntax of Object Definition Language (ODL).
- Chapter 2, “Bridged Functions,” describes transaction production and ODL built-in functions.
- Chapter 3, “Message Objects and Methods,” describes the ODL message objects and their associated methods.
- Chapter 4, “Production Objects and Methods,” describes the ODL production objects and their associated methods.
- Chapter 5, “General Objects and Methods,” describes general ODL objects and their associated methods.
- Chapter 6, “Route Calls,” describes the distributed function calls (DFCs) that are used to submit transactions and that allow one e-Biz Impact application to communicate with another.
- Chapter 7, “Accessing WebSphere MQ Data,” describes how to access IBM WebSphere MQ data.
- Chapter 8, “Using Shared Libraries,” explains how to call custom C/C++ shared library functions from ODL applications.
- Appendix A, “Conversion Tables,” provides conversion tables for ASCII and EBCDIC character sets for the three character translation functions EbsAsc, AscEbc and charTranslate.
- Appendix B, “Error Conditions,” provides a list of error conditions for the I/O file objects.
- Appendix C, “Using the Command Line-to-DFC Program,” describes how to use the command line-to-DFC utility.

Related documents

e-Biz Impact documentation The following documents are available on the Sybase™ Getting Started CD in the e-Biz Impact 5.4.5 product container:

- The e-Biz Impact installation guide explains how to install the e-Biz Impact software.
- The e-Biz Impact release bulletin contains last-minute information not documented elsewhere.

e-Biz Impact online documentation The following e-Biz Impact documents are available in PDF and DynaText format on the e-Biz Impact 5.4.5 SyBooks CD:

- The *e-Biz Impact Application Guide* provides information about the different types of applications you create and use in an e-Biz Impact implementation.
- The *e-Biz Impact Authorization Guide* explains how to configure e-Biz Impact security.
- *e-Biz Impact Command Line Tools Guide* describes how to execute e-Biz Impact functionality from a command line.
- The *e-Biz Impact Configurator Guide* explains how to configure e-Biz Impact using the Configurator.
- The *e-Biz Impact Feature Guide* describes new features, documentation updates, and fixed bugs in this version of e-Biz Impact.
- The *e-Biz Impact Getting Started Guide* provides information to help you quickly become familiar with e-Biz Impact.
- The *Monitoring e-Biz Impact* (this book) explains how to use the Global Console, the Event Monitor, and alerts to monitor e-Biz Impact transactions and events. It also describes how e-Biz Impact uses the standard Simple Network Management Protocol (SNMP).
- *Java Support in e-Biz Impact* describes the Java support available in e-Biz Impact 5.4.5.
- The *e-Biz Impact MSG-IDE Guide* describes MSG-IDE terminology and explains basic concepts that are used to build Object Definition Language (ODL) applications.
- The *e-Biz Impact ODL Guide* provides a reference to Object Definition Language (ODL) functions and objects. ODL is a high-level programming language that lets the developer further customize programs created with the IDE tools.
- The *e-Biz Impact TRAN-IDE Guide* describes how to use the TRAN-IDE tool to build e-Biz Impact production objects, which define incoming data and the output transactions produced from that data.

Note The *e-Biz Impact ODL Application Guide* has been incorporated into the *e-Biz Impact ODL Guide*.

The *e-Biz Impact Alerts Guide*, the *e-Biz Impact SNMP Guide*, and the *e-Biz Impact Global Console Guide* have been combined into a new guide—*Monitoring e-Biz Impact*.

Adaptive Server Anywhere documentation The e-Biz Impact installation includes Adaptive Server® Anywhere, which is used to set up a Data Source Name (DSN) used with e-Biz Impact security and authorization. To reference Adaptive Server Anywhere documentation, go to the Sybase Product Manuals Web site at Product Manuals at http://www.sybase.com/support/manuals/, select SQL Anywhere Studio from the product drop-down list, and click Go.

Note See the *SyBooks Installation Guide* on the Getting Started CD, or the *README.txt* file on the SyBooks CD for instructions on installing and starting SyBooks.

Other sources of information

Use the Sybase Getting Started CD, the SyBooks CD, and the Sybase Product Manuals Web site to learn more about your product:

- The Getting Started CD contains release bulletins and installation guides in PDF format, and may also contain other documents or updated information not included on the SyBooks CD. It is included with your software. To read or print documents on the Getting Started CD, you need Adobe Acrobat Reader, which you can download at no charge from the Adobe Web site using a link provided on the CD.
- The SyBooks CD contains product manuals and is included with your software. The Eclipse-based SyBooks browser allows you to access the manuals in an easy-to-use, HTML-based format.

  Some documentation may be provided in PDF format, which you can access through the PDF directory on the SyBooks CD. To read or print the PDF files, you need Adobe Acrobat Reader.
- The Sybase Product Manuals Web site is an online version of the SyBooks CD that you can access using a standard Web browser. In addition to product manuals, you will find links to EBFs/Maintenance, Technical Documents, Case Management, Solved Cases, newsgroups, and the Sybase Developer Network.

  To access the Sybase Product Manuals Web site, go to Product Manuals at http://www.sybase.com/support/manuals/.

Sybase certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

❖ **Finding the latest information on product certifications**

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Select Products from the navigation bar on the left.
3. Select a product name from the product list and click Go.
4. Select the Certification Report filter, specify a time frame, and click Go.
5. Click a Certification Report title to display the report.

❖ **Creating a personalized view of the Sybase Web site (including support pages)**

Set up a MySybase profile. MySybase is a free service that allows you to create a personalized view of Sybase Web pages.

1. Point your Web browser to Technical Documents at http://www.sybase.com/support/techdocs/.
2. Click MySybase and create a MySybase profile.

Sybase EBFs and software maintenance

❖ **Finding the latest information on EBFs and software maintenance**

1. Point your Web browser to the Sybase Support Page at http://www.sybase.com/support.
2. Select EBFs/Maintenance. Enter user name and password information, if prompted (for existing Web accounts) or create a new account (a free service).
3. Select a product.
4. Specify a time frame and click Go.
5. Click the Info icon to display the EBF/Maintenance report, or click the product description to download the software.

Conventions

The syntax conventions used in this manual are:

Key	Definition
commands and methods	Command names, command option names, utility names, utility flags, Java methods/classes/packages, and other keywords are in lowercase Arial font.
variable	Italic font indicates: • Program variables, such as *myServer* • Parts of input text that must be substituted, for example: *Server*.log • File names
File | Save	Menu names and menu items are displayed in plain text. The vertical bar shows you how to navigate menu selections. For example, File | Save indicates "select Save from the File menu."
`package 1`	Monospaced font indicates: • Information that you enter in a graphical user interface, at a command line, or as program text • Sample program fragments • Sample output fragments

Accessibility features

This document is available in an HTML version that is specialized for accessibility. You can navigate the HTML with an adaptive technology such as a screen reader, or view it with a screen enlarger.

For information about how Sybase supports accessibility, see Sybase Accessibility at http://www.sybase.com/accessibility. The Sybase Accessibility site includes links to information on Section 508 and W3C standards.

If you need help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1

Overview

This chapter gives an overview of the Object Definition Language (ODL).

Introduction

The Object Definition Language (ODL) is a high-level programming language that allows developers to customize applications using the e-Biz Impact IDE tools—MSG-IDE and TRAN-IDE. ODL allows the developer to go a layer deeper into the functionality of Application Interface Modules (AIMs) by manipulating data or accessing various object classes and their methods.

ODL applications are event-driven. ODL instantiates objects according user-supplied properties. The developer can interact with objects, procedurally or through callbacks.

Event-driven programming

In traditional (procedural) programming, the developer dictates the order in which events occur and when information is presented to the user. The user has minimal or no control over the program because the code is not attached to specific objects with which the user interacts.

However, in an event-driven program, users have greater control in directing the order in which events occur. In essence, the code for an event-driven program is written in small blocks that are attached to program objects. It is the user interaction with these objects or events that trigger code execution.

Note It is important that the developer have a firm understanding of how event-driven programming differs from traditional programming. However, fin-depth knowledge of these concepts is not necessary to implement ODL effectively.

ODL and IDE tools

Applications created with MSG-IDE and TRAN-IDE require minimal developer coding. Instead, developers can develop an application using the graphical user interfaces to select and build the objects.

Because applications created with the IDE tools are event-driven, each event provides the developer with an opportunity to take control of the application using the ODL code contained within custom functions and using distributed function calls (DFCs) to talk to other applications.

Some event triggers are hard-coded into application objects and always occur at a specific point in the application, while other objects occur based on user interaction with the application. The ODL code within custom functions determines how your application responds to these events and how control flows between application objects.

There are also functions—such as initialization and deinitialization—that are not triggered by events, but are called at predetermined points. You can also use these functions to control an application.

Application program flow

Control of an application flows in a regulated manner between the application's objects (the parts created used the IDE tools) and the ODL code contained within the program's control flow or function objects.

- MSG-IDE – when an application created in MSG-IDE starts, it executes the initialization function (clinit). Once the communication object (clcomm) connects to the endpoint, the protocol object executes the open control flow. As message frames match data, they execute their associated control flows. If the communication object loses the connection to the endpoint, the protocol object executes the close control flow. When the program shuts down, it executes the deinitialization function (cldeinit).

  See the *e-Biz Impact MSG-IDE Guide* for more information about using this tool.

- TRAN-IDE – when a production object generated by TRAN-IDE has a qualification object attached to it, transaction production executes the qualification before generating the output transaction. The production object executes the appropriate error functions whenever a rule object encounters a processing error, and executes filter and qualification objects attached to encountered objects.

  See the *e-Biz Impact TRAN-IDE Guide* for details about using this tool.

ODL syntax

ODL syntax uses keywords that you cannot use in variable or function names. ODL also provides a fixed set of datatypes with decimal, string, and blob used as predefined ODL classes.

Table 1-1 lists ODL keywords. Table 1-2 list ODL datatypes.

Table 1-1: ODL keywords

_cdecl	define	goto	objects	switch
arguments	deinit	idempotent	operator	symoff
auto	dll	if	out	text
blob	do	in	private	token
break	double	include	procedure	typedef
bysymbol	else	init	protected	union
callbacks	end	inline	public	unsigned

case	enum	int	register	value
class	extern	interface	return	version
const	flavor	lcldefine	sizeof	virtual
continue	float	module	static	void
controls	for	new	string	volatile
default	friend	object	struct	while

Table 1-2: ODL datatypes

Type	Format	Size in bytes	Value range
Binary Large Object	blob	< = 4GB	Variable
Character	char	1	0x00 - 0xff or + -127
Decimal	decimal	Up to 64K	A data variable of type decimal may contain up to 65535 characters with the decimal point located anywhere within those characters.
Floating Point	float	8	Platform dependent
Integer	int	4 bytes	+-32767: 0 - 65535
Long Integer	long	4 bytes	+-32767: 0 -65535
Pointer	type	8 bytes	
Short Integer	short	Varies	+-32767: 0 - 65535
String	string	Up to 64K	Variable
Structures	struct		
Symoff	symoff	Varies	Integers 1 - 65535
C Pointer	_cdecl char*	4 bytes,	

Conversions

When you perform an operation with operands that are different datatypes, ODL converts the operand on the right to the datatype of the operand on the left, then performs the operation. ODL then converts the result to the datatype of the variable on the left of the equal sign, if necessary, before placing that result into the variable.

In the example below, the value of `my_float` is converted to an integer before adding it to `my_int`. The result of `my_int + my_float` is then converted to `float` before it is stored as the value for the answer. For example:

```
int my_int = 5;
float my_float = 10.2;
float answr;
answr = my_int + my_float;
```

Storage classes

The following storage classes are available in ODL:

Storage class	Description
auto	All local variables are auto by default unless declared as static. They can be used only within the function where they are declared, and they do not retain their value when the program exits the function.
extern	External variables are global variables. They are available to the entire program, and they retain their values from one usage to the next. You must define an external variable outside of the function and reference it inside the function as extern. If you define the external variable with an IDE tool, as opposed to within a DLL or header file, then you do not need to reference it as extern within the function.
static	Static variables are local variables. They can be used only within the function in which they are declared, but they retain their value from one call of the function to the next.

Variable declaration and definitions

ODL variables

In ODL, you must explicitly declare all variables before you use them in code, and their declaration must precede any code statements.

You initialize variables when either when you declare them or later on in the code. However, when you declare and initialize a variable on the same line, you can only initialize it to a constant (first bullet below). When you initialize the variable on a separate line, you can use any of the three methods shown below.

- Set the variable equal to a constant.

```
var_name = 100;
```

- Set the variable equal to another variable.

```
var_name = myvar;
```

- Set the variable equal to an expression.

```
var_name = myvar + yourvar;
```

IDE variables

ODL allows you to use data objects defined with the e-Biz Impact IDE tools. Reference the data object by name just as you would a data variable that you defined within your code. Use data objects in any way that you use variables that you define within your code.

Expressions

An expression is a combination of operators and operands. The operands can be constants or variables.

Arithmetic operators

Arithmetic operators are grouped left to right. When a statement contains more than one operator, the operations are performed in their order of precedence. See “Operator order of precedence” on page 12 for information about the precedence of the arithmetic operators.

Use parenthesis ( ) to perform operations in non-precedence order. For example, for `d = a + b * c`, the variable "*d*" contains the result of "*b*" multiplied by "*c*", then added to "*a*", while for `d = (a + b) * c`, the variable "*d*" contains the result of a plus "*b*" multiplied by "*c*". In the examples shown in Table 1-3, "*a*", "*b*", and "*c*" are integer data.

Note Always put a space character between the addition or subtraction operators and the data, so that the data parser does not mistake them for sign characters rather than arithmetic operators. For example: `c = 20 - 10`; not: `c = 20-10`.

Table 1-3: Expressions, arithmetic operations

Operator	Description	Example
=	Assignment. Complex assignment/initialization.	`a = b;` Struct sample = {1,2,3,54,85};
+	Addition	`c = a + b;`
+=	Add and assign	`a += b;`
-	subtraction	`c = a - b;`
-=	Subtract and assign	`a -= b;`
*	Multiplication	`c = a * b;`
*=	Multiply and assign	`a *= b;`
/	Division	`c = a / b;`
/=	Divide and assign	`a /= b;`
%	Modulus	`c = a % b;`
%=	Modulus and assign	`a % = b;`
++	Increment: Pre-increment Post-increment	`++a/b;` (same as `a = a + 1;a/b;`) `a++/b;` (same as `a/b; a = a + 1;`)
--	Decrement: Pre-decrement Post-decrement	`--a/b;` (same as `a = a - 1; a/b;`) `a--/b;` (same as `a/b; a = a - 1;`)

Comparison operators

Comparison operators are grouped from left to right. They return zero (0) if the relation between the expressions is false, and return one (1) if it is true. When a statement contains more than one comparison operator, the operations are performed in their order of precedence. See “Operator order of precedence” on page 12 for information about the precedence of the comparison operators. Use parentheses ( ) to perform operations in non-precedence order. In the examples below, “*a*” and “*b*” are character data.

Table 1-4: Comparison operators

Operator	Description	Example
==	Equality	`a == b ...`
!=	Inequality	`a != b ...`
<	Less than	`a < b ...`
>	Greater than	`a > b ...`
<=	Less than or equal	`a <= b ...`
>=	Greater than or equal	`a >= b ...`

Logical operators

Logical operators are grouped left to right. When a statement contains more than one logical operator, the operations are performed in their order of precedence. See "Operator order of precedence" on page 12 for information about the precedence of the logical operators. Use parenthesis ( ) to perform operations in non-precedence order. In the examples below, "*a*", "*b*", "*c*", and "*d*" are integer data.

Table 1-5: Logical operators

Operator	Description	Returns	Example
!	Logical NOT	One (1) if true, and zero (0) if false.	`a = !b;` "a" is 1 when "b" is 0, and "a" is 0 when "b" is non-zero.
||	Logical OR	One (1) if either expression is non-zero, otherwise returns zero (0). The second expression is not evaluated if the value of the first expression is non-zero.	`d = a > b || c < b;` "d" is 1 when either expression's value is non-zero.
&&	Logical AND	One (1) if both expressions are non-zero, otherwise returns zero(0). The second expression is not evaluated if the value of the first expression is zero.	`if (a > b && c < d) ...` Is 1 when both expression's values are non-zero.

Bitwise operators

Bitwise operators compare each corresponding bit in the expressions and produce a result based upon that comparison. Bitwise operators apply only to integer type expressions.

In the examples below, the bitwise value of "*a*" is 1010 0110 and the bitwise value of "*b*" is 0011 1011.

Table 1-6: Bitwise operators

Operator	Description	Example
&	Bitwise AND. Sets the bit if both bits are set.	`c = a & b;` The bitwise value of "c" is 0010 0010.
&=	Bitwise AND and assign. This is the same as `a = a & b;`.	`a &= b;` The new bitwise value of "a" is 0010 0010.
|	Bitwise OR. Sets the bit if either bit is set.	`c = a | b;` The bitwise value of "c" is 1011 1111.
|=	Bitwise OR and assign. This is the same as `a = a | b;`.	`a |= b;` The new bitwise value of "a" is 1011 1111.
^	Bitwise exclusive OR. Sets the bit when only one of the bits is set.	`c = a ^ b;` The bitwise value of "c" is 1001 1101.
^=	Bitwise exclusive OR and assign. This is the same as `a = a ^ b;`.	`a ^= b;` The new bitwise value of "a" is 1001 1101.
~	1's compliment. Reverses the setting of each bit.	`c = ~b;` The bitwise value of "c" is 1100 0100.

Shift operators

Shift operators shift the bits in an expression to the left or to the right. The integer expression on the right of the operator determines how many places to the left or right to shift the bits within the expression on the left of the operator. When shifting left, all bits in the expression are shifted left one position and a zero is placed on the right. When shifting right, all bits in the expression are shifted right one position and a zero is placed on the left.

In the examples below, the bitwise value of "*a*" is 1010 0110 and "*b*" is equal to 2.

Table 1-7: Shift operators

Operator	Description	Example
<<	Left shift.	`c = a << b;` The bitwise value of "c" is 1001 1000.
<<=	Left shift and assign.	`a <<= b;` The new bitwise value of "a" is 1001 1000.
>>	Right shift.	`c = a >> b;` The bitwise value of "c" is 0010 1001.
>>=	Right shift and assign.	`a >>= b;` The new bitwise value of a is 0010 1001.

Other operators

Operator	Description	Example
.	Structure member dereference.	`mystruct.member`
->	Pointer to structure member dereference.	`mystruct->member`
,	Comma operator. In a list of expressions, causes use of the right-most expression. The comma operator also strings together operations.	`mydata = (1, 24, 94);` Results in `mydata = 94;` for `(a=0, b=5; a+b < = 50; a++, b++)`
?	Conditional operator. `data = condition ? expr1: expr2;` where `data = expr1` if condition is true, and `data = expr2` if condition is false.	`c = a > b ? 1: -1;` "c" = 1 if "a" is greater than "b", and "c" = -1 if "a" is less than "b".

Operator	Description	Example
sizeof	Computes the size, in bytes, of a datatype or variable.	`sizeof (int)` `sizeof (variable)`
/* ... */	Multiple line comment indicator. Everything between /* */ is commented out.	`/* This is a comment that can display on multiple lines. */`
//	Single line comment indicator. Everything from the // to the end of the line is commented out of the code.	`// This is a comment on one line.`

Operator order of precedence

Order of precedence	Operators
Highest	() [] -> .
	! ~ ++ -- sizeof
	* / %
	+ -
	<< <<= >> >>=
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
	?
	&= ^= |=
	= += -= *= /=
Lowest	,

Statements

ODL does not permit null statements; that is, a semicolon not preceded by anything. If you are familiar with C programming, you may be tempted to use statements that include expressions such as `for(;i<5;i++)` for loops without any commands in them, but these do not work properly in ODL. You may not always get a syntax error, but you may have other problems down the line.

Conditional statements

Conditional statements compare a condition or a value and then perform a specific statements based upon the result of that comparison. There are two types of conditional statements, the if/else statement and the switch statement.

if/else

The if/else statement has three formats:

Table 1-8: If/else statements

Format	Meaning

`if (condition)`
`{`
`do this;`
`}`
	If the condition evaluates to true, then the "`do this`" statement is executed.

`if (condition)`
`{`
`do this;`
`}`
`else`
`{`
`do that;`
`}`
	If the condition evaluates to true, then the "`do this`" statement is executed.
	
	If the condition evaluates to false, then the "`do that`" statement is executed.

Format	Meaning

`if (condition1)`
`{`
`do this;`
`}`
`else if (condition2)`
`{`
`do that;`
`}`
`else`
`{`
`get this done;`
`}`
	If condition 1 evaluates to true, then the "`do this`" statement is executed.
	
	If condition1 evaluates to false and condition2 evaluates to true, then the "`do that`" statement is executed.
	
	If neither condition is true, then the "`get this done`" statement is executed.
	
	Note This format may contain more than one "`else if ()`" statement.

switch

The switch statement performs the same kind of comparison as the third format of the if/else statement; it executes a specific statement or set of statements based upon the comparison of different simple type constants to a simple type variable.

Table 1-9: Switch statement

Format	Meaning

`switch (fixed-length type variable)`
`{`
`case constant1:`
`statement1;`
`statement2;`
`break;`
`case constant2:`
`statement3;`
`break;`
`default:`
`statement4;`
`statement5;`
`}`
	If the variable equals `constant1`, then all statements from here... to this break are executed.
	
	If the value of the variable equals `constant2`, then all statements from here... to this break are executed.
	
	If the value of the variable does not equal any of the constants, then all statements from here... to the end of the switch statement are executed.

Note A fixed-length variable is one of the following types: int, short, long, char, and the unsigned variants of those, namely, unsigned int, unsigned short, unsigned long, and unsigned char. All other types, including string and float, are considered complex datatypes in ODL, and do not work correctly as variables in the switch statement.

Loop statements

Loop statements repeatedly perform a statement or set of statements until the expression in the loop statement is false (equals zero). There are three types of loop statements, the while statement, the do statement, and the for statement. You can use the break and continue statements within any type of loop statement. “Jump statements” on page 17 for information about break and continue statements.

Note Since null statements are not permitted in ODL, each loop must contain at least one command, unlike in C, where you can use statements such as:

```
for (i = 1;i < 10;i++);
and
while (1);
```

while

The while statement tests an expression, and if the value of the expression is true (non-zero), executes the associated statements. It then tests the expression again, and continues in this pattern until the expression is false (zero). If using a while loop, loop commands might not execute. To guarantee that the loop commands execute at least once, use a do statement.

```
while (expression)
{
   do these commands;
}
```

do

The do statement executes the associated statements, then tests an expression. If the value of the expression is true (non-zero), it returns to the beginning of the loop and executes the associated statements again. It continues in this pattern until the value of the expression is false (zero).

```
do
{
   do these commands;
}
while (expression);
```

Note When using a while loop, the loop commands may not be executed at all; however, using a do ... while loop guarantees that the loop commands are executed at least once.

---

for

The for statement starts by initializing the variables used in the conditional expression, testing the conditional expression, then executing the loop's statements if the condition is true (non-zero). It then increments the variables used in the conditional expression, tests the conditional expression, then executes the loop's statements if the condition is true (non-zero). It continues in this pattern until the value of the conditional expression is false (zero).

Currently, you must include all three expressions in a for loop.

```
for (init; condition; loop) init = initial condition
{   condition = conditional expression
    do these commands; loop = loop expression
}
```

Note Because ODL, unlike C, does not allow null statements, you must have an expression for each of the three components within the parentheses. You cannot, for example, have a for loop expression statement such as:

```
for (; i<5 ; i++)
```

Also, you must list at least one command to perform within the loop (in addition to the loop expression). You cannot, for example, put this:

```
for (i=0; i<10; i++);
```

Since there is no command preceding the semicolon, that would be a null (command) statement as well.

Jump statements

break

The break statement causes an immediate exit from a loop or switch statement without executing any of the remaining statements in the statement. The next statement is then executed.

Table 1-10: Jump statements

Format	Meaning

`while (expression)`
`{`
`  do these commands;`
`  if (condition)`
`  break;`
`  do these other commands;`
`}`
`do this stuff;`
	If the break statement is executed, then the program immediately leaves the while loop, without executing the remaining statements in the loop, and continues with the "`do this stuff`" statements.

continue

The continue statement causes the execution of the next repetition of a loop statement without executing any of the remaining statements in the loop.

Table 1-11: continue statement

Format	Meaning

`do`
`{`
`  do these commands;`
`  if (condition)`
`  continue;`
`  do these other commands;`
`}`
`while (expression);`
`do this stuff;`
	If this continue statement is executed, then the program immediately goes to the "`while (expression)`" line, tests the expression, and performs the next iteration of the loop without executing the remaining statements in the loop.

goto

The goto statement causes the program to jump to the line in the code that starts with the label identified in the goto statement, skipping any statements between the goto statement and the label.

Table 1-12: goto statement

Format	Meaning

`if (condition)`
`{`
`  do this;`
`  if (another condition)`
`  goto Label1;`
`  do that;`
`}`
`Label1:`
`  do these commands;`
	If this goto statement is executed, then the program skips the "`do that`" statement and continues program execution at the "`do these commands`" statement.

return

The return statement causes the program to immediately return from a function without executing any remaining code in the function. The return statement has two forms: return; and return (value);.

Table 1-13: return statement

Format	Meaning

`func_name(arguments)`
`{`
`do this;`
`if (condition)`
`return (variable);`
`}`
	If this return statement is executed, then the "`do that`" statement is not executed. The program returns the value in the variable, and continues executing with the statement after the function call.

Arrays

The format for an array declaration is:

```
data_type variable_name[size];
```

where

Table 1-14: Arrays

Array	Description
data_type	Datatype available in ODL.
variable_name	Name of the array.
size	Number of array elements.

You access an array element based on its index number. The index of the first element of an array is always 0 (zero) and each element is the next sequential number.

You may initialize the array elements at the time you declare the array, or later in your code. If you initialize the elements when you declare the array, then you may only initialize them to constants. ODL does not perform bounds checking on arrays. Therefore, it is possible to overrun the end of an array or access a nonexistent element. You may not assign one array to another.

Examples

This statement is invalid:

```
arr2 = arr1;
```

However, you can assign individual array elements; so, this statement is valid:

```
arr2[3] = arr2[5]
```

To declare an integer array named my_array with five elements, enter:

```
int my_array[5];
```

To declare and initialize the same array, enter:

```
int my_array[5] = {1, 3, 5, 7, 11};
```

To access the third element in the array, enter:

```
my_array[2];
```

To assign a value to the first element in the array, enter:

```
my_array[0] = 24;
```

Character arrays

ODL contains a String Object class, so you may not need to use character arrays. However, if you do, character arrays in ODL have two characteristics to keep in mind—how the character array is initialized, and how it is passed to a function.

- To initialize a character array, enter:

  ```
  char my_array[5] = {"a", "b", "c", "d", "e"};
  ```

- You cannot pass a character array to a function. You can only pass the address of the array.

Structures

In ODL, structures cannot contain member functions, but can contain data members.

Structure declaration

The format for a structure declaration is:

```
struct tag
{
structure members;
}
variable names;
```

where *tag* is a user selected name, *structure members* are data members accessed by variables declared as this structure type, and *variable names* are the names of the variable declared as this type of structure.

Note You must separate multiple variable names with a comma.

Table 1-15: Structure declaration formats

Format	Description
`tag variable_name;`	Declares a variable as this type.
`variable_name.data_member =value;`	Assigns a value to a data member.

Examples

Structural declaration:

```
struct sample {int test;string name;blob date_time;}
```

Variable declaration:

```
struct sample sm1, sm2;
```

Assigning values:

```
sm1.test = 100;
sm2.test = 500;
```

Note Do not assign one structure variable to another. For example, `sm2 = sm1` is invalid. However, you may assign individual elements; thus, `sm2.test = sm1.test` is valid.

symoff datatype

symoff is an ODL-specific datatype that is a ODL object handle. Many database interface object methods use an ODL object to store retrieved table data. For example, the columnInfo() method requires access to an object in which to store a list of tables.

To supply an object's symoff to a function or method, such as columnInfo(), use the ODL bysymbol() function on the object, which involves three steps:

1 Declare a variable of type symoff, for example:

```
symoff mydest;
```

2 Declare a variable for the type of object for which you need the symoff, for example:

```
string targetstring;
```

3 Use the bysymbol() function on the object variable to return its symoff to the *symoff* variable. Continuing with the same example:

```
mydest=bysymbol(targetstring);
```

Then, use the symoff value as an argument to the method. The code should look similar to this:

```
symoff outsym;
string tbl;
string col;
string out;
int rv;
outsym=bysymbol(out);
tbl="authors";
col="city";
rv=my_cldbi.columnInfo(outsym.tbl.col);
```

Alternatively, you can skip steps 1 and 3 above, and directly pass the function itself as the symoff argument to the method. In that case, the code should look similar to this:

```
string tbl;
string col;
string out;
int rv;
tbl="authors";
col="city";
rv=my_cldbi.columnInfo(bysymbol(out).tbl.col);
```

Message functions

Message functions display a message to the user or place a message into the local *xlog* file. There are two types of message functions, the clMessageBox() statement and the erm() statement. See “clMessageBox” on page 34 and “erm” on page 40 for details.

Programming guidelines

General

In ODL, you may not use brackets without statements between them. In other words, you may not have a bracketed section that contains only comments and no coding statements.

Functions

- You may pass a maximum number of 128 arguments to an ODL function.
- Prototyping the return datatype of a function has no effect. Whatever type your function returns is what ODL uses, regardless of the function's prototyped return type.

CHAPTER 2

Bridged Functions

This chapter describes transaction production and ODL built-in functions.

Transaction production bridged functions

ODL programmers can build user-defined functions by using one of the e-Biz Impact IDE tools. These are bridged functions relating to transaction production.

charTrans

Description

Allows you to call the CharTranslate built-in function within a message AIM, rather than including it in a production object. If source and targets are not properly defined, a error message states that it could not find the target.

Syntax

charTrans (*&data*,'\000', “ASCII”, “T-11 EBCDIC”);

Parameter	Description
&data	Defines the incoming data to be translated. Can be string or blob datatype.
‘\000’	Sets the mode variable. Set to `\000` to fail if a character cannot be translated. Set to `\002` to use the default value if the character cannot be translated.
ASCII	String that defines the source from which any character set is translated; for example, the ASCII character set.

Parameter	Description
T-11 EBCDIC	String that defines the target to which any character set is translated; for example, the T-11 EBCDIC character set.

Return value

Integer.

Returns 1 for success, and 0 for failure.

setTblDir

Description

Specifies a new lookup location for translation table files (*.*tbl*) by allowing you to programmatically specify their directory location. Use this function within AIMs that use the produce() method to perform transaction production. It affects the table lookup location for the AIM from which it is called.

In the initial time that an AIM needs to access a *.tbl* file, the table's contents are into memory where they remain for further use. Changing the table lookup location does not delete from memory any tables previously loaded tables. Therefore, if there is a *.tbl* file in the new table lookup location that has the same name as a *.tbl* file that was in the previous table lookup location, the new table's contents are not loaded into memory and the previous table's contents are still used.

If dynamic table loading is enabled, tables are reloaded if they have a more recent date and time stamp than the previously loaded tables. Use this function prior to first accessing tables.

Syntax

setTblDir(string *newdir*);

setTblDir("C:\the absolute path to the .tbl files");

Parameter	Description
newdir	Directory location of the *.*tbl* files you want to use. This string may contain a relative path or an absolute path. On Windows, an absolute path may include a drive with the directory path.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Usage

```
string newdir;
```

```
newdir = "C:/Sybase/ImpactServer-5_4/tables";
setTblDir(newdir);
```

ODL bridged functions

Bridged functions are supplied with the e-Biz Impact runtime to make it easier to implement your application.

chgCwd

Description

Changes the current working directory for an application, while setting the old working directory into a supplied parameter. This function can be used later to change the working directory back to its previous location.

Syntax

int chgCwd(string *newPath*, string **oldPath*)

Parameter	Description
newPath	Directory of the new path.
oldPath	Pointer to the string that chgCwd places in the current working directory.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
string new_path;
string old_path;
int rv;
new_path = "/home/impact/app1";
rv = chgCwd(new_path, &old_path);
if (rv == 0)
{
    erm("Error changing working directory to [%s]!!!",
          new_path);
}
```

clAlert

Description

Generates a user-defined alert. Depending on how the cluster is configured, this could translate to a SNMP trap being generated, an Open Transport alert message being sent, both, or neither.

Syntax

```
clAlert (int id,
        string reasonCode1,
        string reasonCode2,
        string reasonCode3,
        string reasonCode4,
        string reasonText);
```

where the *id* is user-defined. The value that is seen by the application receiving alert messages is [id+1000000].

Return value

None.

See also

For more information on configuring and using alerts, see *Monitoring e-Biz Impact*.

clCmd

Description

Executes the command line argument by passing an argument to a command processor, enabling such actions as wild card expansion and redirection for the function to act as if it was executed from the command line. This is similar to the UNIX system(3) command.

By contrast, clSystem() executes Cmdline without first passing it as an argument to a command processor.

Note On Windows, the command processor is *cmd.exe*, and on UNIX the command processor is */usr/bin/sh*.

Syntax

```
clCmd (string Cmdline);
```

Parameter	Description
Cmdline	String that defines the command line instruction to pass to Windows. The instruction must be in the format expected by the Windows command line.

Return value

Integer.

Returns the 8-bit exit status of the command line program for success.

On UNIX, returns the negative value of the UNIX error number (as documented in the UNIX *errno.h* file), and on Windows, returns the 8-bit exit status of the command line program for failures.

Examples

```
int rv;
rv = clCmd("copy *.doc a:");
```

clDisable

Description

Disables the current application.

Warning! Use this function with caution. It requires operator intervention to re-enable an application once is has been disabled. Any new DFC commands for the disabled application automatically re-enable it.

Syntax

int clDisable()

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
// something happens in the application
// to determine it should shut down.
rv = clDisable();
```

clExec

Description

Executes a child process, similar to clCmd and clSystem. However, for both UNIX and Windows, the behavior of clExec is to run without a command processor (*/usr/bin/sh* or *cmd.exe*). For each platform, the child process is executed directly.

Note If you need functionality, such as wildcard expansion or redirection of input or output, use *clCmd* because the command line must not include anything a command processor provides. If it does, the function may not behave as designed.

Syntax

int clExec(string *command*)

Parameter	Description
command	String of the full command line for the child process you want to start. Do not include any command processor features (wildcard expansion, redirection, and so on). If you need command processor features, use clCmd instead.

Return value

Integer.

Returns the 8-bit exit status of the command line program for success.

On UNIX, returns the negative value of the UNIX error number (as documented in the UNIX *errno.h* file, and on Windows, returns the 8-bit exit status of the command line program for failures.

Examples

```
string exec_command;
int rv;
exec_command = "/usr/local/stuff/myprog arg1 arg2";
rv = clExec(exec_command);
```

clGetArgc

Description

Determines how many arguments are used on the command line that launches the application. Similar to the C-based `argc` parameter for the C main function.

Syntax

int clGetArgc();

Return value

Integer.

Returns the number of command line arguments.

Examples

```
int max;
max = clGetArgc();
```

clGetArgv

Description

Finds specific arguments that are on the command line that launches the application. Similar to the "`char *argv[]`" that is passed to the C main function.

Syntax

int clGetArgv(int *index*, string **argument*));

Parameter	Description
index	Argument number you want to find, zero index based.
argument	String to which you want to put the argument.

Return values

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
string myArg;
int rv;
rv = clGetArgv(3, &myArg)
```

```
string argv;
clGetArgv(0,&argv)://argv is the name of the
                   //application running the ODL
erm("argv[0]=[%s]",argv);
```

clGetConfig

Description

Retrieves a value from the application configured custom keys as generated by the Configurator. Custom keys are configured on the Advanced tab of the ODL Properties.

Syntax

int clGetConfig(string *tag*, string **value*)

Parameter	Description
tag	String of keys (case sensitive) that are specified in the configurator. This will be "Name" of the key set, plus the individual key, separated by a period (.). For example, if you set the Name to "Flag" and the Key to "Seconds", you would specify a tag of "Flag.Seconds".
value	Pointer to the string to receive the value corresponding to the tag parameter.

Return value

Integer.

Returns 1 for success, and 0 for failure (such as a null tag string to look up, or being unable to find the tag string passed in).

Examples

```
int rv;
string secondsTag;
```

```
string num_seconds;
secondsTag = "Flag.seconds";
rv = clGetConfig(secondsTag, &num_seconds);
if (rv == 1)
{
    erm("Seconds is [%s]", num_seconds);
    // do anything else you want with the data
}
else
{
    erm("Unable to retrieve seconds!!!);
}
```

See also

For more information, see the *e-Biz Impact Configurator Guide*.

clGetDfcError

Description

Gets DFC error text when a DFC error is encountered. Typically, you then erm the text into the *xlog*, so it can be traced.

Syntax

clGetDfcError(&*str*)

Parameter	Description
str	String in which you want to place the DFC error text.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
string str;
//issue DFC command here
//and get an error return value (-1)
clGetDfcError(&str);
erm("DFC error string=[%s]", str);
```

clGetDfcErrno

Description

Gets a DFC error value.

Syntax

clGetDfcErrno()

Return value

Integer.

Returns a DFC error value.

Examples

```
int my_dfcerrno;
//issue DFC command here
//and get an error return value (-1)
my_dfcerrno = clGetDfcErrno();
erm("DFC error number=[%d]", my_dfcerrno);
```

clGetEnv

Description

Looks up environment variables for the current process, and bridges the standard API getenv.

Syntax

int clGetEnv(string *variable*, string **value*)

Parameter	Description
variable	Environment variable to look up. This should not be a null string.
value	Pointer to the location to put the value the environment variable is set to. This is set to a null string if the variable cannot be found.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
int rv;string envar_name;
string envar_value;
envar_name = "PATH";
rv = clGetEnv(envar_name, &envar_value);
if (rv == 1)
{
    erm("PATH envar is [%s]", envar_value);
}
else
{
    erm("Unable to get PATH envar!!!");
}
```

clInstance

Description

Returns the instance number of the current server application. To add this functionality to an application, insert the following `#dll` statement at the top of the project file:

```
#dll lib[X] "rtesrv.dll"
```

where *X* is a unique number.

This is useful when you run multiple instances of a server application.

Syntax

clInstance();

Return value

Integer.

Returns > 0 (the instance number of the application instance running the application) for success, and 0 for failure.

Examples

```
int rv;
string login;
rv = clInstance();
if (rv ==1)
    login="Serv1";
if (rv==2)
    login="Serv2";
```

clIsServer

Description

Determines whether ODL code is running under the client or the server.

Syntax

int clIsServer();

Return value

Integer.

Returns 1 if running under the server, and 0 if running under the client.

clMessageBox

Description

Brings up a message box with an OK button on the client. On the server, it writes the message to the *xlog* file, similar to the erm function.

Syntax

clMessageBox (string *title*, string *message*);

Parameter	Description
title	Title for the dialog box when running on the client. Placed first in the xlog when run on the server. Can also be a literal string, such as `Error`.
message	String to display in the text of the dialog box. Placed second in the xlog when run on the server. This can also be a literal string, such as `Unable to parse data`.

Return value None.

Examples

```
clMessageBox ("Quit Message", "Program terminating");
```

```
string title;
string message;
string pat_name;
pat_name = getPatientName();
title = "Error!";
message = "Unable to parse data for [" + pat_name + "]";
clMessageBox(title, message)
```

clQuit

Description Used to immediately close an application.

Syntax clQuit();

Return value None.

Examples

```
clinit()
{
    bool bGoodConfig;
    bGoodConfig = parseMyConfig();
    if (!bGoodConfig)
    {  // if config is bad, exit
       clQuit();
    }
    ...
}
```

clRelease

Description Reverses the effect of clSuspend() by immediately returning program control to the control flow/function object that initially called clSuspend().

You must call clRelease() from within a control flow/function object other than the one that called clSuspend().

Syntax clRelease();

Return value None.

clSetAimStatus

Description

ODL supports publishing an AIM's status with text to the Global Console. This information displays in the Global Console as custom health code and custom health text. By default, an endpoint failure does not change the status of the destination to which the AIM is being sent. From an e-Biz Impact point of view, the destination is still alive and is capable of accepting transactions from the SFM, so from the Global Console the endpoint displays as healthy. ODL publishing allows the application developer to set the AIM's health to "bad" if this condition occurs.

Syntax

clSetAimStatus(int *code*, string *text*);

Parameter	Description
code	Status value where 0 is for Information (good), 1 is for Warning (problematic), and 2 is for Error (bad).
text	Text description for the current status. Use this to indicate why the current condition is occurring. For example, you can use this to differentiate between two places in your AIM that can result in an Error status.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

Good status:

```
clinit()
{
// rest of your initialization logic here
 clSetAimStatus(0, "Initialization Complete.");
}
Error Status:
close_flow()
{
 clSetAimStatus(2,"Lostdndpoint Connection!!!");
 //the rest of your close flow logic
}
```

clSetDfcTimeout

Description

Overrides the default DFC timeout to the value specified. This is useful if you know that the contents of a particular call takes an abnormally long time to process.

Syntax

int clSetDfcTimeout(int *timeout*);

Parameter	Description
timeout	Timeout value in seconds and greater than 0.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
int new_timeout = 120;
clSetDfcTimeout(new_timeout);
// issue DFC command here


clSetDfcTimeout(120)
// issue DFC command here
```

clSleep

Description

Causes the application to suspend execution for a specified amount of time.

int clSleep(int *sleepTime*);

Parameter	Description
sleepTime	Amount of time in seconds to sleep.

Return value

Integer.

Returns 1 for success, and any other value for failure.

Examples

```
int delay = 5;
...
clSleep(delay);
```

clSuspend

Description

Suspends processing of the current DFC command. A message AIM cannot process data through its clcomm object while servicing a DFC command. clSuspend() causes a control flow/function object that is servicing a DFC command to suspend processing and pass program flow control to the message AIM's protocol object. This allows the application to send and receive data through the clcomm object and process that data through message frames and their associated control flow objects before sending a return on the DFC command. To resume processing the DFC command, call clRelease() from within a control flow other than the one that called clSuspend().

clSuspend() affects only the processing of the associated DFC command; therefore, other control flow objects can still send and receive DFC calls and replies.

Syntax

clSuspend();

Return value

None.

Examples

```
servproc(... < insert args here > ... )
{
    ...
    comm1.send(data);
    // sent data.  time to frame for an ACK or NAK
    clSuspend();
    // control returns here
    if (bHaveAck)
    {
       rv = 1;
    }
    else
    {
       rv = -2;// indicate an error to SFM
    }
    ...
    return rv;
}
```

clSystem

Description

Executes the command line argument by directly running the command. On UNIX, this function behaves similar to system(3). On Windows, this function works with CreateProcess(). To provide wild card expansion or redirection to a file, use clCmd().

Syntax

clSystem(string *Cmdline*);

Parameter	Description
Cmdline	String that allows you to enter the command you want to execute.

Return value

Integer.

Returns the 8-bit exit status of the command line program for success.

On UNIX, returns the negative value of the UNIX error number (as documented in the UNIX *errno.h* file), and on Windows, returns the 8-bit exit status of the command line program for failures.

clTimeToStop

Description

Determines whether an acquire-mode AIM should shut down. Since an acquire-mode AIM is granted a dedicated execution thread, it needs to periodically verify whether it should shut down.

Syntax

int clTimeToStop()

Return value

Integer.

Returns 1 for shut down, and 0 if continue to run.

Examples

```
// in clAcquire
...
while (clTimeToStop() != 1)
{
    // do something to gather data in
    // a non-blocking fashion
    // when you have data, do something
    // with it, such as making a route_*
    // call to SFM.
    // Then, go back up to check if
    // it should shut down or not
}
```

erm

Description

Places a message into the local *xlog* file. The maximum formatted output allowed is 1024 bytes.

Syntax

erm(*message*,*arguments*);

Parameter	Description
message	Prints a style message that may include both plain text and optional conversion specifications.
arguments	Optional arguments that are processed according to the conversion specifications in the “message” parameter.

Return value

None.

Examples

```
erm("Pat_rec: Record not found for patient -
%s",pat_name);
```

getCwd

Description

Retrieves the application’s current working directory.

Syntax

int getCwd(string **workingDir*);

Parameter	Description
workingDir	String to which you want to put the working directory.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
string szHome
getCwd(&szHome);
erm(“my home [%s]”, szHome);
```

GetProcessArgc

Description

Retrieves the number of process command line arguments.

Syntax

int GetProcessArgc();

Return value

Integer.

Returns the number of arguments on the command line.

Examples

```
int max;
max = GetProcessArgc()
```

See also

"GetProcessArgv" on page 41.

GetProcessArgv

Description

Finds specific arguments that are on process' command line.

Syntax

GetProcessArgv(int *index*, string **argument*));

Parameter	Description
index	Argument number you want to find, zero index based.
argument	String to which you want to put the argument.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
string myArg;
int rv;
rv = GetProcessArgv(3, &myArg)
```

See also

"GetProcessArgc" on page 40.

getTime

Description

Gets the current time in the form of a text string. The format is identical to the output of the standard API ctime.

Syntax

int getTime(string **timeStr*, int *flag*);

Parameter	Description
timeStr	Pointer to a string to received the formatted time string output.
flag	Reserved for future use. Must be 0.

Return value

Integer.

Returns 1 for success, and 0 for failure.

Examples

```
string my_time;
```

```
int rv;
rv = getTime(&my_time, 0);
if (rv == 0)
{
    erm("Unable to get time!!!");
}
else
{
    erm("Current time is [%s]", my_time);
}
```

CHAPTER 3

Message Objects and Methods

This chapter describes ODL message objects and their associated methods.

Frame objects

Frame (clframe) objects are used within the protocol object to extract a well-defined packet format from the data received by the communication object.

reset()

Description

Used in dynamic framing and allows string and byte length variables that delimit a message to be modified while messages are coming in, without making changes in the MSG-IDE's Message Frame window.

Frame objects update the value of one or more variables that have been inserted into a message frame.

Note Before using the reset() method, the variables must have been declared in the data section of MSG-IDE's Define window, inserted into the message frame in MSG-IDE's Message Frame window, and initialized in clinit().

For more information on defining variables and inserting them into message frames, see the *e-Biz Impact MSG-IDE Guide*.

Syntax

```
reset();
```

Return value None.

Example

```
........
string end_of_message;
........

clinit()
{
       end_of_message = "ETX"; // initial value
    ........

}
// Frame object is called "msg1_frame" and defined as:
// All Data Until..
// V_string end_of_message
// .. end All Data Until

reset_frame_function()
{
              if( condition )
              {
                      end_of_message = "EOF"; // reset the string variable used
                    // in msg1_frame;
                       msg1_frame.reset(); // msg1_frame will now look for "EOF"
                    //to find the end of the incoming message
             }

........
}
........
```

Communication objects

Communication objects (clcomm) are used within the protocol object to communicate over a TCP/IP connection.

die()

Description Shuts down a communication connection. If the connection is in listen up mode, it closes only the current conversation and continues to listen. Any pending connections remain pending.

Syntax

die()

Return value

Integer.

Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`pcomm->die();`	`mycomm.die();`

kill()

Description

Shuts down a communication connection. When in listen up mode, the current connection is closed. When you run this method, closeflow is not automatically called. To call closeflow functions, you must do it programatically after you run the method.

Syntax

kill();

Return value

Integer.

Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`pcomm->kill();`	`my_comm.kill();`

restart()

Description

When clcomm is in connect mode, attempts to reconnect to the endpoint application. When clcomm is in single-instance listen mode, listens again for the endpoint application after a connection has been lost.

Syntax

restart();

Return value

Integer.

Returns any positive value for success, which indicates that restart() launched the clcomm attempt to reconnect to or listen for the endpoint. However, success does not guarantee a successful connection.

Returns 0 for failure, which indicates that restart() failed to launch the attempt to connect to or listen for the endpoint.

Usage

In a function	In a control flow
`mycomm.restart();` where *mycomm* is the name of the communications object.	1. Click Object. 2. Select the communications object from the drop-down list. 3. Select restart from the Method drop-down list.

send()

Description

Sends data through the clcomm object.

Syntax

send({string | blob} *data*);

Parameter	Description
data	Pointer to the string or blob object that contains the data you want to send.

Return value

Integer.

Returns any positive value for success, which indicates the length of the data sent to clcomm. However, a positive return value does not guarantee that the data was sent successfully through the clcomm object to the endpoint.

Returns a void for failure.

Usage

In a function	In a control flow

If not using an existing data object, declare a string or blob object.
 	
`string mydata;`
`mycomm.send(&mydata);`
 	
where *mycomm* is the name of the communications object.
	1. Click object.
	2. Select the communications object from the drop-down list.
	3. Select send from the method drop-down list.
	4. In the Command Arguments field, type the name of the data object containing the data to send.
	5. In the Command Argument Declaration field, type `string*data`, where "data" is the name of the data object. If using a blob data object, type `blob *data`.

Protocol objects

The protocol object (clproto) contains message frame and communication objects and manages the interaction between them.

clear()

Description: Clears the accumulated, but not framed, data.

Syntax:

```
int clear();
```

Return value: Integer.

Returns 1 for success, and 0 for failure.

Example:

```
proto1.clear()
```

halt()

Description: Suspends framing until new data comes in, or until the process or resume method is called.

Syntax:

```
int halt();
```

Return value: Integer.

Returns 1 for success, and 0 for failure.

Usage

In a Function

```
proto1.halt();
```

process()

Description: Sends data to the protocol object. Used in a message AIM to send a string or blob of data to the protocol object and append it to any unprocessed data currently being held by the protocol object. The protocol object then sends the data through the associated message frame and control flow objects for processing. Once the message frames no longer bid on data in the working blob area, program control returns to the function and control flow that called the process() method.

Syntax:

```
int process(string *data);
int process(blob *data);
```

Parameter	Description
data	A pointer to the string or blob object containing the data to send.

Return value

Integer.

Returns a positive integer for success, and 0 for failure.

Usage

In a function	In a control flow

If not using an existing data object, declare a string or blob object, and initialize it with the data.
`string mydata;`
`mydata = "some stuff";`
`proto1.process (&mydata);`
where *proto1* is the name of the protocol object.
	1. Click Object.
	2. Select the protocol object from the drop-down list.
	3. Select process from the Method drop-down list.
	4. In the Command Arguments field, type the address of the data object containing the data to send; for example, `&mydata`.
	5. In the Command Argument declaration field, type `string *data` where "data" is the name of the data object. If using a blob data object, then type `blob *data`.

start()

Description

Starts the protocol object. You must start a protocol object to begin the connection and data handling in an ODL message AIM. This is typically done in the clinit() function.

Syntax

int start();

Return value

Integer.

Returns 1 for success, and 0 for failure.

Example

```
clinit()
{
//rest of your init
my_proto.start();
//rest of your init
}
```

CHAPTER 4

Production Objects and Methods

This chapter describes the ODL production objects and their associated methods.

Production objects

Production objects (clTran) parse data, qualify parsed data, and produce an output transaction. Use TRAN-IDE to define production objects, which are used in SFMs or routers.

debug()

Description

Used as an error function in transaction production to debug a production object. It dumps information to the *xlog* file about the field, rule, and component objects involved in a runtime processing error.

Syntax

```
debug();
```

Return value

None.

Examples

```
my_prod.debug();
```

where my_prod is the name of the production object.

getAlterrtext()

Description

Used in transaction production to retrieve the values set by the setErrNum() and setErrTxt() methods and place them into an integer and string data variable. You must use both setErrNum() and setErrTxt() for the getAlterrtext() method to function correctly.

Syntax

getAlterrtext(string **msg*);

Parameter	Description
msg	Pointer to the string object to hold the alternate error message value set by the setErrText() method.

Return value

Integer.

Returns the alternate error number value set by the setErrNum() method.

Examples

```
int rv;
string my_msg;
rv = my_prod.getAlterrtext(my_msg);
erm( "Alt. error value - %d, alt. error text : [%s]",
rv , my_msg);
```

where *my_prod* is the name of the production object.

geterrtext()

Description

Retrieves the error text generated by the production object and places the text into a string variable. The error is set when a failure such as error during parsing of data occurs.

Syntax

geterrtext(string **ErrTxt*);

Parameter	Description
ErrTxt	Pointer to a string variable to hold the error text.

Return value

Integer.

Returns the most recent error number set by the production object.

Examples

```
string my_msg;
my_prod.geterrtext(&my_msg);
erm("Production Error : [ %s]", my_msg );
```

where *my_prod* is the name of the production object.

produce()

Description

Parses the input data into production object field objects, sets associated datalink objects, runs the production and field object qualification objects, runs production object rules, components, filters, and places the resulting output data into the defined output value. This method is used in AIM development that includes a transaction production file containing the production object.

Syntax

my_prod.produce(blob *_input_, blob *_output_);

where my_prod is the name of the production object created using TRAN-IDE.

Parameter	Description
input	Pointer to the blob object that contains the input data.
output	Pointer to the blob object that holds the output data.

Return value

Integer.

Returns > 0 for success, and < or = 0 for failure.

Examples

```
blob In_data;
blob Out_data;
In_data = "a bunch of data";
my_prod.produce(&In_data, &Out_data);
// Out_data blob now contains the output of my_prod
// production object
```

qual()

Description

Parses input data into production object field objects, sets associated datalink objects, and runs the associated qualification objects. This method is used in AIM development that includes a transaction production file containing the production object.

Syntax

qual(blob *_input_);

Parameter	Description
input	Pointer to the blob object that contains the input data.

Return value

Integer.

Returns > 0 for success, and < or = 0 for failure.

Examples

```
blob In_data;
```

```
In_data = "a bunch of data that fails to qualify";
my_prod.qual(&In_data);
```

where my_prod is the name of the production object.

release()

Description

Clears out memory allocated by production objects during transaction production.

Syntax

release();

Return value

Integer.

Always returns 1.

Examples

```
my_prod.release();
```

where *my_prod* is the name of the production object.

setErrNum()

Description

Sets a user-defined, alternate error number in the production object.

Syntax

setErrNum(int *num*);

Parameter	Description
num	Integer that serves as the alternate error number.

Return value

Integer.

Returns the value in the parameter num.

Examples

```
int myerr;
my_prod.setErrNum(myerr);
my_other_prod.setErrNum(555);
```

where *my_prod* is a production object.

Usage

This method sets a user-defined alternate error number. You may use it in a production object's custom filter, error, generic, and qualification functions. This allows you to use a unique error number for each function so that you can immediately determine within which function the processing error occurred. This method does not replace the error number generated by the production object.

When a processing failure occurs, the alternate error number displayed is that of the last function that called setErrNum(). For example, a custom filter function calls setErrNum() to set the error code upon failure, then the error function of the production object is called, which also uses this method. The alternate error code contains the value used by the error function, not the custom filter. Also, when you use this method in a function, it sets the alternate error number regardless of whether or not the function encounters a processing error. Therefore, if the failed function does not call setErrNum(), but a previously executed function did, the alternate error number generated by the production object does not reflect the function where the processing failure actually occurred.

setErrTxt()

Description

Sets user-defined, alternate error text for the production object.

Syntax

setErrTxt(string *msg*);

setErrTxt("the alternate error message");

Parameter	Description
msg	The alternate error message to append to the error text. Can be a literal value or string object.

Return value

Integer

Always returns 1.

Examples

```
string my_err_msg;
my_err_msg = "Qual33 bit the dust.";
my_prod.setErrTxt(my_err_msg); //my_prod is a Prod Obj.
my_other_prod.setErrTxt("Filt42 failed.");
```

Usage

The seterrtxt() method sets a user-defined error text in the production object. Use this method in production object custom filter, error, generic, or qualification functions to generate a unique error text from each function. This allows you to immediately determine within which function the processing error occurred.

This method does not replace the error text generated by the production object.

When a processing failure occurs, the alternate error text displayed is that of the last function that called setErrTxt(). For example, a custom filter function that calls this method fails, then the error function of the production object that is executed next also uses this method. The error message generated by the production object contains the alternate error text set by the error function, not that set by the custom filter.

Also, when you use this method in a function, it sets the alternate error text regardless of whether or not the function encounters a processing error. Therefore, if the function that fails does not call setErrText(), but a previously executed function did, the alternate error text generated by the Production object does not reflect the function where the processing failure actually occurred.

setIterMax()

Description

Sets the number of times an iterative production rule executes. This method is used in AIM development that includes a transaction production file containing the production object that has an iterative rule.

Syntax

setIterMax(int *times*);

Parameter	Description
times	The integer value of the number of iterations. Must be a positive integer or the method fails.

Return value

Integer.

Returns the number of iterations for success, and -1 for failure.

Examples

```
int no_times;
no_times = 10;
my_rule.setIterMax(no_times);
```

Qualification objects

Qualification (clQual) objects are used by a production object to check if a certain condition is true or false, and qualify field objects or datalinks.

Only use qualification object methods within the production object's functions that you build using TRAN-IDE.

qual()

Description

Executes the production objects's qualification from within a function contained by the production object. The method allows the construction of complex qualification expressions that check multiple conditions.

Syntax

int qual();

Return value

Integer.

Returns 1 if qualified, and 0 if failed.

Examples

```
int qual_rv;
int rv;
qual_rv = Qual_ZipCode.qual(); // where Qual_ZipCode is
// a Qualification object built using TRAN-IDE
if( qual_rv )
{
    erm(" Zip Code is within the county");
    rv = 1;
}
else
{
    erm(" Zip Code is outside the county");
    rv = 0;
}
return rv;
```

```
if( Qual_City.qual() && Qual_State.qual() )
    return 1;
else
    return 0;
// where Qual_City and Qual_State are Qualification
// objects defined using TRAN-IDE
```

Usage

Pointer to object	Object
clQual *pQual;	int rv;
int rv;	...
...	rv = myQual.qual();
rv = pQual->qual();	if (rv > 0)
if (rv > 0)	{// qualified
{// qualified	...
...	}
}	else
else	{// did not qualify
{// did not qualify	...
...	

CHAPTER 5 General Objects and Methods

This chapter describes the general ODL objects and their associated methods.

Binary large objects

Binary large objects (blobs) objects are a collection of bytes containing arbitrary values that include null characters terminated by a length value. The maximum size of a blob equals the maximum size of an integer on your computer system. For most computer systems, the maximum size is 4 GB. To define a blob, use this format:

```
blob obj_name;
```

Using the += operator to add a string to a blob:

```
blob my_blob;
string str;
my_blob = "something";
str = " stringy";
my_blob += str;
//my_blob now contains "something stringy"
```

Using the += operator to add a blob to another blob:

```
blob my_blob;
blob blob2;
my_blob = "something";
```

```
blob2 = " blob-like";
my_blob += blob2;
//my_blob now contains "something blob-like"
```

Using the add() method to add a string to a blob:

```
blob my_blob;
string str;
my_blob = "something";
str = " methodical";
my_blob.add(str);
// my_blob now contains "something methodical"
//my_blob.add(&str) would serve the same purpose
```

Note The add() method does not support using the blob itself as an argument. Instead, you must use the blob address.

Using the add() method to add a blob to a blob:

```
blob my_blob;
blob blob2;
my_blob = "something";
blob 2 = " methodical";
blob2
my_blob.add(&blob2);
```

You can also access individual bytes of blob data by indexing into the blob object. blob objects have a 0-based index. If you subscript past the current size of the blob, ODL extends the blob with null bytes to fit the required size. You can reference the blob directly as shown in the following example, or you can use a pointer to the blob.

```
blob myblob;
char ch;
//variable to hold the character read
myblob = "abcdefg";
ch = my_blob[5];
//ch now contains "f"
```

Note If you use a pointer to a blob, use the format shown in the "Pointer to object" column of the usage tables. Otherwise, use the formats shown in the "Object" column of the usage tables.

---

add()

Description

Appends data in an item to a blob.

Syntax

add(string *item*);

add(string *&item*);

add(blob *&item*);

Parameter	Description
item or *&item*	Data (or the address of the data) to append to the current blob. Can be the address of a blob or a string, the string itself, or a literal value.

Return value

If successful, returns a blob containing the existing and appended data. If failed, returns a blob containing the existing data before the blob() method was called.

Examples

```
blob my_blob; //current blob object
string new_stuff; //string to hold data to append
my_blob = "a bunch of data";
new_stuff = "append this stuff";
my_blob.add(new_stuff); //appends data in new_stuff to
my_blob
```

Usage

Pointer to object	Object
`pb->add("this is new data");`	`myblob.add("this is new data");`
`pb->add(&patient_name);`	`myblob.add(patient_name);`
`blob more_blob;`	`blob more_blob;`
`pb->add(&more_blob);`	`myblob.add(&more_blob);`

Note The add() method does not support using the blob itself as an argument. You must use the blob address.

clr()

Description

Clears a blob to zero length. This method does not release memory.

Syntax

clr();

Return value

Always returns 0.

Usage

Pointer to object	Object
`pb->clr();`	`myblob.clr();`

copy()

Description

Copies part of a blob, starting at location offset for a user-defined length, and places the copy into another blob or string. The first character in a blob is always at offset 0. If the offset plus the length is greater than the total length of the blob, or if the length is -1, the copy() method copies data from the offset position to the end of the blob.

Syntax

copy(int *offset*, int *length*, {blob | string} *destination*);

Parameter	Description
offset	Starting location to copy data. The first character in a blob is always at offset 0.
length	Number of characters to copy.
destination	blob or string in which to place the data.

Return value

Integer. Returns the specified length of data copied, if successful, and 0, if failed.

Examples

```
blob other_blob;
//Blob object to place copied data into
pb->copy (5,20,&other_blob);


//copies 20 char of data, starting at an offset of 5
blob my_blob;
//places it into new_stuff
string new_stuff;
//blob object to copy data from
my_blob = "abcdefghijk";
//string object to copy data to
my_blob.copy(4,-1,&new_stuff);
//copy data from my_blob from "e" to end of blob, and
//places it into new_stuff. my_blob is unchanged and
//new_stuff now contains "efghijk".
```

Usage

Pointer to object	Object
`pb->copy(0, 4, &some_blob);`	`myblob.copy(0, 4,&some_blob_ptr);`
`pb->copy(4, 4, &namestring);`	`myblob.copy(4, 4, &string_ptr);`

cut()

Description

Removes blob data, starting at location offset for a specified length, and places it into the blob or string destination, overwriting the contents of original destination blob. The first character in a blob is always at offset 0. If the offset plus the length is greater than the total length of the blob, or if the length is -1, the cut() method removes data from the offset position to the end of the blob.

Syntax

cut(int *offset*, int *length*, {blob | string} *destination*);

Parameter	Description
offset	Starting location to remove data. The first character in a blob is always at offset 0.
length	Amount (number of characters) of data to remove.
destination	blob or string in which to place the data.

Return value

Integer. Returns the length of data cut, if successful, and 0, if failed.

Examples

```
blob other_blob;
pb->cut(5, 20, &other_blob);
blob my_blob;
string new_stuff;
my_blob = "abcdefghijk";
new_stuff = "newblob";
my_blob.cut(4, -1, &new_stuff);
```

Usage

Pointer to object	Object
`pb->cut(5, 10, &my_other_blob);`	`myblob.cut(5, 10, &my_other_blob);`
`pb->cut(0, 4, &firstpart);`	`myblob.cut(0, 4, &firstpart);`

debug()

Description

Used in AIM development to dump blob contents into the *xlog* file for debugging purposes. This file resides in the current working directory, usually the directory from which the user runs the program.

Warning! Use this method with caution. Because dumping the contents of a blob is time consuming, this method is not recommended for general use.

Syntax

debug(string *comment*);

Parameter	Description
comment	A string containing a message to be placed into the *xlog* file before the information in the blob. Use to identify the blob contents.

Return value

Integer. Returns the blob size.

Usage

Pointer to object	Object
`string header;`	`myblob.debug();`
`header="This is blob A:";`	
`pb->debug(header);`	

email()

Description

Uses the e-mail system defined in the Configurator to send the blob to the person or location identified by the destination parameter. You can use this method only within an e-Biz Impact instance.

Syntax

email(string *destination*);

Parameter	Description
destination	A string containing the person or location to which the email is sent.

Return value

Always returns 1.

Usage

Pointer to object	Object
`string mail_to;`	`myblob.email(mail_to);`
`mail_to = "johnb@rd";`	
`pb->email(mail_to);`	

extend()

Description

Increases the blob to the size of the value in length and initializes the new space with the character specified in the pad parameter. The new space is always placed at the end of the existing blob data. The value in length must be greater than the current size of the blob or the method fails. You can use any standard notation in the *pad* parameter, including printable characters (“a”), hexadecimal notation (“0X40”), or octal notation (“\011”).

Note The extend() method does not place a null character (\000) at the end of the blob. If you intend to read the blob into a string object, you must add a null to the end of the blob. For example, after running the extend() method on `my_blob`, add `my_blob +='\000';` to add the null character.

Syntax

extend(int *length*, char *pad*);

Parameter	Description
length	Indicates how much to increase blob size.
pad	Character to use to initialize the new space. Enclose this character in single quotes.

Return value

Integer. Returns the new blob size if successful, and 0 if failed.

Examples

```
blob my_blob;
my_blob = “abcdefg”;
my_blob.extend(15,’*’);
```

Usage

Pointer to object	Object
`int newsize;`	`myblob.extend(newsize, ‘0xff’);`
`newsize = 12;`	
`pb->extend(newsize,’’);`	

grow()

Description

Increases the blob to the size of the value in *length*. The new space is always placed at the end of the existing blob data. The *length* value must be greater than the current size of the blob or the method fails. The new space is initialized to null.

Syntax

grow(int *length*);

Parameter	Description
length	How much to increase blob size.

Return value

Integer. Returns the new size of the blob, if successful, and -1, if failed.

Example

```
myblob.grow(500);
```

Usage

Pointer to object	Object
`int newsize;`	`myblob.grow(newsize);`
`newsize = 30;`	
`pb->grow(newsize);`	

jam()

Description

Inserts space into a blob, starting at *offset* for a specified *length*, and initializes the new space with the character specified in the *pad* parameter. The first character in a blob is always at offset 0. You can use any standard notation in the *pad* parameter, including printable characters (“a”), hexadecimal notation (“0X40”), or octal notation (“\011”).

Syntax

jam(int *offset*, int *length*, char *pad*);

Parameter	Description
offset	Location to start inserting. The first character in a blob is always at offset 0.
length	Number of characters to insert.
pad	Character to use to initialize the new space. Enclose the character in single quotes.

Return value

Integer. Returns the amount of space inserted, if successful, and 0, if failed.

Examples

```
blob myblob;
myblob = "abcdefg";
myblob.jam(4, 5, '*');
```

Usage

Pointer to object	Object
`pb->jam(20, 10, ' ');`	`myblob.jam(20, 10, ' ');`

load()

Description

Reads the file identified in the parameter and copies its contents into the blob, appending a line-feed character (“0x0a”). The file can contain any type of data. The load() method reads the data until it encounters an end-of-file condition.

Syntax

load(string *filename*);

Parameter	Description
filename	String object that contains the name of the file to read. Can contain a drive and path designation as well as a data file name.

Return value

Integer. Returns 1 for success, and 0 or the negative value of the UNIX error number for the error that caused the method to fail, if failed.

Usage

Pointer to object	Object
`pb->load("./datafile.dat");`	`myblob.load("/datafile.dat");`
`string data_file_name;`	`data_file_name= "c:\new\files\blob_file";`
`pb->load(data_file_name);`	`myblob.load(data_file_name);`

paste()

Description

Inserts the contents of a blob or string into the blob, starting at the position specified in the *offset* parameter. The first blob character is always at offset 0.

Syntax

paste(int *offset*, {blob | string} **source*);

Parameter	Description
offset	Position to start inserting the data. The first character in a blob is at offset 0.
source	Address of the blob or string from which to copy the data.

Return value

Integer. Returns the number of characters of data pasted into the blob.

Examples

```
blob myblob;
string new_stuff;
myblob = "abcdefg";
new_stuff = "123456";
myblob.paste(4, &new_stuff);
```

Usage

Pointer to object	Object
`pb->paste(10, &otherblob);`	`myblob.paste(10, &otherblob);`
`pb->paste(5, &physician_name);`	`myblob.paste(5,&local_string);`

save()

Description

Saves the contents of the blob to the file specified in the *filename* parameter. This method creates the file, if necessary. If a file with the same name currently exists, the save() method overwrites the contents of the existing file with the new information. To append to an existing file, use write().

Syntax

save(sring *filename*);

Parameter	Description
filename	A string that contains the name of the file to which the blob contents are saved. Can contain a drive and path designation, as well as the *filename*. All backslashes (\) must be escaped(\\).

Return value

Integer. Returns the number of bytes written to the file, if successful, and the negative value of the UNIX error that caused the method to fail, if failed.

Usage

Pointer to object	Object
`pb->save("./newfile.blo");`	`myblob.save("/tmp/newfile");`
`pb->save ("c:\\"+dir+"\\"+filename);`	`myblob.save("c:\\"+dir+"\\"+filename);`

set()

Description

Works like an assignment operator, causing the replacement of blob contents with the data contained in the blob or string referenced in the *source* parameter.

Syntax

set(string *source*);

set(blob **source*);

Parameter	Description
source	A string or a pointer to a blob that contains the data that replaces the current blob data.

Return value

Returns the blob containing the new data, if successful, or the blob containing the old data, if failed.

Examples

```
blob myblob;
string new_stuff;
myblob = "abcdefg";
new_stuff = "123456";
myblob.set(&new_stuff);
```

Usage

Use either the first two lines or the last two lines:

Pointer to object	Object
`blob patient_address`	`blob patient_address;`
`pb->set(&patient_address);`	`myblob.set(&patient_address);`
`blob *p_address;`	`blob *p_address;`
`pb->set (p_address);`	`pb.set(p_address);`

size()

Description

Calculates the number of characters of data in a blob.

Syntax

size();

Return value

Integer. Returns the number of characters in the blob, or 0 if empty for success, and a negative number, if failed.

Usage

Pointer to object	Object
`int size;`	
`size = pb->size();`	`size=pbsize();`

Examples

```
blob myblob;
int num;
myblob = "abcdefg";
num = myblob.size();
```

snip()

Description

Removes part of a blob, starting at *offset* for a specified *length*. This method releases memory. If the *offset* plus the *length* is greater than the total blob size or if the *length* is -1, the snip() method removes data from the *offset* position to the end of the blob.

Syntax

snip(int *offset*, int *length*);

Parameter	Description
offset	Position to start removing data. The first character in a blob is always at offset 0.
length	Number of characters to remove.

Return value

Integer. Returns the new blob size, if successful, and the original blob size, if failed.

Usage

Pointer to object	Object
`pb->snip(5, 20);`	`myblob.snip(5,20);`

truncate()

Description

Reduces the size of the blob to the number of characters identified in *length*. This method truncates the contents of the blob, starting from the rightmost position until the contents fit the new size. The truncate() method releases memory.

Syntax

truncate(int *length*);

Parameter	Description
length	Number of characters to reduce the size of the blob.

Return value

Integer. Returns the new blob size, if successful, and 0, if failed.

Examples

```
blob myblob;
myblob = "abcdefg123456";
myblob.truncate(7);
```

Usage

Pointer to object	Object
`int smaller_size;`	`int smaller_size;`
`smaller_size = 22;`	`smaller_size = 45;`
`pb->truncate(smaller_size);`	`myblob.truncate(smaller_size);`

write()

Description

Writes the blob contents to the file specified in the *filename* parameter. If the file does not exist, this method creates the file at the location specified. If the file already exists, the write() method appends the contents of the blob to the end of the file. To overwrite an existing file, use the save() method.

Syntax

write(string *filename*);

Parameter	Description
filename	The name of the file in which to save the blob contents. Can be a string or a literal and can contain a drive and path designation as well as the filename. All backslashes (\) must be escaped (\\). Enclose a literal in double-quotes.

Return value

Integer. Returns the number of bytes written to the file, if successful, and the negative value of the UNIX error that caused the method to fail, if failed.

Usage

Pointer to object	Object
`pb->write("./newfile.blo");`	`string new_file;`
`pb->write ("c:\\new\\files\\pb_file");`	`new_file = "/tmp/pb_hold";`
	`myblob.write(new_file);`
	`myblob.write("./newfile.blo");`
	`myblob.write ("c:\\new\\files\\pb_file");`

Database interface objects

Database interface (DBI) objects (clDbi) provide connectivity support to a database.

For more information, see the *e-Biz Impact MSG-IDE Guide* and the *e-Biz Impact TRAN-IDE Guide*.

Note The ODBC layer native to e-Biz Impact is the current implementation. Dbtil is no longer the underlying implementation.

begin()

Description

Begins a unit of work. An explicit way for users to control units of work; however, the users are responsible for committing or rolling back their own work.

Syntax

begin();

Return value

Always returns 1.

Usage

Pointer to object	Object

`pdbi->begin();`	`inline Str = "insert into dltable(ID, LNAME) values(6, 'Scotty')";`
	`my_dbi.begin();`
	`my_dbi.setStmt("inline_stmt");`
	`my_dbi.exec();`
	`my_dbi.commit();`
	`my_dbi.deinitialize();`
	or
	`inlineStr = "insert into dltable(ID, LNAME) values(5, 'Sha')";`
	`my_dbi.begin();`
	`my_dbi.setStmt("inline_stmt");`
	`my_dbi.exec();`
	`my_dbi.rollback();`

close()

Description

Disconnects from a database.

Syntax

close();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`pdbi->close();`	`my_dbi.close();`

columnInfo()

Description

Retrieves data from a specified column in a specified table, such as column type, storage length, precision, and scale attributes, and places the data in the object accessed by the *dest* value.

Note To ensure that *dest* points to the object in which to have the retrieved column information placed, use the bysymbol() function on that object, then pass the returned symoff value into the method, as in the example below. For more information, see the "Introduction" on page 1.

Syntax

columnInfo(symoff *dest*, string *tablename*, string *colname*);

Pattern	Description
dest	symoff to an object to hold the incoming data.
tablename	The name of the table that contains the column with the desired data.
colname	The name of the column that contains the desired data.

Return value

Integer. Returns a positive integer if successful, and zero or negative integer, if failed.

Usage

Pointer to object	Object

`pdbi->columnInfo(bysymbol (out),tbl,col);`	`string tbl;`
	`string col;`
	`string out;`
	`int rv;`
	`tbl = "authors";`
	`col = "city";`
	`rv = my_cldbi.columnInfo(bysymbol (out),tbl,col);`
	`if(!rv)`
	`  erm("columnInfo() FAIL ");`
	`else`
	`{`
	`  erm("columnInfo() PASS ")'`
	`  out.debug()`
	`}`

commit()

Description

Executes a commit statement that registers as permanent any changes in the database made by the statement object. Using commit() prevents a rollback.

Syntax

commit();

Return value

Always returns 1.

Usage

Pointer to object	Object
`pdbi->commit();`	`my_dbi.commit();`

connect()

Description

Connects to the session name defined in clDbi resource field, RsDbSessionName. This name corresponds to the session entry specified in *nnsyreg.dat* file.

Syntax

```
connect();
```

Return value

Integer. Returns a positive integer if successful, and zero or negative integer, if failed.

Usage

Pointer to object	Object
`pdbi->connect();`	`my_dbi.connect()`

connectStr( )

Description

Allows the clDbi object to connect to a data source during runtime.

Note e-Biz Impact versions earlier than version 5.4.5 used OpenTransport as the underlying connection using *nnsyreg.dat*.

Syntax

```
public int connectStr(string strDsn,
                      string strUser,
                        string strPassword)
```

Parameters

- *strDsn* – the DSN name configured on the system.
- *strUser* – the user name with which you want to connect to the DSN.
- *strPassword* – the password for the user name you entered.

Return value

Returns 1 for a successful connection; 0 for a failed connection.

debugOff()

Description

Terminates debugging for the specified object.

Syntax

```
debugOff();
```

Return value

Always returns a 1.

Usage

Pointer to object	Object
`pdbi->debugOff();`	`my_dbi.debugOff();`

See also debugOn()

debugOn()

Description Enables debugging for the specified object and logs detailed messages about the object in the *xlog* file.

Syntax

```
debugOn();
```

Return value Always returns a 1.

Usage

Pointer to object	Object
`pdbi->debugOn();`	`my_dbi.debugOn();`

See also debugOff()

deinitialize()

Description Closes connection and frees statement resources.

Syntax

```
deinitialize();
```

Return value Integer. Returns a positive integer if successful, and zero or negative integer, if failed.

Usage

Pointer to object	Object
`pdbi->deinitialize();`	`my_dbi.deinitialize();`

drop()

Description Purges the current result set, terminates the current statement object execution, and removes the prepared version of the current statement object.

Syntax

```
drop();
```

Return value Integer. Returns a positive integer if successful, and zero or negative integer, if failed.

Usage

Pointer to object	Object
`pdbi->drop();`	`my_dbi.drop();`

exec()

Description

Executes the SQL in the current statement object.

syntax

exec();

Return value

Integer. Returns a positive integer if successful, and zero or negative integer, if failed.

Usage

Pointer to object	Object
`pdbi->exec();`	`my_dbi.exec();`

fetch()

Description

Retrieves the next row of data and uses the current statement object to process the data in the row. Can be used in conjunction with getResultRow() in a loop, as shown in the example below.

Syntax

fetch();

Return value

Integer. Returns a positive integer if successful, 0 for the end of the result set, or negative integer, if failed.

Examples

```
while(resultRows < Rows)
{
     resultRows = db_main.getResultRows();
     if(resultRows == 10)break;
     db_main.fetch();
     db_main.getRowData(resultRows + 1, &dataptr);
     dataptr.debug();
     tempest = dataptr;
     i++;
     zaxxon = i;
     if(tempest.substr(zaxxon) > 1)
     {
          message = "rowcolCheck(" + (i+17)+")";
          WriteResults(message,1 );
     }
     else
     {
          message = "rowcolCheck(" + (i+17)+)";
     }
}
```

getCols()

Description

Retrieves the number of columns in a result set.

Syntax

getCols();

Return value

Integer. Returns the number of columns in the result set, if successful, and a negative integer, if failed.

Usage

Pointer to object	Object

`int rv;`
`rv = pdbi->getCols();`
`erm("getCols(): Number of result-set columns=%d", rv);`
	`int rv;`
	`rv = my_dbi.getCols();`
	`erm("getCols(): Number of result-set columns = %d:, rv);`

getErrInfo()

Description

Retrieves the error string and error code from the database layer and places into *buff*. The *buff* argument points to the error string and error code.

Syntax

getErrInfo(string **buff*);

Parameter	Description
buff	Pointer to a string to hold the error string and error code.

Return value

Integer. Returns a positive integer if successful, and zero or negative integer, if failed.

Usage

Pointer to object	Object

`int rv;`
`string buff;`
`rv = pdbi->getErrInfo(&buff);`
`erm("getErrInfo(): Error code =[%d] Message is %s",rv,buff);`
`buff.debug();`
	`int rv;`
	`string buff;`
	`rv = my_dbi.getErrInfo(&buff);`
	`erm("getErrInfo(): Error code =[%d] Message is %s,rv,buff);`
	`buff.debug();`

getResultRows()

Description

Returns the number of rows fetched after each exec(). Used primarily with fetch().

Syntax

getResultRows();

Return value

Integer. Returns the number of rows found.

Usage

Pointer to object	Object
`pClDbi->getResultRows();`	`my_dbi.getResultRows();`

Examples

```
while(resultRows < Rows)
{
     resultRows = db_main.getResultRows();
     if(resultRows == 10)break;
     db_main.fetch();
     db_main.getRowData(resultRows + 1, &dataptr);
     dataptr.debug();
     tempest = dataptr;
     i++;
     zaxxon = i;
     if(tempest.substr(zaxxon) > 1)
     {
          message = "rowcolCheck(" + (i+17)+")";
          WriteResults(message,1 );
     }
     else
     {
          message = "rowcolCheck(" + (i+17)+)";
     }
}
```

getRowData()

Description

Retrieves the current result set row data and puts the data into the object pointed to by *dataptr*. The *dataptr* parameter points to the current result set row. Note that the index starts at 1, not 0.

Syntax

getrowData(int *rowNum*, blob **dataptr*)

getRowData(long *rowNum*, blob **dataptr*)

getRowData(int *rowNum*, string **dataptr*)

getRowData(long *rowNum*, string **dataptr*)

Parameter	Description
rowNum	Number of the desired data row. Row numbers begin with zero (0).
dataptr	Pointer to a string or blob to hold the result set data row.

Return value

Integer. Returns a positive integer if successful, and zero or negative integer, if failed.

Usage

Pointer to object	Object

`pdbi->getRowData(2, &dataptr);`	`int rv;`
	`blob dataptr;`
	`my_dbi.setStmt("stmt1");`
	`my_dbi.exec();`
	`rv = my_dbi.getRowData(2, &dataptr);`
	`if(!rv)`
	`  erm("getRowData(): METHOD FAILED");`
	`else`
	`  erm("getRowData(): METHOD PASS");`
	`dataptr.debug();`

getRows()

Description

Retrieves the number of rows in a result set.

Syntax

long getRows();

Return value

Long integer. Returns the numbers of rows in a result set, if successful, and a negative integer, if failed.

Usage

Pointer to object	Object

`int rv;`
`rv = pdbi->getRows();`
`erm("getRows(): Number of Rows inresult-set = %d", rv);`
	`int rv;`
	`rv = my_dbi.getRows();`
	`erm("getRows(): Number of Rows inresult-set = %d", rv);`

rollback()

Description

Executes a rollback statement that removes any changes in the database made by the statement object.

Syntax

rollback();

Return value

Always returns 1.

Usage

Pointer to object	Object
`pdbi->rollback();`	`my_dbi.rollback();`

setStmt()

Description

Establishes a new statement object as the current statement object.

Syntax

setStmt(short *StmtID*); or

setStmt(string *StmtName*);

Parameter	Description
StmtID	Short integer specifying the statement object ID number.
StmtName	String containing the statement object name.

Return value

Integer. Returns a positive integer if successful, and zero or negative integer, if failed.

Usage

Pointer to object	Object
`pdbi->setStmt("Stmt1");`	`my_dbi.setStmt("Stmt1");`
`pdbi->exec()`	`my_dbi.exec()`
or	or
`pdbi->setStmt(1)`	`my_dbi.setStmt(1);`
`pdbi->exec()`	`my_dbi.exec()`

I/O file objects

Use I/O file (clfile) objects to work with files. For a list of error conditions, see Appendix B, "Error Conditions."

assoc()

Description

Associates the specified file or directory with the I/O file object, causing the I/O file object to reference the specified disk file or directory. The path within *filename* can be either an absolute or relative path.

Syntax

assoc(string *filename*);

Parameter	Description
filename	String containing the path and name of the file or directory. Use forward slashes (/) in the path.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`string new_file;`	`my_iofobj.assoc("/usr/old_dir");`
`new_file="/usr/tmp/nw_file";`	
`piof->assoc(new_file);`	

chmod()

Description

Changes the permissions of an I/O file object associated file.

Syntax

chmod(short *mode*);

Parameter	Description
mode	Integer value that identifies the new permissions. Must be four digits in length. Use heading zeros (0) to pad the length; for example, 0777.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int mode;`	`my_iofobj.chmod(0777);`
`mode = 0777;`	
`piof_>chmod(mode);`	

close()

Description

Closes an I/O file object associated file.

Syntax

close();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->close();`	`my_iofobj.close();`

copy()

Description

Copies the contents of an I/O file object associated file, naming the new copied file with the value in *newName*. This method creates the new file with the permissions of the source I/O file object. If a file named *newName* already exists, this method overwrites the existing file.

Syntax

copy(string *newName*);

Parameter	Description
newName	The name of the copied file.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`string newName;`	`string newName;`
`newName = "my_copy".`	`newName = “my_copy”;`
`piof->copy(newName);`	`my_iofobj.copy(newName);`

crc()

Description

Reads the I/O file object associated file and uses the CCITT (International Telegraph and Telephone Consultant Committee) algorithm to generate a 16-bit CRC (Cyclic Redundancy Check). The method then returns the CRC, high byte first.

Syntax

crc();

Return value

Integer. Returns the CRC, high byte first, if successful, and > or = 0, if failed.

Usage

Pointer to object	Object
`piof->crc();`	`my_iofobj.crc();`

debugOff()

Description: Terminates debugging for the specified object.

Syntax:

```
debugOff();
```

Return value: Always returns a 1.

Usage:

Pointer to object	Object
`piof->debugOff();`	`my_iofobj.debugOff();`

See also: debugOn()

debugOn()

Description: Enables debugging for the specified object and logs detailed messages about the object in the *xlog* file.

Syntax:

```
debugOn();
```

Return value: Always returns a 1.

Usage:

Pointer to object	Object
`piof->debugOn();`	`my_iofobj.debugOn();`

See also: debugOff()

delete()

Description: Deletes the I/O file object associated file.

Syntax:

```
delete();
```

Return value: Integer. Returns 1 for success, and 0 for failure.

Usage:

Pointer to object	Object
`piof->delete();`	`my_iofobj.delete();`

errno()

Description: Gets the error number returned by the operating system when an I/O file object encounters an error condition.

Syntax

errno();

Return value

Integer. Returns the file error number, if successful. If no error occurred during the last disk operation, no value is returned.

Usage

Pointer to object	Object
`piof->errno();`	`my_iofobj.errno();`

exist()

Description

Verifies that the I/O file object associated file or directory exists at the specified location.

Syntax

exist();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->exist();`	`my_iofobj.exist();`

fixOptns()

Description

Programmatically clears the I/O file object current I/O and write options and sets new permission.

Note This method does not perform error checking.

Syntax

fixOptns(int *opt*);

Argument	Description
opt	The mnemonic associated with the option being set. For any mnemonic not passed in the argument, the option is set to off. To alter multiple options, place a pipe symbol (|) between each mnemonic. See Table 5-1 for a list of mnemonics.

Return value

None.

Usage

Table 5-1 lists available mnemonics for the *opt* argument. Each description corresponds to an option on the user interface screens.

Table 5-1: opt argument mnemonics

Mnemonic	Description
RDONLY	Read Only
WRONLY	Write Only
RDWR	Read and Write Only
APPEND	Append
TRUNC	Truncate
CREAT	Create
EXCL	Exclusive

Pointer to object	Object
`piof->fixOptns(CREAT | TRUNC);`	`my_iofobj.fixOptnjs(RDWR);`

fixPerms()

Description

Programmatically clears an I/O file object's current create permissions settings and sets new permissions.

Note This method does not perform error checking. Use fixprop() to modify an I/O file object's create permissions without clearing the permissions first.

Syntax

fixPerms(int *perm*);

Parameter	Description
perm	The mnemonic associated with the permissions being set. For any mnemonic not passed in the parameter, the permission is set to off. To set multiple permissions, place a pipe symbol (|) between each mnemonic. See Table 5-2 for a list of mnemonics.

Return value

None.

Usage

Table 5-2 lists available mnemonics for the *perm* parameter. Each description corresponds to an option on the user interface screens.

Table 5-2: perm argument mnemonics

Mnemonic	Description
RD_OWNER	Read - Owner
WR_OWNDER	Write - Owner
EX_OWNER	Execute - Owner
RD_GROUP	Read - Group
WR_GROUP	Write - Group
EX_GROUP	Execute - Group
RD_OTHER	Read - Other
WR_OTHER	Writer - Other
EX_OTHER	Execute - Other

Pointer to object	Object
`piof->fixPerms(EX_GROUP);`	`my_iofobj.fixPerms(RD_OTHER|WR_OTHER);`

getFileName()

Description

Retrieves the name of the file associated with the I/O file object.

Syntax

getFileName(string **name*);

Parameter	Description
name	Pointer to the string in which to put the filename of the associated file.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`string name;`	`string name;`
`piof->getFileName(&name)`	`my_iofobj.getFileName(&name);`

getPathName()

Description

Retrieves the path of the file associated with the I/O file object.

Syntax

getPathName(string **path*);

Parameter	Description
path	Pointer to the string to hold the *path* to the associated file.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`string path;`	`string path;`
`piof->getPathName(&path);`	`my_iofobj.getPathName(&path);`

isAssocWith()

Description

Retrieves the name and path to the I/O file object's associated file.

Syntax

isAssocWith(string **filename*);

Parameter	Description
filename	Pointer to the string in which to put the name of and path to the associated file.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`string full_path;`	`string full_path;`
`piof->isAssocWithfull_path);`	`my_iofobj.isAssocWithfull_path);`

isDir()

Description

Verifies that the I/O file object is referencing a directory and not a file.

Syntax

isDir();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->isDir();`	`my_iofobj.isDir();`

isFile()

Description

Verifies that the I/O file object is referencing a file and not a directory.

Syntax

isFile();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->isFile();`	`my_iofobj.isFile();`

isFileLocked()

Description

Determines if a file associated with an I/O file object is locked.

Syntax

isFileLocked();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->isFileLocked();`	`my_iofobj.isFileLocked();`

isSegLocked()

Description

Determines if the specified portion of an I/O file object associated file is locked.

Syntax

isSegLocked(long *size*);

Parameter	Description
size	Number of bytes to examine for a lock.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`long num;`	`long num;`
`num = 3295`	`num = 3295;`
`piof->isSegLocked(num);`	`my_iofobj.isSegLocked(num);`

lockEnforceOff()

Description

Disables file lock enforcement on an I/O file object's associated file on UNIX.

Syntax

lockEnforceOff();

Return value

Integer. Returns 1 for success, and 0 for failure.

usage

Pointer to object	Object
`piof->lockEnforceOff();`	`my_iobobj.lockEnforceOff();`

See also lockEnforceOff()

lockEnforceOn()

Description Enables file lock enforcement on an I/O file object's associated file on UNIX.

Syntax lockEnforceOn();

Return value Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->lockEnforceOn();`	`my_iofobj.lockEnforceOn();`

See also lockEnforceOn()

lockFile()

Description Places an advisory lock on an I/O file object's associated file, blocking if necessary. Blocking is not available on Windows.

Note An advisory lock does not prevent other users from reading and writing to the file. Call lockEnforceOn() before calling lockFile() to place enforcement locking on the file and prevent other users from reading and writing to the file. Enforcement locking is available only on UNIX, and the file must not have "group" execute permissions.

Syntax lockFile();

Return value Integer. Returns 1 for success, and 0 for failure.

Usage To successfully lock a file using lockFile(), you must have opened the file using the open() method, and the file's I/O object must have the "write only" or the "read and write" I/O option set. You should always call unlockFile() once you are finished with the file. When calling lockFile() from a program running on a UNIX platform, if another user has locked the file, then your program blocks this lockFile() call until the other user unlocks the file. When calling lockFile()

method from a program running on Windows, if another user has locked the file, then your program makes multiple attempts to establish the lock, but it will not perform blocking.

Pointer to object	Object
`piof->lockFile();`	`my_iobobj.lockFile();`

lockSeg()

Description

Enables advisory locking of part of an I/O file object's associated file, blocking if necessary. Blocking is not available on Windows.

Note Use pos() before locking a file segment to determine the current location of the file position indicator. Reading or writing to a file causes the file position indicator to move. Because unlockSeg() always unlocks a file from the indicator's current location, before calling unlockSeg(), reposition the file position indicator to the position it was at when you called lockSeg() so that you unlock exactly the segment you locked.

An advisory lock does not prevent other users from reading and writing to the file segment. Call lockEnforceOn() before calling lockSeg() to place enforcement locking on the file segment and prevent other users from reading and writing to it. Enforcement locking is available only UNIX, and the file must not have "group" execute permissions.

Syntax

lockSeg(log *size*);

Parameter	Description
size	Number of bytes to examine for a lock.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`long num;`	`long num;`
`num = 45783`	`num = 45783;`
`piof->lockSeg(num);`	`my_iofobj.lockSeg(num);`

mkNewFile()

Description

Creates a file at the specified *location*, writes the contents of the *data* parameter to it, and associates the I/O file object to the new file.

Syntax

mkNewFile(string **location*, blob **data*);

Parameter	Description
location	Pointer to the string object that identifies where to create the file.
data	Pointer to the blob from which to retrieve the data to write to a file.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

The *location* parameter can be an absolute or relative path. When *location* contains at least one forward slash (/), the software interprets it as an actual path. Use a forward slash (/) in the path for both Windows and UNIX paths. Use "./" to specify the current directory.

Pointer to object	Object
`string loc;`	`string loc;`
`blob data;`	`blob data;`
`loc = "/usr/tmp";`	`loc = "/usr/tmp";`
`data = "a bunch of data..."`	`data = "a bunch of data..."`
`piof->mkNewFile(&loc, &data);`	`my)_iofobj.mkNewFile(&loc, &data);`

mkTmp()

Description

Creates a new file and associates it with the I/O file object.

Syntax

mkTmp();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->mkTmp();`	`my_iofobj.mkTmp();`

nbLockFile()

Description

Enables advisory locking of an I/O file object's associated file. Does not block.

Syntax

nbLockFile();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->nbLockFile();`	`my_iofobj.nbLockFile();`

nbLockSeg()

Description

Enables advisory locking of part of an I/O file object's associated file. Does not block.

Syntax

nbLockSeg(long *size*);

Parameter	Description
size	Number of bytes to lock.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Use pos() before locking a file segment to determine the current location of the file position indicator. Reading from or writing to a file causes the file position indicator to move. Because unlockSeg() always unlocks from the indicator's current location, before calling unlockSeg(), reposition the file position indicator to the position it was at when you called nbLockSeg() so that you unlock exactly the segment you had locked. An advisory lock does not prevent other users from reading and writing to the file segment. Call lockEnforceOn() before calling nbLockSeg() to place enforcement locking on the file segment and prevent other users from reading and writing to the file segment. Enforcement locking is available only on UNIX and the file must not have "group" execute permissions.

Pointer to object	Object

`long num;`
`num = 45783`
`piof->nbLockSeg(num);`
	`my_iofobj.nbLockSeg(num);`

open()

Description

Opens the file associated with the I/O file object using the options and permissions set in the I/O file object. If the file is already open, the I/O file object closes it before executing the open call.

Syntax

open();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->open();`	`my_iofobj.open();`

openDir()

Description

Opens the directory associated with the I/O file object and creates an internal listing of the contents.

Syntax

openDir(string *mask*);

Parameter	Description
mask	UNIX regular expression that identifies the files or directories to place in the internal listing. To place all the associated directory contents into the internal listing, use double quotes (“_”) as the *mask*.

Return value

Integer. Returns the number of items placed into the internal listing, if successful, and -1, if failed.

Usage

Commands	Command Modifiers
.(any character)	*(*none or more*)
[(character list)	+(*one or more*)
[^(not in list)	{*m,n*} {*range*}
$(end of match)	

Pointer to object	Object
`piof->openDir(“P[a-z]”);`	`myiofobj.openDir(“”);`

Examples

```
string mymask;
mymask = “[a-d]”;
mymask = “[^a-d]”;
mymask = “.[m-p]”
mymask = “sybs$”
mymask = “a[a-z].*001
```

pos()

Description

Determines the location of the file position indicator in an I/O file object's associated file.

Syntax

pos();

Return value

Integer. Returns the value of the file position indicator, if successful, and -1, if failed.

Usage

Pointer to object	Object
`long rv;`	`long rv;`
`rv = piof->pos();`	`rv - myiofobj.pos();`

posCurrent()

Description

Moves the file position indicator in a file associated with an I/O file object based on current position.

Syntax

posCurrent(long *size*);

Parameter	Description
size	Number of bytes to move. Use a positive value to move forward or a negative value to move backward.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`long to_beg;`	`long to_end;`
`to_beg = -527;`	`to_end = 394;`
`piof->posCurrent(to_beg);`	`my_iofobj.posCurrent(to_end);`

posEnd()

Description

Moves the file position indicator in an I/O file object's associated file to the end of the file.

Syntax

posEnd();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->posEnd();`	`my_iofobj.posEnd();`

posStart()

Description

Moves the file position indicator in an I/O file object's associated file to the beginning of the file.

Syntax

posStart();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->posStart();`	`my_iofobj.posStart();`

read()

Description

Reads the contents of a file associated with an I/O file object. This reads the contents of the file from the current offset for the number (*num*) of bytes, and places the bytes read into the *buffer*.

Syntax

read({blob | string} **buffer*, int *num*);

Parameter	Description
buffer	Pointer to a string or blob *buffer* to hold the bytes read.
num	Integer that specifies the number of bytes to read.

Return value

Integer. Returns the number of bytes read, if successful, and -1, if failed.

Usage

Pointer to object	Object
`blob buff;`	`string buffer;`
`int num;`	`int number;`
`num = 25;`	`number = 13;`
`piof->read(&buff, int num)`	`my_iofobj.read(&buffer, number);`

readDir()

Description

Reads the files from the list generated by openDir(). readDir() takes the first file or directory in the internal list generated by OpenDir() method and associates the file or directory to the I/O file object in the *file_obj* parameter. Then, readDir() removes the first file or directory from the internal list and a subsequent call to readDir() associates the next file or directory in the internal list to *file_obj*. Each call to readDir() takes the file or directory in the internal list, associates it with *file_obj*, then removes the file or directory from the list. When no files/directories remain, readDir() returns a 0.

Syntax

readDir(clFile **file_obj*);

Parameter	Description
file_obj	Pointer to the I/O file object to associate with the file or directory read from the list.

Return value

Integer. Returns the following if successful:

- ISFILE – read a file.
- ISDIR – read a directory.
- NOEXIST – deleted a file since the last time that openDir() was called.

Returns 0, indicating that an error occurred or an absence of files in the list.

Usage

Pointer to object	Object
`piof->readDir (&p_other_iof);`	`my_iofobj.readDir(&other_iofobj);`

readFile()

Description

Reads the entire contents of the I/O file object's associated file into the *dataBlob* parameter. Because this method reads the entire file, it is generally used only when moving files through a system. Maximum file size is determined by operating system limits.

Syntax

readFile(blob **dataBlob*);

Parameter	Description
dataBlob	Pointer to the blob object to hold the data read from the file.

Return value

Integer. Returns the number of bytes of data read, if successful, and 0, if failed.

Usage

Pointer to object	Object
`blob data_read;`	`blob data_read;`
`piof->readFile (&data_read);`	`my_iofobj.readFile (&data_read);`

rename()

Description

Renames the I/O file object associated file.

Syntax

rename(string *newName*);

rename(blob **newName*);

rename(string *newName*, int *flag*);

rename(blob **newName*, int *flag*);

Parameter	Description
newName	The string or blob that contains the name of the new file.
flag (optional)	Flag to specify rename operations: • Set to 1 to specify that if a rename fails, then attempt the copying of the associated file to the new name. • Do not set, or set to 0 to instruct the program to not attempt the copy of the associated file if the rename fails.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object

`piof->rename("oldstuff", 1);`	`string newName;`
	`newName = "acct_data";`
	`my_iofobj.rename(newName);`

size()

Description

Finds the *size* of an I/O file object associated file.

Syntax

size();

Return value

Integer. Returns the size of file in bytes, if successful, and -1, if failed

Usage

Pointer to object	Object
`piof->size();`	`my_iobobj.size();`

unAssoc()

Description

Removes the association between the I/O file object and its associated file.

Syntax

unAssoc();

Return value

Integer. Returns 1, if successful, indicating the presence of a file associated with the I/O File Object. Returns 0, if failed, indicating that no file is associated with the I/O file object.

Usage

Pointer to object	Object
`piof->unAssoc();`	`my_iofobj.unAssoc();`

unlockFile()

Description

Unlocks an I/O file object's associated file.

Syntax

unlockFile();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`piof->unlockFile();`	`my_iofobj.unlockFile();`

unlockSeg()

Description

Unlocks the specified portion of an I/O file object's associated file.

Syntax

unlockSeg(long *size*);

Parameter	Description
size	Number of bytes to examine for an unlock.

Return value

Integer. Returns 1 for success, and 0 for failure, indicating that the segment could not unlock or that the segment was not locked initially.

Usage

Pointer to object	Object

`long num;`
`num = 875`
`piof->unlockSeg(num);`
	`my_iofobj.unlockSeg(num);`

write()

Description

Writes the contents of the *buffer* for length (*len*) to the I/O file objects's associated file starting from the current offset position in the file. If *len* equals zero, all data in the *buffer* is written to the file. If the I/O file object has "append" set to "on," this method always appends the data to the file.

Syntax

write(blob **buffer*, int *len*);

write(string **buffer*, int *len*);

write(string *buffer*, int *len*);

Parameter	Description
buffer	String object or a pointer to a string or blob object containing the data from which to write to the file.
len	The number of bytes to write. If *len* is 0, it writes all data in the *buffer* to the file.

Return value

Integer. Returns the number of bytes written to the file, if successful, and 0, if failed.

Usage

Pointer to object	Object

`my_iofobj.write("new data", 0);`	`string buffer;`
	`buffer = "some old data plus new";`
	`piof->write(&buffer, 15);`

Map objects

Map (clMap) objects store string pairs in a <key, element> format. Map objects are mainly used in the IBM WebSphere MQ functionality to pass Open Transport properties from an acquisition AIM to the delivery AIM.

add()

Description

Adds a string pair to clMap.

Syntax

int add(string *key*, string *element*);

Parameter	Description
key	String that identifies the item to be added.
element	String that contains the data that goes with the *key*.

Return value

Always returns 1.

Usage

Pointer to object	Object
`pclMapObj->add("myKey", "myElement");`	`myclMapObj.add("myKey", "myElement");`

clear()

Description

Removes all string pairs stored in the object.

Syntax

int clear ();

Return value

Always returns 1.

Usage

Pointer to object	Object
`pclMapObc->clear()`	`myclMapObj.clear( );`

clearKey()

Description

Removes a string pair identified by a key.

Syntax

int clearKey(string *key*);

Parameter	Description
key	String value of the *key* to remove. A literal string ("`string`") may also be used.

Return value

Always returns 1.

Usage

Pointer to object	Object
`pyclMapObj->clearKey("myKey");`	`myclMapObj.clearKey("myKey");`

get()

Description

Retrieves a string pair or string element from clMap object, depending on the parameter provided. There are two methods.

Syntax

int get(string *key*, string **pElement*);

Parameter	Description
key	String of the *key* that looks up the element.
pElement	Pointer to the string to receive the element data that is paired with the supplied *key*.

Return value

Integer. Returns 1 if data is retrieved; otherwise, returns 0.

Usage

Takes a string (*key*) and a string pointer (*pElement*) as parameters. The first string parameter is used as the *key* to retrieve the corresponding element. The string pointer parameter is used for returning retrieved data.

Pointer to object	Object
`string myElement;`	`string myElement;`
`pclMapObj->get ("myKey",&myElement);`	`myClMapObj.get("myKey",&myElement);`

get()

Description

Retrieves a string pair given an index.

Syntax

int get(int *index*, string **pKey*, string **pElement*);

Parameter	Description
index	*Index* into the clMap of which key/element pair to retrieve. The first pair has an *index* value of zero.
pKey	Pointer to the location to put the key value in once retrieved.
pElement	Pointer to the location to put the element value in once it is retrieved.

Return value

Integer. Returns 1 if data is retrieved; otherwise, returns 0.

Usage

Takes an integer and two string pointers as parameters. The integer is used as an *index* to indicate which string pair to retrieve in the object. Notice the ordering of string pairs does not equal the order in which they were added to the object. The two string pointers are used for returning the string pair.

Pointer to object	Object
`string key;`	`string key;`
`string element;`	`string element;`
`pclMapObj->get (3, &key, &element);`	`myclMapObj.get (3, &key, &element);`

size()

Description

Use to determine how many string pairs are in an object.

Syntax

int size ();

Return value

Integer. Returns the number of string pairs in the object.

Usage

Pointer to object	Object
`int numPairs;`	`int numPairs;`
`numPairs = pmyclMapObj->size( );`	`numPairs = myclMapObj.size( );`

NDO objects

NDO (clNdo) objects provide support for the New Era Data Objects (NDOs), which allow you to use, manipulate, and access NNDOObjects and NNDODataNodes within ODL applications. This allows you to work with data in tree form, such as processing an XML document. The clNdo object can function as both an NNDOObject and NNDODataNode with the ODL application, depending on the usage. Generally, clNdo contains a handle to an NNDOObject and a handle to its root node, NNDODataNode.

Note The parameters preceded by an asterisk (*) are passed in by pointers.

attributeEnd()

Description

Test the internal iterator to see if it has reached the end of the map of attributes of the current NNDODatanode associated with this clNdo. Can be used as a test condition in a loop to retrieve attributes.

Syntax

attributeEnd()

Return value

Integer. Returns 1 if the internal iterator is a past one of the last attributes of the map, and 0 if at the end.

Examples

```
if(!pCINdo->attributeEnd())
{
pCINdo->getCurrentAttribute
   (&attributeTag,&attributeValue);
pCINdo->nextAttribute();
}
```

copyNDO()

Description

Copies and overwrites the current clNdo contents to another clNdo object. If the current clNdo is not associated with an NNDOObject, one is created.

Syntax

copyNDO(clNdo **pClNdo*)

Parameter	Description
pClNdo	Pointer to another clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

copyDataNode()

Description

Copies and overwrites the data node to which the current clNdo is associated. If the current clNdo is not associated with an NNDOObject, one is created and the copied node becomes the root node of the newly crated NNDOObject.

Syntax

copyDataNode(clNdo **pClNdo*)

Parameter	Description
pClNdo	Pointer to another clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

createChild()

Description

Creates a child node for the current clNdo object.

Syntax

createChild (string *childName*)

Parameter	Description
childName	String that contains the child name provided.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object

`int rv;`
`clNdo *pCINdp()'`
`rv=pCINdo0->`
`  createChild("child1");`
	`int rv;`
	`clNdo clNdo0;`
	`rv=clNdo0.createChild(child1");`

createChild()

Description

Creates a child node for the current clNdo object. In this version, a pointer to another clNdo is passed to serve as a handle to the newly created child.

Syntax

createChild(clNdo **pClNdo*, string *childName*)

Parameter	Description
pclNdo	Pointer to another clNdo object to hold the newly created child.
childName	String that contains the child name provided.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object

`int rv;`
`clNdo *pclNdo0;`
`clNdo clNdo1;`
`rv = pClNdo0->`
`  createChild(&clNdo1, "child"`
	`int rv;`
	`clNdo ClNdo0;`
	`clNdo clNdo1;`
	`rv = clNdo0.createChild`
	`  (&clNdo1, 'child");`

createRoot()

Description

Creates the root node of an NDO. To clear an existing root, use ndoClear().

Syntax

createRoot(string *rootName*)

Parameter	Description
rootName	String that contains the root name.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`clNdo *pCINdo0;`	`clNdo clNdo0;`
`rv=pCINdo0p->` `  createRoot(CLNDO1-ROOT");`	`rv=clNdo0.createRoot` `  ("CLNDO1-ROOT");`

deserializeNCF()

Description

Deserializes the New Era Canonical Formats (NCF) contained in the blob into this clNdo object.

Syntax

deserializeNCF(blob **pBlob*)

Parameter	Description
pBlob	Pointer to a blob that contains the serialized NCF.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`clNdo clNdo6;`	`clNdo clNdo6;`
`int rv;`	`int rv;`
`blob testBlob;`	`blob testBlob;`
`//do something to load the blob` `//with NCF`	`//do something to load the blob` `//with NCF`
`rv=pCINdo6->` `  deserializeNCF(&testBlob)`	`rv = clNdo6.deserializeNCF` `  (&testBlob);`

deserializeNCFnsd()

Description

Note Prior to this call, a successful call to setSchema() on this clNdo object is required.

Deserializes the NDO nonself-describing NCF contained in the blob into this clNdo object.

Syntax

deserializeNCFnsd(blob *pBlob*)

Parameter	Description
pBlob	Pointer to a blob containing a serialized NCF.

Return value

None.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`clNdo *pcCINdo6;`	`clNdo cINdo6;`
`blob testBlob;`	`blob testBlob;`
`blob schemaBlob;`	`blob schemaBlob;`
`testBlob.load("rootmy.ncf");`	`testBlob.load("rootmy.ncf");`
`schemaBlob.load ("NAX8.IC.root.ncm");`	`schemaBlob.load ("NAX8.IC.root.ncm");`
`rv=pCINdo6-> setSchema(&SchemaBlob);`	`rv=pCINdo6-> setSchema(&SchemaBlob);`
`rv=pCINdo6-> deserializeNCFnsd(&TestBlob);`	`rv=pCINdo6-> deserializeNCFnsd(&TestBlob);`

deserializeXML()

Description

Deserializes the XML contained in the blob into this clNdo object.

Syntax

deserializeXML(blob *pBlob*)

Parameter	Description
pBlob	Pointer to a blob that contains the serialized XML.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object

`clNdo clNdo6;`
`int rv;`
`blob testBlob;`
`testBlob.load;`
`rv=pclNdo6->`
`  deserializeXML(&testBlob)`
	`clNdo clNdo6;`
	`int rv;`
	`blob testBlob;`
	`rv = clNdo6.deserializeXML`
	`  (&testBlob);`

findChild()

Description

Finds a particular child by name from the current clNdo. The found child is then saved to a clNdo object at the address passed into this method.

Syntax

findChild(clNdo **pClNdo*, string *childName*)

Parameter	Description
pClNdo	Address of a clNdo object to save the found child of this NDO.
childName	String that contains the child name to find.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object

`clNdo NDO_HOLD_3;`
`clNdo *pNDO_Child_02;`
`//intermediate code not shown`
`pNDO_Child_02->findChild`
`  (&NDO_HOLD_3,"IMPALA");`
	`clNdo NDO_HOLD_3;`
	`clNdo NDO_Child_02;`
	`//intermediate code not shown`
	`NDO_Child_02.findChild`
	`  (&NDO_HOLD_3,"IMPALA");`

findChildAttribute()

Description

Finds a particular child and attribute from the current clNdo. The found child is then saved to a clNdo object at the address passed to this method.

Syntax

findChildAttribute(clNdo **pClNdo*,
string *childName*,
string *Attribute*,
string *AttributeValue*)

Parameter	Description
pClNdo	Address of a clNdo object to save the found child of this NDO.
childName	String that contains the child name to find.
Attribute	String that contains the attribute.
AttributeValue	String that contains the value of the attribute.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`clNdo NDO_HOLD_1;`	`clNdo NDO_HOLD_1;`
`clNdo *pNDO_Child_01;`	`clNdo NDO_Child_01;`
`//intermediate code not shown`	`//intermediate code not shown`

`pNDO_Child_01->`
`  findChildAttribute`
`  (&NDO_HOLD_1,"PINTO",`
`  "COLOR","GREEN");`
	`NDO_Child_01.findChildAttribute`
	`  (&NDO_HOLD_1,"PINTO",`
	`  "COLOR", "GREEN");`

findChildR()

Description

Recursive version of the findChild() that searches all the children of a node for a node of a given name.

Syntax

findChildR(clNdo **pClNdo*,
string *childName*)

Parameter	Description
pClNdo	Pointer to a clNdo that provides a handle to the found child.
childName	String that contains the child name to find.

first()

Description

Changes the clNdo to hold the first node among the current node's siblings.

Syntax

first()

Integer — Integer. Returns 1 for success, 0 if the current node does not have a parent, and -1 for errors.

getAttribute()

Description — Gets the value of a specific node attribute.

Syntax — getAttribute(string *attribute*, string **pValue*)

Parameter	Description
attribute	String to the attribute desired.
pValue	Pointer to the string to store the attribute data.

Return value — Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`clNdo *pNDO_HOLD_1;`	`clNdo NDO_HOLD_1;`
`string tempString1;`	`string tempString1;`
`//intermediate code not shown`	`//intermediate code not shown.`
`pNDO_HOLD_1->getAttribute ("TYPE",&tempString1);`	`NDO_HOLD_1.getAttribute ("TYPE",&tempString1);`

getAttributeCount()

Description — Returns the number of attributes contained in the current node that clNdo is associated to.

Syntax — getAttributeCount()

Return value — Integer. Returns a positive integer for the number of attributes contained in the current node, and 0 when the current clNdo is not associated with a data node or the current data node has no attributes.

getCurrentAttribute()

Description — Retrieves the current attribute tag and value and saves them as two strings.

Syntax — getCurrentAttribute(string **attributeTag*, string **attributeValue*)

Parameter	Description
attributeTag	Pointer to a string to save the current node's attribute tag.
attributeValue	Pointer to a string to save the current node's attribute value.

Return value

Integer. Returns 1 for success, and 0 for failure.

getData()

Description

Gets character data from the clNdo object.

Syntax

getData(char **pChar*)

Parameter	Description
pChar	Pointer to a character data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object

`int rv;`
`char c;`
`clNdo *pClNdo1;`
`.`
`.`
`rv=pclNdo1->getData(&c);`
	`rv=cINdo1.getData(&c);`

getData()

Description

Gets integer data from the clNdo object.

Syntax

getData(int **pInt*)

Parameter	Description
pInt	Pointer to integer data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`int i;`	`int i;`
`clNdo *pCINdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv=pCINdo1->getData(&i);`	`rv=cINdo1.getData(&i);`

getData()

Description

Gets long data from the clNdo object.

Syntax

getData(long **pLong*)

Parameter	Description
pLong	Pointer to long data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`long l;`	`long l;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv=pCINdo1->getData(&I);`	`rv = clNdo1.getData(&l);`

getData()

Description

Gets short data from the clNdo object.

Syntax

getData(short **pShort*)

Parameter	Description
pShort	Pointer to short data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`short s;`	`short s;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv=pCINdo1->getData(&s);`	`rv = clNdo1.getData(&s);`

getData()

Description

Gets string data from the clNdo object.

Syntax

getData(string **pString*)

Parameter	Description
pString	Pointer to string data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`string s;`	`string s;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pCINdo1->getData(&s);`	`rv = clNdo1.getData(&s);`

getData()

Description

Gets float data from the clNdo object.

Syntax

getData(float **pFloat*)

Parameter	Description
pFloat	Pointer to float data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`float f;`	`float f;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pCINdo1->getData(&f);`	`rv = clNdo1.getData(&f);`

getData()

Description

Gets blob data from the clNdo object.

Syntax

getData(blob **pBlob*)

Parameter	Description
pBlob	Pointer to blob data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`blob b;`	`blob b;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`.`	`.`
`rv = pCINdo1->getData(&b);`	`rv = clNdo1.getData(&b);`

getData()

Description

Gets unsigned character data from the clNdo object.

Syntax

getDataUnsigned(char **pChar*)

Parameter	Description
pChar	Pointer to an unsigned character to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`unsigned char uc;`	`unsigned char uc;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pCINdo1->getData(&uc);`	`rv = clNdo1.getData(&uc);`

getData()

Description

Gets unsigned short data from the clNdo object.

Syntax

getDataUnsigned({*short* | *long*} {**pShort* | **pLong*})

Parameter	Description
pShort	Pointer to unsigned short data to store the data from the clNdo.
pLong	Pointer to unsigned long data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`unsigned short s;`	`unsigned short s;`
`clNdo *pCINdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pCINdo1->getData(&us);`	`rv = clNdo1.getData(&us);`
or	or
`int rv;`	`int rv;`
`unsigned long ul;`	`unsigned long ul;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pCINdo1->getData(&ul);`	`rv = clNdo1.getData(&ul);`

getData()

Description

Gets unsigned integer data from the clNdo object.

Syntax

getDataUnsigned(int *pInt*)

Parameter	Description
pInt	Pointer to an unsigned integer data to store the data from the clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`unsigned int i;`	`unsigned int i;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pCINdo1->getData(&ui);`	`rv = clNdo1.getData(&ui);`

getName()

Description

Gets the name of the current NNDODataNode associated with this clNdo object.

Syntax

getName(string **nodeName*)

Parameter	Description
nodeName	Pointer to a string to store the name of a node.

Return value

None.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`string s;`	`string s;`
`clNdo *pCINdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pCINdo1->getName(&s);`	`rv = clNdo1.getName(&s);`

getFirstChild()

Description

Retrieves the first child of the current NNDODataNode and saves the parent node to the target clNdo. It does not change the NNDOObject association.

Syntax

getFirstChild(clNdo *pClNdo)

Parameter	Description
pClNdo	Pointer to the target clNdo that contains the first child node.

Return value

Integer. Returns 1 for success, 0 if the current node does not have children, and -1 for errors.

getLastChild()

Description

Retrieves the last child of the current NNDODataNode and saves the parent node to the target clNdo. It does not change the NNDOObject association.

Syntax

getLastChild(clNdo *pClNdo)

Parameter	Description
pClNdo	Pointer to the target clNdo that contains the last child node.

Return value

Integer. Returns 1 for success, 0 if the current node does not have children, and -1 for errors.

getParent()

Description

Retrieves the parent of the current NNDODataNode and saves the parent node to the target clNdo. It does not change the NNDOObject association.

Syntax

getParent(clNdo *pClNdo)

Parameter	Description
pClNdo	Pointer to the target clNdo that contains the parent node.

Return value

Integer. Returns 1 for success, 0 if the current node does not have a parent, and -1 for errors.

grabAttribute()

Description

Retrieves an attribute from the node that the current clNdo is associated with and stores the attribute tag and value in the two string pointers that are passed to it.

Syntax

grabAttribute(int *attributeIndex*,
string **attributeTag*,
string **attributeValue*)

Parameter	Description
attributeIndex	Index of the attribute pair to retrieve.
attributeTag	Pointer to a string to hold the attribute tag at a particular index.
attributeValue	Pointer to a string to hold the attribute value at a particular index.

Return value

Integer. Returns 1 for success, 0 if the *attributeIndex* is out of range.

Usage

Pointer to object	Object
`clNdo *pCINdo1;`	`clNdo clNdo1;`
`string tempString1;`	`string tempString1;`
.	.
.	.
`pCINdo1->getAttribute ("TYPE",&tempString1);`	`cINdo1.getAttribute ("TYPE",&tempString1);`

isFirst()

Description

Checks if the current clNdo is the first among siblings.

Syntax

is First()

Return value

Integer. Returns 1 for the first node, 0 there are no nodes, and -1 for errors.

isLast()

Description

Checks if the current clNdo is the last among siblings.

Syntax

is Last()

Return value

Integer. Returns 1 for the last node, 0 there are no nodes, and -1 for errors.

last()

Description — Changes the clNdo to hold the last node among the current node's siblings.

Syntax — last()

Return value — Integer. Returns 1 for success, 0 if the current node does not have a parent, and -1 for errors.

ndoClear()

Description — Clears the contents of clNdo. Removes the associations to the NDO and NNDODataNodes that it contains. Use this method before calling createRoot().

Syntax — ndoClear()

Return value — Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`rv = pClNdo1->ndoClear();`	`rv = clNdo1.ndoClear();`

ndoDump()

Description — Dumps the contents of the NNDOObject associated with this clNdo object into an *xlog*.

Syntax — ndoDump()

Return value — Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`cINdo *pCINdo1;`	`cINdo cINdo1;`
.	.
.	.
`pCINdo1->ndoDump();`	`cINdo1.ndoDump();`

next()

Description — Changes the current clNdo to hold the next sibling node.

Syntax next()

Return value Integer. Returns 1 for success, 0 if the current node is already the last node or does not have a parent, and -1 for errors.

nextAttribute()

Description Sets the internal attribute iterator of a clNdo to the next attribute relative to the current attribute to which it is assigned. Use this method with getCurrentAttribute(), setAttributeBegin(), and setAttributeEnd().

Syntax nextAttribute()

Return value Integer. Returns 1 for success, 1 when the current clNdo is not associated with a data node.

nodeDump()

Description Dumps the contents of the NNDOObject associated with this clNdo object into an *xlog*.

Syntax nodeDump()

Return value Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`clNdo NDO_HOLD_1;`	`NDO_HOLD_1.nodeDump();`
`clNdo *pCINdo1;`	`clNdo cINdo1;`
.	.
.	.
.	.
`pCINdo1->nodeDump();`	`cINdo1.nodeDump();`

prev()

Description Changes the current clNdo to hold the previous sibling node.

Syntax prev()

Return value Integer. Returns 1 for success, 0 if the current node is already the last node among the siblings or does not have a parent, and -1 for errors.

prevAttribute()

Description

Sets the internal attribute iterator of a clNdo to the previous attribute relative to the current attribute it is assigned to. Use with getCurrentAttribute(), setAttributeBegin(), and setAttributeEnd().

Syntax

prevAttribute

Return value

Integer. Returns 1 for success, and 0 when the current clNdo is not associated with a data node.

removeChild()

Description

Removes a child of the NNDODataObject associated with the current clNdo. The child can be found by the findChild() or findChildAttribute() methods.

Syntax

removeChild(clNdo **pClNdo*)

Parameter	Description
pClNdo	Pointer to a clNdo object associated with a child of the current clNdo.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`clNdo ClNdoChild;`	`clNdo clNDdoChild;`
`clNdo *pClNDdoParent->`	`clNdo clNDdoParent;`
`//clNdoChild is a child of //clNdoParent.`	`//clNdoChild is a child of //clNdoParent.`
`rv = clNdoParent.removeChild (&clNdoChild);`	`rv = clNdoParent.removeChild (&clNdoChild);`

saveNCF()

Description

Saves the contents of the clNdo into a self-describing NCF in a file.

Note This overwrites an existing file with the same name.

Syntax

saveNCF(string *filename*)

Parameter	Description
filename	File that contains the saved clNdo in NCF form.

Return value

Integer. Returns 1 for success, and 0 for failure.

saveNCFnsd()

Description

Saves the contents of the clNdo into a nonself-describing NCF in a file.

Note This overwrites an existing file with the same name.

Syntax

saveNCFnsd(string *filename*)

Parameter	Description
filename	File that contains the saved clNdo in NCF form.

Return value

Integer. Returns 1 for success, and 0 for failure.

saveXML()

Description

Saves the contents of clNdo in an XML form in a file.

Note This overwrites an existing file with the same name.

Syntax

saveXML(string *filename*)

Parameter	Description
filename	File that contains the saved clNdo in XML form.

Return value

Integer. Returns 1 for success, and 0 for failure.

serializeNCF()

Description

Serializes the contents within this clNdo object into the designated blob using NCF.

Syntax

serializeNCF(blob *pBlob*)

Parameter	Description
pBlob	Pointer to a blob to hold the serialized NCF.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`clNdo *pClNDdo6;`	`clNdo clNDdo6;`
`int rv;`	`int rv;`
`blob testBlob;`	`blob testBlob;`
`.`	`.`
`.`	`.`
`rv=pCINdo6->serializeNCF (&testBlob);`	`rv = clNdo6.serializeNCF (&testBlob);`

serializeXML()

Description

Serializes the contents within this clNdo object into the designated blob using an XML format.

Syntax

serializeXML(blob *pBlob*)

Parameter	Description
pBlob	Pointer to a blob to hold the serialized XML.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`clNdo *pClNDdo1;`	`clNdo clNDdo1;`
`int rv;`	`int rv;`
`blob testBlob;`	`blob testBlob;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->serializeXML (&testBlob);`	`rv = clNdo1.serializeXML (&testBlob);`

serializeNCFnsd()

Description

Serializes the contents within this clNdo object into the designated blob using a nonself-describing NCF.

Syntax

serializeNCFnsd(blob *pBlob*)

Parameter	Description
pBlob	Pointer to a blob to hold the serialized NCF.

Return value

None.

Usage

Pointer to object	Object
`clNdo *pClNDdo6;`	`clNdo clNDdo6;`
`int rv;`	`int rv;`
`blob testBlob;`	`blob testBlob;`
`.`	`.`
`.`	`.`
`rv = pclNdo6->serializeNCFnsd (*testBlob);`	`rv = clNdo6.serializeNCFnsd (&testBlob);`

setAttribute()

Description

Resets or creates an attribute to a given value of the NNDODataNode associated with this clNdo.

Syntax

setAttribute(string *attribute*, string **attributeValue*)

Parameter	Description
attribute	String that contains the attribute.
attributeValue	String that contains the value of the attribute.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`clNdo *pNDO_Child_01a;`	`clNdo NDO_Child_01a;`
`.`	`.`
`.`	`.`
`pNDO_Child_01a->setAttribute ("COLOR","GREEN");`	`NDO_Child_01a.setAttribute ("COLOR","GREEN");`

setAttributeBegin()

Description: Sets the internal attribute iterator of a clNdo to the beginning of the map of attributes. This object is used with nextAttribute(), prevAttribute(), and getCurrentAttribute() to iterate and retrieve each attribute without the attribute name.

Syntax: setAttributeBegin()

Return value: Integer. Returns 1 for success, and 0 when the current clNdo is not associated with a data node.

setAttributeEnd()

Description: Sets the internal attribute iterator of a clNdo to one pass the last attribute element in its attribute map.

Syntax: setAttributeEnd()

Return value: Integer. Returns 1 for success, and 1 when the current clNdo is not associated with a data mode.

setData()

Description: Sets character data from the clNdo object.

Syntax: setData(char *c*)

Parameter	Description
c	Sets character data to the clNdo object.

Return value: Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`char c;`	`char c;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(c);`	`rv = clNdo1.setData(c);`

setData()

Description

Sets short data from the clNdo object.

Syntax

setData(short *s*)

Parameter	Description
s	Sets short data to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`short s;`	`short s;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(s);`	`rv = clNdo1.setData(s);`

setData()

Description

Sets integer data from the clNdo object.

Syntax

setData(int *i*)

Parameter	Description
i	Sets integer data to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`int i;`	`int i;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(i);`	`rv = clNdo1.setData(i);`

setData()

Description

Sets long data from the clNdo object.

Syntax

setData(long *l*)

Parameter	Description
l	Sets long data to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`long l;`	`long l;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(l);`	`rv = clNdo1.setData(l);`

setData()

Description

Sets float data from the clNdo object.

Syntax

setData(float *f*)

Parameter	Description
f	Sets float data to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`char c;`	`char c;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(c);`	`rv = clNdo1.setData(c);`

setData()

Description

Sets string data from the clNdo object.

Syntax

setData(string *s*)

Parameter	Description
s	Sets string data to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`string s;`	`string s;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(s);`	`rv = clNdo1.setData(s);`

setData()

Description

Sets blob data from the clNdo object.

Syntax

setData(blob *b*)

Parameter	Description
b	Sets blob data to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`blob b;`	`blob b;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(b);`	`rv = clNdo1.setData(b);`

setData()

Description

Sets unsigned character data from the clNdo object.

Syntax

setData(char *uc*)

Parameter	Description
uc	Sets an unsigned character to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`unsigned char uc;`	`unsigned char uc;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(uc);`	`rv = clNdo1.setData(uc);`

setData()

Description

Sets unsigned short data from the clNdo object.

Syntax

setData(short *us*)

Parameter	Description
us	Sets signed short data to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`unsigned short us;`	`unsigned short us;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(us);`	`rv = clNdo1.setData(us);`

setData()

Description

Sets unsigned integer data from the clNdo object.

Syntax

setData(int ui)

Parameter	Description
ui	Sets unsigned integer data to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`unsigned integer ui;`	`unsigned integer ui;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
`.`	`.`
`.`	`.`
`rv = pClNdo1->setData(ui);`	`rv = clNdo1.setData(ui);`

setData()

Description

Sets unsigned long data from the clNdo object.

Syntax

setData(long *ul*)

Parameter	Description
ul	Sets an unsigned long to the clNdo object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`unsigned long l;`	`unsigned long l;`
`clNdo *pClNdo1;`	`clNdo clNdo1;`
.	.
.	.
`rv = pClNdo1->setData(ul);`	`rv = clNdo1.setData(ul);`

setName()

Description

Sets the name of the current NNDODataNode associated with this clNdo object.

Syntax

setName(string newName)

Parameter	Description
newName	String that contains the new name of a node.

Return value

None.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`clNdo *pClNdo0;.`	`clNdo clNdo0;`
.	.
.	.
`pClNdo0->setName("TEMPNAME");`	`clNdo0.setName("TEMPNAME");`

setSchema()

Description

Sets a schema for NDO self-describing New Era Canonical Format (NCF).

Syntax

setSchema(blob **pBlob*)

Parameter	Description
pBlob	Pointer to a blob containing a corresponding schema to a nonself-describing NCF to be serialized.

Return value

None.

Usage

Pointer to object	Object
`int rv;`	`int rv;`
`clNdo *pClNdo6;`	`clNdo clNdo6;`
`blob testBlob;`	`blob testBlob;`
`blob schemaBlob;`	`blob schemaBlob;`
`testBlob.load("rootmy.ncf");`	`testBlob.load("rootmy.ncf");`
`schemaBlob.load ("NAX8.IC.root.ncm");`	`schemaBlob.load ("NAX8.IC.root.ncm");`
`rv = pClNdo6.setSchema (&schemaBlob);`	`rv = clNdo6.setSchema (&schemaBlob);`
`rv = pClNdo6.deserializeNCFnsd (&testBlob);`	`rv = clNdo6.deserializeNCFnsd (&testBlob);`

writeNCF()

Description

Saves the contents of the clNdo into a self-describing, New Era Canonical Format (NCF) in a file.

Note This appends the NCF to an existing file of the same name.

Syntax

writeNCF(string filename)

Parameter	Description
filename	File that contains the saved clNdo in an NCF form.

Return value

Integer. Returns 1 for success, and 0 for failure.

writeNCFnsd()

Description

Saves the contents of the clNdo into a nonself-describing, New Era Canonical Format (NCF) in a file.

Note This appends the NCF to an existing file of the same name.

Syntax

writeNCFnsd(string *filename*)

Parameter	Description
filename	File that contains the saved clNdo in an NCF form.

Return value

Integer. Returns 1 for success, and 0 for failure.

writeXML()

Description

Saves the contents of the clNdo into an XML format in a file.

Note This appends the XML to an existing file of the same name.

Syntax

writeXML(string *filename*)

Parameter	Description
filename	File that contains the saved clNdo in an XML form.

Return value

Integer. Returns 1 for success, and 0 for failure.

Open Transport objects

Open Transport (clot) objects are used to implement Open Transport calls through ODL. Open Transport allows you to use different drivers to communicate with non-Impact processes through standard transport systems, including IBM MQSeries and Microsoft MSMQ.

For more information on MQSeries, see Chapter 7, “Accessing WebSphere MQ Data.”

For information about Open Transport configuration, see *Open Transport Configuration Guides* provided with this release.

addProp()

Description

Adds properties to a message. See the Syntax section.

Syntax

addProp(proptype *ptype*, string *propname*, string *propvalue*);

Parameter	Description
ptype	Type of the property to add. These include: *OT_PROP_CONTEXT* *OT_PROP_TRANSPORT* *OT_PROP_GET* *OT_PROP_PUT*
propname	Name of the property to add.
propvalue	Value of the property to add.

Return value

None.

Usage

Pointer to object	Object

`pot->addProp (OT_PROP_TRANSPORT, "NNOT_TIL_OPEN_TSI", "TestQueue");`	`string propname;`
	`propname = "NNOT_TIL_OPEN_TSI";`
	`my_otobj.addProp (OT_PROP_TRANSPORT,propname, "TestQueue");`

begin()

Description

Begins a unit of work. Provides users with an explicit way to control units of work; however, users are responsible for committing or rolling back their own work.

Syntax

begin();

Return value

Always returns 1.

Examples

```
ot1.addProp(OT_PROP_TRANSPORT,"NNOT_TIL_OPEN_TSI","Tes
tQueue");
ot1.create("putContext");
ot1.open("putTransport");
data= "test message";
ot1.setData(&Data);
ot1.begin();
ot1.put();
ot1.commit();
ot1.close();
```

close()

Description: Closes a transport.

Syntax:

```
close();
```

Return value: Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`pot->close();`	`my_otobj.close();`

commit()

Description: Commits a transaction to a transport mechanism.

Syntax:

```
commit();
```

Return value: Always returns 1.

Usage

Pointer to object	Object
`pdbi->commit();`	`my_dbi.commit();`

clearProps()

Description: Clears message properties for the given type.

Syntax:

```
clearProps(proptype ptype);
```

Parameter	Description
ptype	Property type to clear. These include: • OT_PROP_CONTEXT • OT_PROP_TRANSPORT • OT_PROP_GET • OT_PROP_PUT

Return value: None.

create()

Description: Creates a context using default properties from the *nnsyreg.dat* configuration file.

Syntax

create(string *ContextID*);

Parameter	Description
ContextID	Open Transport contextID from the *nnsyreg.dat*.

Return value

None.

Examples

```
create(ContextID)
```

debugOff()

Description

Terminates debugging for the specified object.

Syntax

debugOff();

Return value

Always returns a 1.

Usage

Pointer to object	Object
`pot->debugOff();`	`my_otobj.debugOff();`

See also

debugOn()

debugOn()

Description

Enables debugging for the specified object and logs detailed messages about the object to the *xlog* file.

Syntax

debugOn();

Return value

Always returns a 1.

Usage

Pointer to object	Object
`pot->debugOn();`	`my_otobj.debugOn();`

See also

debugOff()

deleteProp()

Description

Deletes message properties for the given type.

Syntax

deleteProp(proptype ptype, string *propname*);

Parameter	Description
ptype	Property type to delete. These include: • *OT_PROP_CONTEXT* • *OT_PROP_TRANSPORT* • *OT_PROP_GET* • *OT_PROP_PUT*
propname	Name of the property to delete.

Return value

None.

Usage

Pointer to object	Object

`pot->deleteProp (OT_PROP_TRANSPORT, "NNOT_TIL_OPEN_TSI");`	`string propname;`
	`propname = "NNOT_TIL_OPEN_TSI";`
	`my_otobj.deleteProp (OT_PROP_TRANSPORT,propname);`

dumpProps()

Description

Dumps message properties into the *xlog* file. This is useful for troubleshooting.

Syntax

dumpProps(proptype *ptype*);

Parameter	Descriptions
ptype	Property type to log into the *xlog* file. These include: • *OT_PROP_CONTEXT* • *OT_PROP_TRANSPORT* • *OT_PROP_GET* • *OT_PROP_PUT*

Return value

None.

Usage

Pointer to object	Object
`pot->dumpProps (OT_PROP_TRANSPORT);`	`my_otobj(OT_PROP_TRANSPORT);`

get()

Description

Gets a message from a transport.

Syntax

get();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`pot->get();`	`my_otobj.get();`

getData()

Description

Gets data from a message.

Syntax

getData({string | blob} **data*);

Parameter	Description
data	Pointer to the location to put the data read from the message. This can be either a string or a blob. However, if it is a string, and the data contains a null, the string terminates.

Return value

Always returns 1.

Usage

Pointer to object	Object
`string msgData;` `pot->getData(&msgData);`	`string msgData;` `my_otobj.getData(&msgData);`
`blob binData;` `pot->getData(&binData);`	`blob binData;` `my_otobj.getData(&binData);`

getProp()

Description

Gets one of the message properties from the list.

Syntax

getProp(proptype *ptype*, string *propname*, string **propvalue*);

Parameter	Description
ptype	Property type to get. These include: • *OT_PROP_CONTEXT* • *OT_PROP_TRANSPORT* • *OT_PROP_GET* • *OT_PROP_PUT*
propname	Name of the property to select.

Parameter	Description
propvalue	Pointer to the string that receives the value of the property.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object

`string myPropValue;`
`pot->getProp (OT_PROP_TRANSPORT, "NNOT_TIL_OPEN_TSI", &myPropValue);`
	`string myPropValue;`
	`string propname;`
	`propname = "NNOT_TIL_OPEN_TSI";`
	`my_otobj.getProp (OT_PROP_TRANSPORT,propname, &myPropValue);`

hasMessage()

Description

Determines if a queue currently has a message.

Syntax

hasMessage();

Return value

Integer. Returns 0 if no messages are found in the queue. Returns 1 if there is at least one message present.

Usage

Pointer to object	Object

`int has_message;`
`has_message = pot->hasMessage();`
`if (has_message = 1)`
`{`
`// process the message`
`}`
`else`
`{// handling for no message`
`}`
	`int has_message;`
	`has_message = my_otobj.hasMessage();`
	`if (has_message = 1)`
	`{`
	`// process the message`
	`}`
	`else`
	`{// handling for no message`
	`}`

open()

Description

Opens the transport associated with *TransportID*.

Syntax

int open(string *TransportID*);

Parameter	Description
TransportID	String corresponding to a transport configured in the *nnsyreg.dat*. This can be a string variable or a literal string.

Return value

Integer. Returns 1 for success, and 0 for failure.

Examples

```
my_otobj.open(char *TransportID)
//open transport using transport id in nnsyreg.dat
```

put()

Description

Puts a message into a transport.

Syntax

put();

Return value

Integer. Returns 1 for success, and 0 for failure.

Examples

```
ot1.addProp(OT_PROP_TRANSPORT,"NNOT_TIL_OPEN_TSI","Tes
tQueue");
ot1.create("putContext");
ot1.open("putTransport");
data= "test message";
ot1.setData(&Data);
ot1.begin();
ot1.put();
ot1.commit();
ot1.close();
```

rollback()

Description

Executes a rollback that removes any changes in the transport made by the OT object.

Syntax

rollback();

Return value

Always returns 1.

Usage

Pointer to object	Object
`pdbi->rollback();`	`my_dbi.rollback();`

setData()

Description

Sets the message data for the next put operation.

Syntax

int setData(string *data*);

int setData(string **data*);

int setData(blob **data*);

Parameter	Description
data	Data to put in the Open Transport message buffer. This can be a string, a pointer to a string, or a pointer to a blob.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`pot->setData ("this is some data");`	`my_otobj.setData ("this is some data") blob blobData;`
`string strData; strData = "some data"; pot->setData(&strData);`	`blobData.load("datafile.dat"); my_otobj.setData(&blobData);`

String objects

String objects are null-terminated linear collection of printable characters. To define a string object, use the form:

```
string obj_name;
```

You can use C language operators on strings, as shown below:

- Assignment –

```
a = b;
```

- Concatenation –

```
a + b;
a +=b;
```

- Equality check –

```
if (a ==b);
```

```
if (a != b);
```

- True/false –

```
if (a);
if(!a);
```

Note Because strings are null terminated, when converting from a blob to a string, if the blob contains a null character, ODL copies into the string only the contents of the blob up to that null character

---

Use these methods to manipulate string object data, or use the following arithmetic and comparison operators: =, +, ++, !+, <, >, <=, and >=. Adding control characters to a string must be done using the format method or by converting the string to a blob and appending the control character.

You can access individual bytes of string object data by indexing into the string object. String objects have a zero-based index. Do not subscript past the size of the string:

```
char ch;
string stuff;
stuff = "this is a test.";
ch = stuff[3];
//ch now contains "s"
```

You can truncate a string by assigning a null to one of the data bytes in the string object:

```
string stuff;
stuff = "some data, and other data you don't need";
stuff[9] = '\000':
```

You can also use a pointer to the string object:

```
string *pb;
char first_char;
first_char = (*pb) [0];
```

Note If you use a pointer to a string object, use the format shown in the Usage section under the table heading "Pointer to object."

---

cat()

Description

Gets data from the source, starting at the offset for a specified length, and appends it to the string. The first character in the source is always at an offset of zero. The *offset* parameter is optional. If an *offset* is not given, copying starts with the first character in source and continues for the specified length.

Syntax

cat(string *source*, int *length*, int *offset*)

Parameter	Description
source	Source of the data to copy. Can be another string object or a literal, enclosed in double quotes.
length	The number of characters of data to copy.
offset (optional)	Location from which to begin copying data. The first character in a string or literal is always at offset zero.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`ps->cat("abc123", 3, 3);`	`mystring.cat("def456", 3);`
`ps->cat(my_string_var,5, 2);`	`mystring.cat(my_string_var, 10);`

Examples

```
string mystr;
mystr = "abcdefg";
mystr.cat("123456789", 3, 4);
//mystr now contains "abcdefg567"
```

copy()

Description

Copies data from the source, starting at the offset for a specified length, and replaces the data in the string with the copied data. The first character in the source is always at an offset of zero. The *offset* parameter is optional. If an *offset* if not given, copying starts with the first character in the source and continues for the specified length.

Syntax

copy(string *source*, int *length*, int *offset*);

Parameter	Description
source	Source of the data to copy. Can be another string object or a literal, enclosed in double quotes.

Parameter	Description
length	The number of characters of data to copy.
offset (optional)	Location from which to begin copying data. The first character in a string or literal is always at offset zero.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`ps->copy("abc123",3,3);`	`mystring.copy("def456",3,1);`
`ps->copy(my_string_var, 5);`	`mystring.copy(my_string_var, 10);`

Examples

```
string mystr;
string new_stuff;
mystr = "abcdefg";
new_stuff = "123456789";
mystr.copy(new_stuff, 5, 2);
// mystr now contains "34567"
```

debug()

Description

Used by developers for debugging purposes. This method causes the string object to dump its contents into an *xlog* file in hexadecimal and standard formats. This file resides in the current working directory, usually the directory from which the user runs the AIM. Due to the time it takes to dump the contents of a string object, especially if it has a lot of data, use this method with caution, and do not include it in a production AIM.

Syntax

debug();

Return value

Integer. Always returns 1.

Usage

Pointer to object	Object
`ps->debug();`	`my_string.debug();`

format()

Description

Mimics the C sprintf API, putting the data into the string. The arguments are the format string and the arguments are listed in the format string. *printform* uses the same formatting as the C sprintf API.

Syntax

format(string *printform*, *args*);

Parameter	Description
printform	Print formatting.This string should be in the format of C language sprintf strings, with parameter characters preceded by a percentage character (%).
args	Any arguments called for in the *printform*. If the character is a control character, it must be in either local or hexidecimal format.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`ps->format ("%02d times %01d = %i",a,3.5, a*3.5);`	`mystring.format ("Hello, %s,name);`
`The object would then hold "3.66 times 3.5 = 12.81"`	`mystring would then hold "Hello, Ed"`

Examples

The format string has 3 arguments: a decimal with 2 places, a decimal with 1 place, and an unlimited decimal. The object would hold "3.66 times 3.5 = 12.81".

```
mystring.format ("%.2d times %.1d = %i", a,3.5, a*3.50);
```

log()

Description

Dumps the contents of the string object into an *xlog* file, using a standard format.

Syntax

log();

Return value

Integer. Always returns 1.

Usage

Pointer to object	Object
`ps->log();`	`mystring.log();`

Examples

```
string mystr;
int rv;
rv =some_function_call (Argument1, Argument2..);
mystr = "Return value from myDfcCommand = " +rv;
mystr.log();
```

offsetCat()

Description

Copies data from the source, starting at *offset*, and appends it to the string object. The first character in the source is always at an offset of zero.

Syntax

offsetCat(string *source*, int *offset*);

Parameter	Description
source	Source of the data to copy. Can be another string object or a literal, enclosed in double-quotes.
offset	Location to begin copying. The first character in a string or literal is always at offset 0.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`ps->offsetCat("123abc",3);`	`mystring.offsetCat (my_string_var,5);`

Examples

```
string mystr;
mystr = "abcdefg";
mystr.offsetCat("123456789",5);
//mystr now contains "abcdefg6789"
```

offsetCopy()

Description

Replaces data in the string object with data copied from a *source*, starting at the *offset*. The first character in the source is always at an offset of zero.

Syntax

offsetCopy(string *source*, int *offset*);

Parameter	Description
source	Source from which to copy the data. This can be another string object or a literal, enclosed in double quotes.
offset	Location from which to begin copying data. The first character in a string or literal is always at offset zero.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`ps->offsetCopy("123abc",3);`	`mystring.offsetCopy (my_string_var, 5);`

Examples

```
string mystr;
string new_stuff;
mystr = "abcdefg";
new_stuff = "123456789";
mystr.offsetCopy(new_stuff, 3);
// mystr now contains "456789"
```

size()

description

Calculates the number of characters of data in a string.

Syntax

size();

Return value

Integer. Returns the number of characters in the string (0 if empty), if successful, and a negative number, if failed.

Usage

Pointer to object	Object
`int count;`	`int count;`
`count = ps->size();`	`count = mystring.size();`

Examples

```
string mystr;
int num;
mystr = "abcdefg";
num = mystr.size();
//num is now 7, the number of characters in mystr
```

strchr()

Description

Searches, starting on the left of a string, for the first occurrence of *ch*. This method is case-sensitive.

Syntax

strchr(char *ch*);

Parameter	Description
ch	The character for which to search.

Return value

Integer. Returns the index position of the character, if successful, and -1, if failed.

Usage

Pointer to object	Object
`char g;`	`char g;`
`g = 'A';`	`g = 'A';`
`ps->strchr(g);`	`mystring strchr(g);`

strlft()

Description

Truncates a string from the left, up to and including the first occurrence of *pattern*. This method is case-sensitive.

Syntax

strlft(string *pattern*);

Parameter	Description
pattern	Pattern for which to search. Can be a literal value or another string object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`string digest;`	`string digest;`
`digest = "3RR56";`	`digest = "Abc-334";`
`ps->strlft (digest);`	`mystring.strlft (digest);`
`ps->strlft ("some literal value");`	`mystring.strlft ("PR-01");`

Examples

```
string mystr;
mystr = "hello/goodbye";
mystr.strlft("/");
//mystr now contains "goodbye"
```

strrchr()

Description

Searches for the first occurrence of *ch*, starting on the right of the string. This method is case-sensitive.

Syntax

strrchr(char *ch*);

Parameter	Description
ch	The character for which to search.

Return value

Integer. Returns the index position of the character, if successful, and -1, if failed.

Usage

Pointer to object	Object
`char g;`	`char g;`
`g = 'A';`	`g = 'A';`
`ps->strrchr(g);`	`mystring.strrchr(g);`

strrtt()

Description

Truncates the string, starting from the right. This method is case-sensitive.

Syntax

strrtt(string *pattern*);

Parameter	Description
pattern	Pattern for which to search. Can be a literal value or another string object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`string digest;`	`string digest;`
`digest = "3RR56";`	`digest = "Abc-334";`
`ps->strrtt (digest);`	`mystring.strrtt (digest);`
`ps->strrtt ("some literal value");`	`mystring.strrtt ("PR-01");`

Examples

```
string mystr;
string pattern;
mystr = "hello world/goodbye world";
pattern = "world";
mystr.strrtt(pattern);
//mystr now contains "hello "
```

strrem()

Description

Removes the first occurrence of the *pattern* from the string. This method is case-sensitive.

Syntax

strrem(string *pattern*);

Parameter	Description
pattern	Pattern for which to search. Can be a literal value or another string object.

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`string digest;`	`string digest;`
`digest = "3RR56";`	`digest = "Abc-334";`
`ps->strrem (digest);`	`mystring.strrem (digest);`
`ps->strrem ("a literal value");`	`mystring.strrem ("PR-01");`

Examples

```
string mystr;
mystr = "hello sir/goodbye sir"
mystr.strrem("sir");
//mystr now contains "hello /goodbye sir"
```

substr()

Description

Searches the string for pattern and returns the ordinal position of the first occurrence of pattern. The first character in a string is always at an ordinal position of 1. This method is case-sensitive.

Syntax

substr(string *pattern*);

Parameter	Description
pattern	Pattern for which to search. Can be a literal value or another string object.

Return value

Integer. Returns the ordinal position of the first occurrence of pattern, if successful, and 0, if failed.

Usage

Pointer to object	Object
`int location;`	`int location;`
`pattern="Something";`	`pattern="1234";`
`location = ps->substr (pattern);`	`location = mystring.substr (pattern);`

toLower()

Description

Changes the letters in a string to lowercase.

Syntax

toLower();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`ps->toLower();`	`mystring.toLower();`

Examples

```
string mystr;
mystr = "ABCDEFG";
mystr.toLower();
//mystr is now "abcdefg"
```

toUpper()

Description

Changes the letters in a string to uppercase.

Syntax

toUpper();

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`ps->toUpper();`	`mystring.toUpper();`

Examples

```
string mystr;
mystr = "abcdefg";
mystr.toUpper();
//mystr is now "ABCDEFG"
```

Timer objects

Timer objects (clTimer) enable events to occur at timed intervals. Use this method to implement a timeout, or a periodic event, such as polling a directory for files to process. Use these methods in a callback function to start the timer, adjust its interval setting, or to stop the timer.

kill()

Description

Shuts down a timer.

Syntax

int kill()

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`timerPtr->kill();`	`timer.kill();`

set()

Description

Sets or changes a timer *interval* and starts the timer.

Syntax

int set(int *internal*);

int set(long *interval*);

Parameter	Description
interval	The timer *interval*. Use an integer to set the time in seconds. Use a long datatype to set the time in milliseconds

Return value

Integer. Returns 1 for success, and 0 for failure.

Usage

Pointer to object	Object
`int seconds;`	`int seconds;`
`ptimer->set(seconds);`	`my_timer.set(seconds);`
`long m_seconds;`	`long m_seconds;`
`ptimer->set (m seconds);`	`my_timer.set(m_seconds);`

Examples

```
ptimer1->set(0);
//in milliseconds:
long m_sec;
m_sec = 500;
ptimer1->set(m_sec);
//in seconds:
int sec1;
sec1 = 2;
ptimer1->set(sec1);
```

CHAPTER 6

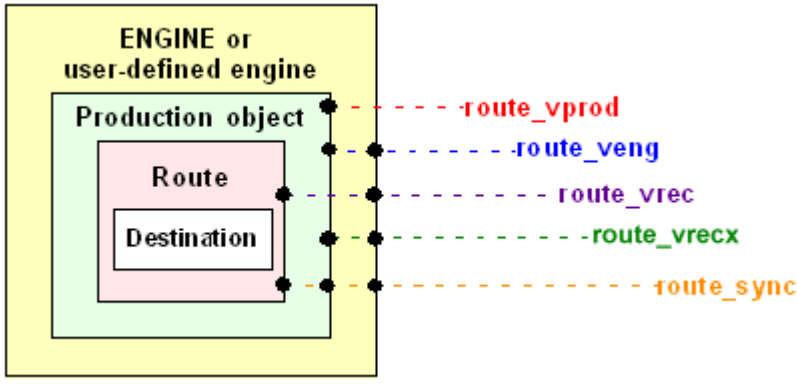
Route Calls

This chapter describes the distributed function calls (DFCs) used to send transactions to an SFM application.

Introduction

Acquisition AIMs uses a special distributed function call (DFC) to send a transaction to the Store and Forward Manager (SFM). DFC commands, also called “routing commands” or “routing functions,” consist of arguments that allow you to send the transaction to specific areas of the SFM, such as an engine (a user-defined group of production objects), one production object, all production objects (“ENGINE”), or a route.

Figure 6-1: SFM routing entry points



SFM transaction processing

The SFM places each transaction it receives into one of its log files. For the transaction to be accepted and written into the transaction log file, a transaction must pass certain tests. If it does not pass these tests, the SFM writes the transaction to its unrouteable log file.

Note The user who has access to the unrouteable log file through TRAN-IDE can correct these transactions and re-submit them to the SFM for processing.

The SFM saves transactions that it cannot post to destinations in the transaction log file. If one or more production objects process a transaction before it goes to its destinations, the SFM saves the original transaction until the destination is available, then processes the transaction through its production objects and sends it on.

When the SFM receives notification that a transaction in the transaction log file has been received and processed by its destination delivery AIM, the SFM marks the transaction as processed. If the incoming transaction activity is high enough so that all of the file's space is used up, the SFM wraps to the beginning of the file and reuses the space used by processed transactions.

The SFM uses a simple transaction ID or production ID value to route a transaction through the SFM distributed computing environment. You must assign an ID value to each unique transaction record structure.

A production ID value is the name of a production object generated with TRAN-IDE. You may use one of the routing functions described in this chapter to send a transaction directly for processing by a single production object. In this case, SFM does not check the transaction to see if it qualifies for any other production objects.

Routing DFCs

The available routing DFCs are:

- route_vprod() – routes data to the SFM through a single production object. See "route_vprod()" on page 155.
- route_veng() – sends data to the SFM through a specific engine. An engine contains a specific set of production objects listed in a specific order. See "route_veng()" on page 156.

- route_vrec() – sends data to the SFM through a specific route. This operation bypasses any production objects. The route_vrec() function also allows you to send the transaction to all productions object when the reserved keyword “ENGINE” is used. See “route_vrec()” on page 158.
- route_recx() – sends data to the SFM through an “ENGINE” (all production objects), or through an engine, production object, or a route. Arguments passed to the function determine which activity is performed. See “route_recx()” on page 160.
- route_sync() – sends data to a router application through a route or through all production objects when “ENGINE” is specified. The route_sync() function is used by an MQAcq acquisition AIM and propagates out-of-bounds information growing from each message in a queue. See “route_sync()” on page 164.

Table 6-1: Routing function summary

Routing function	A route	A production objection	An engine	All production objects using “ENGINE”
route_vprod()		X		
route_veng()			X	X
route_vrec()	X			X
route_recx()		X	X	X
route_sync()	X	X	X	X

Transactions using a standard ID

Transactions with a standard transaction ID value that do not refer to production objects (sent with the route_vrec() or route_recx()), are processed as follows:

1 The SFM checks its configuration file for a corresponding transaction ID record, and routes the transaction to one or more delivery AIMS as identified in the record.

2 If the SFM does not find a transaction ID record, it writes the data into its unrouteable log file, and returns a value of -2 to the acquisition AIM.

Transactions using a production object ID

Transactions using a production ID value (sent with route_vprod() or route_recx()), are processed using these steps:

1 Transactions must parse correctly according to the field object definitions of the specified production object.

2 If one or more of the field objects has specific datatypes defined, the transaction's data that parses into those fields must match the datatype.

3 If one or more of the field objects has a qualification object attached to it, the transaction's data must successfully pass the qualification criteria.

4 If a production object that will process the transaction has one or more qualification objects attached to it, the transaction's data must successfully pass the qualification criteria.

5 If the transaction passes all of the tests in steps 2 through 4 that apply to it, the SFM writes it to the transaction log file, and returns a value of 1 to the acquisition AIM, to indicate that it received the transaction successfully.

   If the transaction does not pass the tests, the SFM writes the transaction to its unrouteable log file, and returns a value of -6 to the acquisition AIM.

Transaction using "ENGINE" or user-defined engines

Transactions with a value of "ENGINE" (sent using route_vrec() or route_vrecx()), or with a user-defined engine name (sent with a route_recx()), are processed using this procedure:

- The same process occurs as when the transaction ID is set to a single production object ("Transactions using a production object ID" on page 153), except that now the transaction is tested against each of the production objects specified in the user-defined engine, or against all productions objects (using "ENGINE").
- If the transaction passes all of the relevant tests for one or more of the production objects, SFM writes it to the transaction log file, and returns a value of 1 to the acquisition AIM.

- If the transaction does not pass any of the relevant tests for any production objects, the SFM writes the transaction to its unrouteable log file, and returns a value of -6 to the acquisition AIM.

Note All production objects must have a production record in the SFM configuration file. See the *e-Biz Impact Configuration Guide* for details.

route_vprod()

Description

Use route_vprod() to send a transaction to the SFM and route it directly to a single production object before sending the transaction to a delivery AIM.

When the SFM receives the transaction, it:

- Checks the transaction against the production object's field objects (that is, does the transaction match these definitions in field sizes and datatypes?);
- Runs any qualification objects attached to the field objects against the transaction data, then, if required;
- Runs the production object's qualification objects against the transaction data.

If the transaction passes all these tests, the SFM writes it to the transaction log file. If the transaction does not pass any one of these tests, the SFM writes the transaction to the unrouteable log file and returns a value of -6 to the acquisition AIM.

The acquisition AIM is in charge of responding to the endpoint if routing succeeded or failed by composing an acknowledgement or negative ACK and sending the peer.

After the SFM writes the transaction into its log file, it returns a value of 1 to the acquisition AIM to indicate that the transaction was accepted. The acquisition AIM can continue to gather data without waiting for the delivery AIMs to finish transaction processing.

Format

```
route_vprod(int [flavor] Flavor, string [in] SRCRef,
   string [in] ProdName, {string | blob} [in] Data);
```

Parameter	Description
Flavor	The flavor assigned to the SFM module that should receive the data. If you are running only one copy of the SFM, use a value of zero in this argument.

Parameter	Description
SRCRef	The Source Reference Name. A user-supplied reference name for the current Acquisition AIM. Allowable characters include A – Z, a – z, 0 – 9. The string datatype is required.
ProdName	The name of a production object that should process the data. The name can be a maximum of 32 characters plus a null character. Allowable characters include A – Z, a – z, 0 – 9. A String or blob datatype is required.
Data	A blob or string field that contains the transaction's data.

Return values Integer.

Value	Description
> 0	The SFM has accepted the record and has written it to the transaction log file.
0	General error. Check the error log and the standard UNIX "errno" field.
-1	The transaction could not be passed to the SFM. Your program should check the unrouteable error log to find out why this occurred.
-2	The SFM could not find the name of the production object in its configuration file.
-3	The SFM could not process the transaction because the log file was full.
-4	The SFM received a transaction larger than the log file's transaction size.
-5	The SFM refused the record because it is in Refuse mode.
-6	The transaction data did not pass the field object validation or production object qualification checks in the referenced production object.
-7	There are no valid destinations for the production object for which the transaction qualified. Occurs when the production object uses the setDestName and/or the setDestNameData dynamic routing built-ins to override the production object's configured destinations, but the built-ins do not specify valid destinations.
-8	The SFM could not process the transaction because an invalid destination was specified with a dynamic routing built-in filter function.

route_veng()

Description Use route_veng() to send transactions to the SFM for transaction production through a specified engine.

Note "ENGINE" is all defined production objects defined (use reserved or named declared in the SFM configuration), but an "engine" contains a subset of production objects. If you define "engine to contain all production objects, then it is the same as "ENGINE."

Transactions routed by an engine qualify against all the production objects specified for this engine in the order they have been declared. The transaction is considered qualified if it qualifies for one or more production objects, otherwise, the transaction is considered unrouteable.

Note To use the same functionality as route_veng(), but not have the transaction written to the unrouteable log if the transaction is unrouteable (for example, does not qualify, or has a bad route), use route_veng2() instead.

Format

```
int route_veng(int [flavor] Flavor, string [in] SRCRef,
string [in] engName, {string | blob} [in] Data);
```

Parameter	Description
Flavor	The flavor assigned to the SFM module that should receive the data. If you are running only one copy of the SFM, use a value of zero in this argument.
SRCRef	The Source Reference Name. A user-supplied reference name for the current Acquisition AIM. Allowable characters include A – Z, a – z, 0 – 9. The string datatype is required.
engName	the names of the production objects that should process the data. The names can be a maximum of 32 characters plus a null character. Allowable characters include A – Z, a – z, 0 – 9. A String or blob datatype is required
Data	A blob or string field that contains the transaction's data.

Return values

Integer.

Value	Description
> 0	The SFM has accepted the record and has written it to the transaction log file.
0	General error. Check the error log and the standard UNIX "errno" field.
-1	A transaction could not be passed to the SFM. Your program should check the unrouteable error log to find out why this occurred.
-2	The SFM could not find the names of the production objects in its configuration file.
-3	The SFM could not process the transaction because the log file was full.
-4	The SFM received a transaction larger than the log file's transaction size.
-5	The SFM refused the record because it is in Refuse mode.
-6	The transaction data did not pass the field object validation or production object qualification checks in the referenced production objects.
-7	There are no valid destinations for the production objects for which the transaction qualified. Occurs when the production objects use the setDestName and/or the setDestNameData dynamic routing built-ins to override the production object's configured destinations, but the built-ins do not specify valid destinations.
-8	The SFM could not process the transaction because an invalid destination was specified with a dynamic routing built-in filter function.

route_vrec()

Description

Use route_vrec() to send data to the SFM and to a route, or to send data through all production objects when the reserved keyword “ENGINE” is used.

If the transaction ID argument (*TranID*) has a value of “ENGINE:”

1 The SFM checks the transaction’s form and content against all production objects available to it:

   a The form of the data must match at least one field object set available to the current SFM.

   b The data must match the datatypes assigned to the field object definitions (for example, a field defined as a numeric datatype may not include alphabetic characters).

   c The field data must pass any non-optional qualification objects attached to the field objects.

   d The field data must pass any non-optional qualification objects attached to the production object.

2 If the transaction passes these tests for one or more production objects, the SFM writes it to the transaction log file and returns a value of 1 to the acquisition AIM to indicate that it received the transaction successfully.

   If the transaction does not pass any of these tests, the SFM writes the transaction to its unrouteable log file and returns a value of -6 to the acquisition AIM.

If the transaction ID argument (*TranID*) has a value other than “ENGINE,” the SFM checks its configuration for a corresponding route, and routes the transaction to delivery AIMs identified by the route. If the SFM does not find a route record, it writes the data into its unrouteable log file, and returns a value of -2 to the acquisition AIM.

Format

```
route_vrec(int [flavor] Flavor, string [in] SRCRef,
string [in] TranID, {string | blob} [in] Data);
```

Parameter	Description
Flavor	The flavor assigned to the SFM module that should receive the data. If you are running only one copy of the SFM, use a value of zero in this argument.
SRCRef	The Source Reference Name. A user-supplied reference name for the current Acquisition AIM. Allowable characters include A – Z, a – z, 0 – 9. The string datatype is required.

Parameter	Description
TranID	A transaction ID reference for the particular record type carried in the current *Data* argument. The SFM uses this transaction ID to route the transaction to the correct delivery AIM. A value of “ENGINE” routes the data through all available production objects. A custom engine allows you to group multiple production objects into one engine routing package. **Note** To create a custom engine, open the SFM properties window in the Configurator and select the Routing tab. Click New in the Engines section.The SFM Engine Info dialog box appears.Enter a user-defined engine name in the Engines Add field. In the ProdObjs box, select the production objects you want to include, then click the right arrows (>>) button to move the select objects to the Grouped ProdObjs box. Click Add. The engine name appears in the Engine section. Click OK to save your entries and close the window.
Data	A blob or string field that contains the transaction’s data.

Return values Integer.

Value	Description
> 0	The SFM has accepted the record and has written it to the transaction log file.
0	General error. Check the error log and the standard UNIX “errno” field.
-1	A transaction could not be passed to the SFM. Your program should check the unrouteable error log to find out why this occurred.
-2	The SFM could not find the names of the route (also known as the transaction ID) in its configuration file.
-3	The SFM could not process the transaction because the log file was full.
-4	The SFM received a transaction larger than the log file’s transaction size.
-5	The SFM refused the record because it is in Refuse mode.
-6	The transaction data did not pass the field object validation or production object qualification checks in the referenced production objects.
-7	There are no valid destinations for the production objects for which the transaction qualified. Occurs when the production objects use the setDestName and/or the setDestNameData dynamic routing built-ins to override the production object’s configured destinations, but the built-ins do not specify valid destinations.
-8	The SFM could not process the transaction because an invalid destination was specified with a dynamic routing built-in filter function.

route_recx()

Use route_recx() to send a transaction to the SFM, through a specific production object, or through all the production objects currently available to the SFM when “ENGINE” is specified.

If the *Opts* argument does not contain the RO_BYPRODNAME mnemonic and the Transaction ID argument has a value other than “ENGINE,” the SFM checks its configuration file for a corresponding route, and routes the transaction to one or more delivery AIMs. If the SFM does not find a route in its configuration file, it writes the data into its unrouteable log file, and returns a value of -2 to the acquisition AIM.

If the *Opts* argument does not contain the RO_BYPRODNAME mnemonic and the *TranID* argument has a value of “ENGINE,” the SFM sends the input transaction through transaction production. Transaction production checks the transaction’s form and content against all production objects available to it.

If the transaction ID argument (*TranID*) has a value of “ENGINE:”

1 The SFM checks the transaction’s form and content against all production objects available to it:

   a The form of the data must match at least one field object set available to the current SFM.

   b The data must match the datatypes assigned to the field object definitions (for example, a field defined as a numeric datatype may not include alphabetic characters).

   c The field data must pass any non-optional qualification objects attached to the field objects.

   d The field data must pass any non-optional qualification objects attached to the production object.

2 If the transaction passes these tests for one or more production objects, the SFM writes it to the transaction log file and returns a value of 1 to the acquisition AIM to indicate that it received the transaction successfully.

   If the transaction does not pass any of these tests, the SFM writes the transaction to its unrouteable log file and returns a value of -6 to the acquisition AIM.

If the *Opts* argument contains the RO_BYPRODNAME mnemonic, the SFM interprets the *TranID* argument as a production object name and sends the transaction data on to the specified production object. If the transaction passes the same transaction production tests described previously, the SFM writes the transaction to the transaction log file and returns a value of 1 to the acquisition AIM to indicate that the transaction was received successfully. If the transaction does not pass any of these tests, the SFM writes the transaction to its unrouteable log file and returns a value of -6 to the acquisition AIM.

Format

```
route_recx(int [flavor] Flavor, string [in] SRCRef,
string [in] TranID, blob [in] *Data, int [in] Opts, char
[in] Priority, blob [in] Fkey, blob [out] *Serial, blob
[out] *ErrText, int [out] *Stat, blob [out] *Dests, blob
[out] *SfmName, int [out] *SfmFlav {string | blob} [out]
*hostname);
```

Note To use one of the optional arguments, you must define all of the other optional arguments that come before it so that the SFM can correctly parse the function. However, you can assign null values to the optional arguments that precede the argument you want to use.

Parameter	Description
Flavor	The flavor assigned to the SFM module that should receive the data.
SRCRef	The Source Reference Name. A user-supplied reference name for the current Acquisition AIM. Allowable characters include A – Z, a – z, 0 – 9. The string datatype is required.
TranID	A transaction ID reference for the particular record type carried in the current *Data* argument. The SFM uses this transaction ID to route the transaction to the correct delivery AIM. A value of “ENGINE” routes the data through all available production objects. A custom engine allows you to group multiple production objects into one engine routing package. **Note** To create a custom engine, open the SFM properties window in the Configurator and select the Routing tab. Click New in the Engines section.The SFM Engine Info dialog box appears.Enter a user-defined engine name in the Engines Add field. In the ProdObjs box, select the production objects you want to include, then click the right arrows (>>) button to move the select objects to the Grouped ProdObjs box. Click Add. The engine name appears in the Engine section. Click OK to save your entries and close the window.
Data	A string or blob field that contains the transaction’s data.

Parameter	Description
Opts	Optional. This argument is a bit mask. Use one or more of the following mnemonics to specify the options to set. Separate each mnemonic with a pipe symbol (|). • Leave this argument blank to have the SFM interpret the *TranID* argument as a transaction ID (like route_vrec()). • RO_BYPRODNAME – have the SFM interpret the *TranID* argument as the name of a production object (like route_vprod()). • RO_BYENGINENAME – have the SFM interpret the *Tranid* argument as an engine name and route the transaction by engine. • RO_SERIALDISPATCH – have the SFM serially dispatch the transaction among its intended destinations. • RO_MUSTBEUP – have the SFM refuse the transaction if all of the destinations for the transaction are not up. If any of the delivery AIMs specified for this transaction are not currently active, then the route_recx() command returns with a value of -10. • RO_MUSTBEPOLLED – have the SFM query the transaction's destinations with a servayt() call and refuse the transaction if all of the delivery AIMs for the transaction are not up. If any of the delivery AIMs specified for this transaction are not currently active, then the route_recx() command returns with a value of -10. • QO_QUALFORONE – when the *TranID* is "ENGINE," use this value to have the SFM stop submitting the transaction to production objects once the transaction has qualified for one production object. Use this option only when you are certain that the transaction will qualify for only one production object. If it qualifies for more than one production object, depending upon the order that the SFM presents the transaction to the production objects, the transaction may never get to the production object you want it to reach.
Priority	Optional. Place a number from 0 through 255 in a character argument to assign a priority to the transaction, where 1 is the lowest priority and 255 the highest.Using a priority of 0 removes a priority set in the transaction ID record. You may want to limit priorities to the 0 through 16 range, as the 17 through 255 range are reserved for future use. Any priority assigned in this argument overrides the priority assigned for the transaction in the transaction ID record in the SFM configuration. Whenever the SFM receives a transaction that has a priority set, it processes that transaction before any non-prioritized transactions that are in the log file. When more than one prioritized transaction is waiting for processing, the SFM processes transactions from the highest priority to the lowest. If multiple transactions have the same priority, the SFM processes them based on their timestamp. You can also set or change a transaction's priority with the tranPriority built-in filter function in the Post-Qualify Rule in a production object. A priority set with the tranPriority built-in function overrides the priority assigned in this argument. **Note** When the SFM receives a transaction that has a priority set, it ignores all blocking and serialization options and processes the prioritized transaction first.

Parameter	Description
Fkey	Optional. A user-supplied foreign key that endpoints can use to further identify a transaction. The SFM uses this value in the "Once and Once Only" feature in TRAN-IDE to prevent servicing duplicate transactions. The SFM does not change the value of this argument. See the *e-Biz Impact TRAN-IDE Guide* for more information.
Serial	Optional. The serial number assigned by the SFM to this transaction. The SFM passes this value back to the acquisition AIM when returning from the route_recx() function.
ErrText	Optional. The error text generated by the SFM and/or the production object. The SFM passes this value back to the acquisition AIM if the route_recx() function fails.
Stat	Optional. The status of the transaction. The SFM passes this value back to the acquisition AIM when returning from the route_recx() function with a value greater than 0. Possible statuses are: • `RT_QUEUED` – the transaction is queued for processing. • `RT_SENT` – the transaction is being processed.
Dests	Optional. The name of the destinations to receive the transaction.
SfmName	Optional. The name of the SFM that serviced the route_recx() function.
SfmFlavor	Optional. The flavor of the SFM that serviced the route_recx() function.
hostname	This argument is optional and is used to return the host name of the SFM servicing this transaction to the acquisition AIM.

Return values Integer.

Value	Description
> 0	The SFM accepted the record and has written it to the transaction log file.
0	General error. Check for DFC errors and the standard UNIX "errno" field.
-1	e-Biz Impact could not pass the transaction on to the SFM. Your program should check for DFC errors to find the problem.
-2	The SFM could not find the route (the transaction ID) in its configuration.
-3	The SFM could not process the transaction because the log file was full.
-5	The SFM refused the record because it is in Refuse mode.
-6	The transaction ID was "ENGINE" and transaction data did not pass form and content validation for the production objects available to the SFM.
-7	There are no valid destinations for the production objects for which the transaction qualified. Occurs when the production objects uses the setDestName() and/or the setDestNameData() dynamic routing built-ins to override the production object's configured destinations but the built-ins do not specify valid destinations.
-8	The SFM could not process the transaction because an invalid destination was specified with a dynamic routing built-in filter function.
-9	The SFM could not append file contents to the transaction.
-10	The *Opts* argument contains `RO_MUSTBEUP` or `RO_MUSTBEPOLLED` and one of the delivery AIMs for the transaction is down.

Example

```
// format for the route_recx() functionc[idempotent]
int route_recx (int [flavor] flv,string [in] src_ref,
string [in] tranid,string [in] *databuf,int [in] opts,
char [in] pri,string [in] fkey,string [out] *serial,
string [out] *errs,int [out] *status,string [out]
*dests,string [out] *sfmname,int [out] *sfmflav,);
```

route_sync()

Description

Use route_sync() to have an MQAcq acquisition AIM pass data to the router for qualification, translation, and dispatch. The *type* parameter specifies the route behavior of this DFC.

- If *type* equals 0 (zero), this call behaves the same as route_vrec().
- If *type* equals 1, this call behaves the same as route_vprod().
- If *type* equals 2, this call behaves the same as route_veng().

  The clmap object points to the clmap object with the message descriptor field values.

Format

```
int route_sync (int [flavor] Flavor, string [in] Src,
string [in] TranId, blob [in] *Data, int [in] type,
clmap [in] *forward_opts);
```

Return value

Return values identical to the return codes of route_vrec(), route_vprod(), and route_veng().

CHAPTER 7

Accessing WebSphere MQ Data

This chapter describes how e-Biz Impact supports using transport objects to access data from transport applications supported by Open Transport drivers.

Introduction

e-Biz Impact is Open Transport enabled, which means you can use the transport objects to access data from transport applications supported by Open Transport drivers. The IBM WebSphere MQ (formerly MQSeries) driver for Open Transport maps message properties to and from message descriptor fields. The driver translates message descriptor fields from their native datatypes to a string representation, because a string representation promotes usability by graphical interfaces and command line tools. The driver translates binary message descriptor fields into string-encoded hexadecimal values such as “12ABC...” The driver also converts long message descriptor fields into string encoded long values such as “1000.”

The data header for MQSeries Integrator 1.1 or 2.0.2 includes an MQSeries message descriptor (MQMD). The MQMD structure contains the control information that accompanies the application data when a message travels between the sending and receiving applications. The data in the MQMD is necessary to correctly process a message with MQSeries Integrator.

Note For more information about the MQMD and an explanation of the fields in the MQMD, see the *IBM Manual MQSeries Application Programming Reference*.

You must complete the following two steps to enable e-Biz Impact to access MQSeries data:

- Configure *nnsyreg.dat* to define transport properties. See the Open Transport documentation include on the e-Biz Impact SyBooks CD for more detailed information about configuring *nnsyreg.dat*.
- Develop an ODL application, similar to the example provided, that uses the Open Transport clot object to define Open Transport instances. These instances can then be defined to access desired transport data.

Configuration

Use *nnsyreg.dat* to configure Open Transport context, transport, session, and transaction manager properties. The following code is an example configuration of the *nnsyreg.dat* file.

```
OTContext.CLOTOBJ_CON
    NNOT_CTX_DEFAULT_TIL_ID                = CLOTOBJ_TRAN
    NNOT_CTX_TMID                          = QMCLOTOBJ_TM
    NNOT_CTX_ENFORCE_TX                    = TRUE
Transport.CLOTOBJ_TRAN
    NNOT_SHARED_LIBRARY                    = db226mqs
    NNOT_FACTORY_FUNCTION                  = NNMQSQueueFactory
    NNOT_TIL_OPEN_SESSION_ID               = QMCLOTOBJ_SES
    NNOT_TIL_PUT_MSG_PROTOCOL              = NNOT_HDR_IF_NEEDED
    NNOT_TIL_OPEN_TSI                      = CLOTOBJ24
Session.QMCLOTOBJ_SES
    NNOT_SHARED_LIBRARY                    = dbt26mqs
    NNOT_FACTORY_FUNCTION                  = NNMQSSessionFactory
    NNMQS_SES_OPEN_MGR                     = QMCLOTOBJ
TransactionManager.QMCLOTOBJ_TM
    NNOT_SHARED_LIBRARY                    = oti26mqstm
    NNOT_FACTORY_FUNCTION                  = NNOTMQSeriesTXManagerFactory
    NN_TM_MQS_QMGR                         = QMCLOTOBJ
```

Sample ODL Application

The following code example is the transport portion of an ODL application.

```
static int clotFunc(int mode, string queue, blob *sourceData, Blob *destData);

//MQSeries Transport Properties
string    NNMQS_TIL_GET_OPTIONS;
string NNMQS_TIL_GET_MATCH_OPTIONS;
string NNMQS_TIL_GET_MATCH_MSG_ID;
string NNMQS_TIL_GET_MATCH_CORREL_ID;
string NNMQS_TIL_OPEN_QMGR;
string NNMQS_TIL_OPEN_OPTIONS;
string NNMQS_TIL_DYN_Q_NAME;
string NNMQS_TIL_PUT_OPTIONS;
string NNMQS_TIL_GET_MAP_AG_FRM_MD;
string NNMQS_TIL_GET_MAP_MT_FRM_MD;

//Generic message properties
string NNOT_MSG_CORREL_ID;
string NNOT_MSG_EXPIRATION;
string NNOT_MSG_ID;
string NNOT_MSG_PERSISTENCE;
string NNOT_MSG_MSG_PRIORITY;

//MQ Series Message Properties PUT and GET properties
string NNMQS_MSG_ACCOUNTING_TOKEN;
string NNMQS_MSG_APPL_IDENTITY_DATA;
string NNMQS_MSG_APPL_ORIGIN_DATA;
string NNMQS_MSG_BACKOUT_COUNT;
string NNMQS_MSG_CODED_CHAR_SET_ID;
string NNMQS_MSG_MSG_ENCODING;
string NNMQS_MSG_FEEDBACK;
string NNMQS_MSG_FLAGS;
string NNMQS_MSG_FORMATS;
string NNMQS_MSG_GROUP_ID;
string NNMQS_MSG_OFFSET;
string NNMQS_MSG_ORIGINAL_LENGTH;
string NNMQS_MSG_PUT_APPL_NAME;
string NNMQS_MSG_PUT_APPL_TYPE;
string NNMQS_MSG_PUT_DATE;
string NNMQS_MSG_PUT_TIME;
string NNMQS_MSG_REPLY_TO_Q;
string NNMQS_MSG_REPLY_TO_QMGR;
string NNMQS_MSG_MSG_REPORT;
string NNMQS_MSG_SEQUENCE_NUMBER;
string NNMQS_MSG_TYPE;
string NNMQS_MSG_USER_IDENTIFIER;

//Rules Formatter Specific Properties
```

```
string OPT_APP_GRP;
string OPT_MSG_TYPE;

//Session configuration Properties
string NN_TIL_SES_MQS_OPEN_QMGR;
string NNOT_SHARED_LIBRARY;
string NNOT_FACTORY_FUNCTION;
string NNOT_TIL_OPEN_SESSION_ID;
string NNOT_TIL_OPEN_TSI;
string NNOT_TIL_GET_BLOCKING_TIMEOUT;
string NNOT_BLOCK_INFINITE;

       clot ot00;
       clot ot01;

static int clotobjFunc(int mode, string queue, blob *sourceData, blob*destData)
{


       int rv;

//if mode==1, then PUT if mode == 2, then GET

if(mode==1)
{
//***->MQSeries Transport Properties
NNMQS_TIL_GET_OPTIONS    ="";
NNMQS_TIL_OPEN_OPTIONS = "MQOO_PASS_ALL_CONTEXT|MQOO_FAIL_IF_QUIESCING";
NNMQS_TIL_PUT_OPTIONS =
"MQPMO_SYNCPOINT|MQPMO_PASS_ALL_CONTEXT|MQPMO_FAIL_IF_QUIESCING";
|MQPMO_NEW_MSG_ID|MQPMO_NEW_CORREL_ID;

//***->MQSeries Message Properties
NNMQS_MSG_REPLY_TO_Q= "REPLY";
NNMQS_MSG_REPLY_TO_Q_MGR = "QMCLOT";
NNMQS_MSG_TYPE = "MQMT_DATAGRAM";
NNMQS_MSG_MSG_REPORT = "MQMT_REPORT"

//***->Rules Formatter Specific Properties
OPT_APP_GRP = "IMPACT_APP_GRP"
OPT_MSG_TYPE = "IMPACT_MSG"

//Add transport properties
ot00.addProp(OT_PROP_TRANSPORT, "NNOT_TIL_OPEN_TSI", queue);

//Create context
```

```
ot00.create("CLOTOBJ_CON)";

//Open transport
ot00.open("CLOTOBJ_TRAN");

//Set data
ot00.setData(destData);

//add put properties
ot00.addProp(OT_PROP_PUT, "NNMQS_MSG_REPLY_TO_Q_MGR",
NNMQS_MSG_REPLY_TO_Q_MGR);
ot00.addProp(OT_PROP_PUT, "NNMQS_MSG_REPLY_TO_Q", NNMQS_MSG_REPLY_TO_Q);
ot00.addProp(OT_PROP_PUT, "NNMQS_MSG_TYPE", NNMQS_MSG_TYPR);
ot00.addProp(OT_PROP_PUT, "NNMQS_MSG_MSG_REPORT", NNMQS_MSG_MSG_REPORT)'
ot00.addProp(OT_PROP_PUT, "NNMQS_TIL_PUT_OPTIONS", NNMQS_TIL_PUT_OPTIONS);

//Begin transaction
ot00.begin();

//Put data to queue
ot00.put();

//commit transaction
ot00.commit();

//close transport
ot00.close();
}
else
if(mode==2)
{
//***->MQSeries Message Properties
NNMQS_MSG_REPLY_TO_Q = "REPLY";
NNMQS_MSG_REPLY_TO_Q_MGR = "QMCLOT";
NNMQS_TIL_OPEN_QMGR = "QMCLOT";
NNMQS_MSG_TYPE = "MQMT_REQUEST";
NNMQS_MSG_REPLY = "MQMT_REPLY";
NNMQS_MSG_MSG_REPORT = "MQMT_REPORT";
NNMQS_TIL_OPEN_QMGR = "QMCLOT";

//Add transport props
ot01.addProp(OT_PROP_TRANSPORT, "NNOT_TIL_OPEN_TSI", queue);

//Create context
ot01.create("CLOTOBJ_CON");
```

```
//Set data
ot01.open("CLOTOBJ_TRAN");

//Begin transaction
ot01.begin();

//Get data from queue
rv = ot01.get();

//get data from clot
ot01.getData(sourceData);

//commit transaction
ot01.commit();

//close transport
ot01.close();

}

return 1;
}
```

Note See the *MQSeries Driver for Open Transport Configuration Guide*, MQRFH2 Support, for detailed information on accessing information in an MQSeries header.

CHAPTER 8

Using Shared Libraries

This chapter provides procedures calling shared libraries from ODL applications.

Introduction

e-Biz Impact enables you to create a library that bridges the ODL applications and C/C++ functions that you want to implement. The user creates a definition file that defines the ODL to C/C++ mappings, then uses that file and BIDL to generate code that is compiled into the bridge library that allows ODL to call the C/C++ functions.

Note The following procedures apply to both Windows and UNIX shared libraries. Windows library files are used in the examples.

Implementing custom C/C++ functions in ODL projects

To call C/C++ shared library functions from ODL applications, follow these steps:

1 Create a collection of desired C/C++ functions callable from ODL. All functions must have a return value type of int.

2 Use a C++ compiler to compile the C/C++ functions collection into library form.

3 Create a custom C/C++ function definition file, which is loaded by the ODL engine, and correctly maps ODL calls to the custom C functions.

You must save this file as *definition.bdl*, and the file must be in the following format for the BIDL code generator to properly generate the appropriate code. If the definition file is incorrectly formatted, you receive an error when executing the BIDL code generator.

1 Definition file format:

```
interface name
{
[import("custom-c-function-name")]
int corresponding-odl-function-name (argument type argument name, ...);

[import("custom-c-function-name")]
int corresponding-odl-function-name (argument type argument name, ...);

[import("custom-c-function-name")]
int corresponding-odl-function-name (argument type argument name, ...);

.
.
.
};
```

- *name* – user-defined name for the custom collection.
- *custom-c-function-name* – custom C/C++ function name, mapped to the *corresponding-odl-function-name* in the ODL application.
- *corresponding-odl-function-name* – a corresponding ODL function mapped to the custom C/C++ function. Call this in the ODL application to call the custom C/C++ function in your library.
- *argument type* – matches the custom *custom-c-function-name*.
- *argument name* – matches the custom *custom-c-function-name*.

Repeat the definitions to define all functions and function arguments. Save the file with a *.bdl* file name extension.

2 Example definition file:

```
interface abc2002
{
[import("pepperBrook")]int myfunction0(char hlschar,
                                       short slshort,
                                       int hlsint,
                                       long hlslong);
```

```
[import("shadyMeadow")]int myfunction1(unsigned char hlschar,
                                       unsigned short slshort,
                                       unsigned int hlsint,
                                       unsigned long hlslong);
[import("mayWind")]int myfunction2(char *hlschar,
                                   short *slshort,
                                   int *hlsint,
                                   long *hlslong);
[import("dawnHaven")]int myfunction3(unsigned char *hlschar,
                                     unsigned short *slshort,
                                     unsigned int *hlsint,
                                     unsigned long *hlslong);
[import("creekside")]int myfunction4(string sl, string *ps2);
);
```

When you create the custom definition file, note that:

- The return type of each function must be int.
- For each C/C++ function, there is an ODL function mapping. In the example definition file, the C function pepperBrook() is mapped to the ODL function myfunction0. Within the ODL project, the C function pepperBrook() is callable only through the myfunction0 alias.
- The signatures of the C functions and ODL functions must match up. For the custom library to work correctly, the definition file must accurately define the custom C functions argument. In the case of the example C function pepperBrook(), it takes in a char, short, int, long. On the ODL project side, the C function can be called by the corresponding ODL function name, myfunction0 and with the char, short, int, and long argument variables passed to it.

4 Run the BIDL code generator along with the function definition file and the provided template file to create code for the mapping library.

```
ims bidl [function def file] [template file] [output filename]
```

For example:

```
ims bidl definition.bdl bidlTemplate.cpp test.cpp
```

Note The location of the ims wrapper script must be in your path, or you must enter the complete path when you enter the ims command.

- *function def file* – used by the BIDL code generator to create a *.cpp* file loaded by the ODL engine that correctly maps the ODL calls to custom C/C++ functions. In the example given, this is *definition.bdl*.

- *template file* – contains header files required by the BIDL code generator. This file—*bidlTemplate.cpp*—is located in *x:\Sybase\ImpactServer-5_4\include\IMPACT\BIDL* on Windows and in *<installation_directory>/Sybase/ImpactServer-5_4/include/IMPACT/BIDL* on UNIX. Do not modify the template file.
- *output filename* – the mapping library code used when compiling the custom C/C++ library. In the example shown, this is *test.cpp*.

5 Compile the library.

Use the mapping *.cpp* file generated by the BIDL code generator, along with your make file, custom DLL, and *ims54odl* library to compile the custom library that contains your custom C/C++ functions.

6 Configure the ODL application using the Configurator. Specify the resulting *.cpp* file by modifying the configuration values for the associated ODL application (right-click the application and select Properties), on the Advanced tab, in the External Libraries section. Identify the alias name in the Name field, and enter the actual library name in the Library field.

Be sure that the ODL project file also references the resulting library. For example:

```
#dll libx "aliaslib.dll" //alias for actual library

clinit()
{
.
.
```

APPENDIX A

Conversion Tables

This appendix provides conversion tables for ASCII and EBCDIC character sets for the three-character translation functions EbsAsc, AscEbc, and charTranslate.

EbcAsc translations

Table A-1 lists the translations form EBCDIC to ASCII, using the EbcAsc function.

When translating an EBCDIC hexadecimal value, such as A2, go to row value A0 and move across to the third column (2A). This EBCDIC hexadecimal translates to the ASCII character “s.”

Use the upper column numbers for the first 8 values in a hexadecimal sequence (0 to 7), and the lower column numbers for the second 8 values (8 to F). To look up the hexadecimal value 6E, go to row value 68, and move across to column E. This translates to the ASCII character “>”.

Table A-1: EBCDIC to ASCII translations

Row value	0 8	1 9	2 A	3 B	4 C	5 D	6 E	7 F
0x00	0	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x08	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x10	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x18	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘*‘	‘ ‘	‘;’	‘ ‘
0x20	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x28	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘

Row value	0 8	1 9	2 A	3 B	4 C	5 D	6 E	7 F
0x30	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x38	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x40	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x48	‘ ‘	‘ ‘	‘\ \‘	‘.‘	‘<‘	‘(‘	‘+’	‘|’
0x50	‘&‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x58	‘ ‘	‘ ‘	‘!‘	‘$‘	‘*‘	‘)’	‘;‘	‘^ ‘
0x60	‘-‘	‘ / ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x68	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘%‘	‘_‘	‘>‘	‘?‘
0x70	‘ ‘	‘^’	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x78	‘ ‘	‘ ’ ‘	‘:‘	‘#‘	‘@‘	‘ \ ‘	‘=‘	‘ ” ‘
0x80	‘ ‘	‘a’	‘b’	‘c’	‘d’	‘e’	‘f’	‘g’
0x88	‘h’	‘i’	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0x90	‘ ‘	‘j’	‘k’	‘l’	‘m’	‘n’	‘o’	‘p’
0x98	‘q’	‘r’	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0xA0	‘ ‘	‘~’	‘s’	‘t’	‘u’	‘v’	‘w’	‘x’
0xA8	‘y’	‘z’	‘ ‘	‘ ‘	‘ ‘	‘[‘	‘ ‘	‘ ‘
0xB0	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0xB8	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘]’	‘ ‘	‘ ‘
0xC0	‘{‘	‘A’	‘B’	‘C’	‘D’	‘E’	‘F’	‘G’
0xC8	‘H’	‘I’	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0xD0	‘}’	‘J’	‘K’	‘L’	‘M’	‘N’	‘O’	‘P’
0xD8	‘Q’	‘R’	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0xE0	‘\ \’	‘ ‘	‘S’	‘T’	‘U’	‘V’	‘W’	‘X’
0xE8	‘Y’	‘Z’	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘
0xF0	‘0’	‘1’	‘2’	‘3’	‘4’	‘5’	‘6’	‘7’
0xF8	‘8’	‘9’	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘	‘ ‘

AscEbc translations

Table A-2 lists the translations form ASCII to EBCDIC, using the AscEbc function.

When translating ASCII hexadecimal values, such as 0x5A (‘Z’), go to row value 0x58, and move over to column A, giving the value 0xE9.

Use the upper column numbers for the first 8 values in a hexadecimal sequence (0 to 7), and the lower column numbers for the second 8 values (8 to F).

Table A-2: ASCII to EBCDIC translations

Row value	0 8	1 9	2 A	3 B	4 C	5 D	6 E	7 F
0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x08	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x10	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x18	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
0x20	0x40	0x5A	0x7F	0x7B	0x5B	0x6C	0x50	0x7D
0x28	0x4D	0x5D	0x5C	0x4E	0x6B	0x60	0x4B	0x61
0x30	0xF0	0xF1	0xF2	0xF3	0xF4	0xF5	0xF6	0xF7
0x38	0xF8	0xF9	0x7A	0x5E	0x4C	0x7E	0x6E	0x6F
0x40	0x7C	0xC1	0xC2	0xC3	0xC4	0xC5	0xC6	0xC7
0x48	0xC8	0xC9	0xD1	0xD2	0xD3	0xD4	0xD5	0xD6
0x50	0xD7	0xD8	0xD9	0xE2	0xE3	0xE4	0xE5	0xE6
0x58	0xE7	0xE8	0xE9	0xAD	0xE0	0xBD	0x5F	0x6D
0x60	0x79	0x81	0x82	0x83	0x84	0x85	0x86	0x87
0x68	0x88	0x89	0x91	0x92	0x93	0x94	0x95	0x96
0x70	0x97	0x98	0x99	0xA2	0xA3	0xA4	0xA5	0xA6
0x78	0xA7	0xA8	0xA9	0xC0	0x4F	0xD0	0xA1	0x00

Character translations

The charTranslate built-in function allows you to convert between ASCII, EBCDIC, T-11 EBCDIC, and TN EBCDIC.

Table A-3: Character translation

char	ASCII	EBCDIC	T-11	TN
“NUL”	0x00	0x00	0x00	0x00
“ACK”	0x06	0x2e	0x2e	0x2e
“BEL”	0x07	0x2f	0x2f	0x2f
“BS”	0x08	0x16	0x16	0x16
“BYP”	CTTNOT	0x24	0x24	0x24
“CAN”	0x18	0x18	0x18	0x18
“CR”	0x0d	0x0d	0x0d	0x0d

char	ASCII	EBCDIC	T-11	TN
“CSP”	CTTNOT	0x2b	0x2b	0x2b
“CU1”	CTTNOT	0x1b	0x1b	0x1b
“CU3”	CTTNOT	0x3b	0x3b	0x3b
“DC1”	0x11	0x11	0x11	0x11
“DC2”	0x12	0x12	0x12	0x12
“DC3”	0x13	0x13	0x13	0x13
“DC4”	0x14	0x3c	0x3c	0x3c
“DEL”	0x7f	0x07	0x07	0x07
“DLE”	0x10	0x10	0x10	0x10
“DS”	CTTNOT	0x20	0x20	0x20
“EM”	0x19	0x19	0x19	0x19
“ENQ”	0x05	0x2d	0x2d	0x2d
“EO”	CTTNOT	0xff	0xff	0xff
“EOT”	0x04	0x37	0x37	0x37
“ESC”	0x1b	0x27	0x27	0x27
“ENQ”	0x05	0x2d	0x2d	0x2d
“ETB”	0x17	0x26	0x26	0x26
“ETX”	0x03	0x03	0x03	0x03
“FF”	0x0c	0x0c	0x0c	0x0c
“FS”	0x1c	0x22	0x22	0x22
“GE”	CTTNOT	0x08	0x08	0x08
“GS”	0x1d	CTTNOT	CTTNOT	CTTNOT
“HT”	0x09	0x05	0x05	0x05
“IFS”	CTTNOT	0x1c	0x1c	0x1c
“IGS”	CTTNOT	0x1d	0x1d	0x1d
“IR”	CTTNOT	0x33	0x33	0x33
“IRS”	CTTNOT	0x1e	0x1e	0x1e
“IT”	CTTNOT	0x39	0x39	0x39
“IUS”	CTTNOT	0x1f	0x1f	0x1f
“LF”	0x0a	0x25	0x25	0x25
“MFA”	CTTNOT	0x2c	0x2c	0x2c
“NAK”	0x15	0x3d	0x3d	0x3d
“NBS”	CTTNOT	0x36	0x36	0x36
“NL”	CTTNOT	0x15	0x15	0x15
“NSP”	CTTNOT	0xe1	0xe1	0xe1
“POC”	CTTNOT	0x17	0x17	0x17
“PP”	CTTNOT	0x34	0x34	0x34

char	ASCII	EBCDIC	T-11	TN
“RES”	CTTNOT	0x14	0x14	0x14
“RFF”	CTTNOT	0x3a	0x3a	0x3a
“RNL”	CTTNOT	0x06	0x06	0x06
“RPT”	CTTNOT	0x0a	0x0a	0x0a
“RS”	0x1e	CTTNOT	CTTNOT	CTTNOT
“RSP”	CTTNOT	0x41	0x41	0x41
“SA”	CTTNOT	0x28	0x28	0x28
“SBS”	CTTNOT	0x38	0x38	0x38
“SEL”	CTTNOT	0x04	0x04	0x04
“SFE”	CTTNOT	0x29	0x29	0x29
“SHY”	CTTNOT	0xca	0xca	0xca
“SI”	0x0f	0x0f	0x0f	0x0f
“SM”	CTTNOT	0x2a	0x2a	0x2a
“SO”	0x0e	0x0e	0x0e	0x0e
“SOH”	0x01	0x01	0x01	0x01
“SOS”	CTTNOT	0x21	0x21	0x21
“SP”	0x20	0x40	0x40	0x40
“SPS”	CTTNOT	0x09	0x09	0x09
“STX”	0x02	0x02	0x02	0x02
“SUB”	0x1a	0x3f	0x3f	0x3f
“SYN”	0x16	0x32	0x32	0x32
“TRN”	CTTNOT	0x35	0x35	0x35
“UBS”	CTTNOT	0x1a	0x1a	0x1a
“US”	0x1f	CTTNOT	CTTNOT	CTTNOT
“VT”	0x0b	0x0b	0x0b	0x0b
“WUS”	CTTNOT	0x23	0x23	0x23
“ **!** ”	0x21	0xdd	0x5a	0x5a
“ \ ”	0x22	CTTNOT	0x7f	0x7f
“ # ”	0x23	0x7b	0x7b	0x7b
“ **$** ”	0x24	0x5b	0x5b	0x5b
“ % ”	0x25	0x6c	0x6c	0x6c
“ **&** ”	0x26	0x50	0x50	0x50
“ ‘ ”	0x27	CTTNOT	0x7d	0x7d
“ **(** ”	0x28	0x4d	0x4d	0x4d
“ **)** ”	0x29	0x5d	0x5d	0x5d
“ ***** ”	0x2a	0x5c	0x5c	0x5c
“ + ”	0x2b	CTTNOT	0x4e	0x4e

char	ASCII	EBCDIC	T-11	TN
“ **“** ”	0x2c	0x6b	0x6b	0x6b
“ **-** ”	0x2d	0x60	0x60	0x60
“ **.** ”	0x2e	0x4b	0x4b	0x4b
“ / ”	0x2f	0x61	0x61	0x61
“0”	0x30	0xf0	0xf0	0xf0
“1”	0x31	0xf1	0xf1	0xf1
“2”	0x32	0xf2	0xf2	0xf2
“3”	0x33	0xf3	0xf3	0xf3
“4”	0x34	0xf4	0xf4	0xf4
“5”	0x35	0xf5	0xf5	0xf5
“6”	0x36	0xf6	0xf6	0xf6
“7”	0x37	0xf7	0xf7	0xf7
“8”	0x38	0xf8	0xf8	0xf8
“9”	0x39	0xf9	0xf9	0xf9
“ **:** ”	0x3a	0x7d	0x7a	0x7a
“ **;** ”	0x3b	0x5e	0x5e	0x5e
“ < ”	0x3c	0x4e	0x4c	0x4c
“ = ”	0x3d	CTTNOT	0x7e	0x7e
“ > ”	0x3e	0x7e	0x6e	0x6e
“ **?** ”	0x3f	0xc0	0x6f	0x6f
“ **@** ”	0x40	0x7c	0x7c	0x7c
“A”	0x41	0xc1	0xc1	0xc1
“B”	0x42	0xc2	0xc2	0xc2
“C”	0x43	0xc3	0xc3	0xc3
“D”	0x44	0xc4	0xc4	oxc4
“E”	0x45	0xc5	0xc5	0xc5
“F”	0x46	0xc6	0xc6	0xc6
“G”	0x47	0xc7	0xc7	0xc7
“H”	0x48	0xc8	0xc8	0xc8
“I”	0x49	0xc9	0xc9	0xc9
“J”	0x4a	0xd1	0xd1	0xd1
“K”	0x4b	0xd2	0xd2	0xd2
“L”	0x4c	0xd3	0xd3	0xd3
“M”	0x4d	0xd4	0xd4	0xd4
“N”	0x4e	0xd5	0xd5	0xd5
“O”	0x4f	0xd6	0xd6	0xd6
“P”	0x50	0xd7	0xd7	0xd7

char	ASCII	EBCDIC	T-11	TN
“Q”	0x51	0xd8	0xd8	0xd8
“R”	0x52	0xd9	0xd9	0xd9
“S”	0x53	0xe2	0xe2	0xe2
“T”	0x54	0xe3	0xe3	0xe3
“U”	0x55	0xe4	0xe4	0xe4
“V”	0x56	0xe5	0xe5	0xe5
“W”	0x57	0xe6	0xe6	0xe6
“X”	0x58	0xe7	0xe7	0xe7
“Y”	0x59	0xe8	0xe8	0xe8
“Z”	0x60	0xe9	0xe9	0xe9
“ **[** ”	0x5b	0xad	0xad	0xad
“ \ \ ”	0x5c	0x6e	0x0e	0xe0
“ **]** ”	0x5d	0xbd	0xbd	0xbd
“ ^ ”	0x5e	0x5f	0x5f	0x5f
“ _ ”	0x5f	CTTNOT	0x6d	0x6d
“ ‘ ”	0x60	0x79	0x79	0x79
“a”	0x61	CTTNOT	0x81	0x81
“b”	0x62	CTTNOT	0x82	0x82
“c”	0x63	CTTNOT	0x83	0x83
“d”	0x64	CTTNOT	0x84	0x84
“e”	0x65	CTTNOT	0x85	0x85
“f”	0x66	CTTNOT	0x86	0x86
“g”	0x67	CTTNOT	0x87	0x87
“h”	0x68	CTTNOT	0x88	0x88
“i”	0x69	CTTNOT	0x89	0x89
“j”	0x6a	CTTNOT	0x91	0x91
“k”	0x6b	CTTNOT	0x92	0x92
“l”	0x6c	CTTNOT	0x93	0x93
“m”	0x6d	CTTNOT	0x94	0x94
“n”	0x6e	CTTNOT	0x95	0x95
“o”	0x6f	CTTNOT	0x96	0x96
“p”	0x70	CTTNOT	0x97	0x97
“q”	0x71	CTTNOT	0x98	0x98
“r”	0x72	CTTNOT	0x99	0x99
“s”	0x73	CTTNOT	0xa2	0xa2
“t”	0x74	CTTNOT	0xa3	0xa3
“u”	0x75	CTTNOT	0xa4	0xa4

char	ASCII	EBCDIC	T-11	TN
“v”	0x76	CTTNOT	0xa5	0xa5
“w”	0x77	CTTNOT	0xa6	0xa6
“x”	0x78	CTTNOT	0xa7	0xa7
“y”	0x79	CTTNOT	0xa8	0xa8
“z”	0x7a	CTTNOT	0xa9	0xa9
“ ”	0x7b	CTTNOT	0xc0	0xc0
“ | ”	0x7c	0x6a	0x6a	0x6a
“ ”	0x7d	CTTNOT	0xd0	0xd0
“ ~ ”	0x7e	0xa1	0xa1	0xa1

ASCII character set

Table A-4: ASCII characters

Mne	Dec	Oct	Hex	Description
nul	0	00	0x0	Null or all zeros
soh	1	01	0x1	Start of heading
stx	2	02	0x2	Start of text
etx	3	03	0x3	End of text
eot	4	04	0x4	Eng of transmission
enq	5	05	0x5	Enquiry
ack	6	06	0x6	Acknowledge
bel	7	07	0x7	Bell or alarm
bs	8	010	0x8	Backspace
ht	9	011	0x9	Horizontal tab
lf	10	012	0xa	Line feed
vt	11	013	0xb	Vertical tab
ff	12	014	0xc	Form feed
cr	13	015	0xd	Carriage return
so	14	016	0xe	Shift out
si	15	017	0xf	Shift in
dle	16	020	0x10	Data link escape
dcl	17	021	0x11	Device control 1
dc2	18	022	0x12	Device control 2
dc3	19	023	0x13	Device control 3

Mne	Dec	Oct	Hex	Description
dc4	20	024	0x14	Device control 4
nak	21	025	0x15	Negative acknowledge
syn	22	026	0x16	Snyc
etb	23	027	0x17	End transmission block
can	24	030	0x18	Cancel
em	25	031	0x19	End of medium
sub	26	032	0x1a	Substitute
esc	27	033	0x1b	Escape
fs	28	034	0x1c	File separator
gs	29	035	0x1d	Group separator
rs	30	036	0x1e	Record separator
us	31	037	0x1f	Unit separator
sp	32	040	0x20	Space
del	33	041	0x21	Delete

APPENDIX B

Error Conditions

This appendix provides a list of error conditions for the I/O file objects.

I/O file error conditions

The following list describes the possible error conditions that an I/O file object may encounter during a disk operation, such as reading from or writing to the file. To determine the file error number returned by the operating system when the I/O file object encounters an error, use the errno method.

Table B-1: Error conditions

Mnemonic	Description
FENOERROR	No error.
FEPERM	User needs root permission to access the file.
FENOENT	No such file or directory at the specified location.
FEINTR	A system call was interrupted.
FEIO	An I/O error occurred.
FEBADF	Bad file number. You used the read method on a file before using the open method to open the file.
FENOMEM	Not enough memory available on the system.
FEACCES	Permissions set do not allow access to the file.
FEEXIST	The file or directory exists at the specified location.

Mnemonic	Description
FEXDEV	You tried to use the rename method across file systems.
FENOTDIR	Not a directory.
FEISDIR	The I/O file object is referencing a directory.
FEINVAL	You passed an invalid parameter to a method.
FENFILE	The internal table of the number of files open has reached the operating system's limit. This limit is O/S dependent and may be reconfigurable. Refer to your system's operating manual for details.
FEMFILE	There are too many files open on the system.
FETXTBSY	You tried to delete a file that is in use.
FEFBIG	The file is too large. How large a file you may open depends upon the operating system's limit and may be reconfigurable.Refer to your system's operating manual for details.
FENOSPC	Out of disk space.
FESPIPE	Illegal seek. You read past the end of the file.
FEROFS	Read only file system.
FENOLCK	System record lock table is full.
FEILSEQ	You passed a bad regular expression to the openDir method.
FENOTEMPTY	The directory is not empty.
FENAMETOOLONG	A component of the directory path in the I/O File object's associated file field is too long, or the entire path is too long. This limit is O/S dependent and may be reconfigurable. Refer to your system's operating manual for details.
FEUNKNOWN	Unable to determine the error.

APPENDIX C

Using the Command Line-to-DFC Program

The command line-to-DFC program enables you to send DFC calls from within a shell script or from other types of applications that are not designed to use the normal DFC routing function commands. The function called must be listed in the file.

Syntax

ims54cmdlinedfc -cluster.name *clusterName*
-cluster.server *clusterServer*
-domain.type *domainType*
-domain.name *domainName*
-env.snmp.port *snmpPort*
-env.trap.port *trapPort*
-file *fileName*
-dfc.function *dfcFunctionName*
-dfc.flavor *dfcFlavor*
-dfc.args[*arg1,arg2,arg3,...*]

Parameters

- *clusterName* – name of the cluster that receives the DFC call.
- *clusterServer* – name of the e-Biz Impact server on which the cluster is running.
- *domainType* – type of the domain under which the cluster is running. The default value is "Impact."
- *domainName* – the name of the domain under which the cluster is running. The default value is "Impact."
- *snmpPort* – the SNMP port to which the cluster publishs telemetries. The default value of is 161.
- *trapPort* – the SNMP trap port to which the cluster publish alerts. The default value is 162.
- *fileName* – name of an NNConfig style file that can optionally contain all or some command line arguments, and in which the command line argument takes precedence over a value in the file.
- *dfcFunctionName* – name of the DFC function to call.
- *dfcFlavor* – flavor of the DFC to call. The default value is "0."

- *arg1, arg2, arg3.....* – optional list of string arguments to the DFC function, and if provided on the command line, the *-dfc.args* key and its values must be the last arguments on the command line.

Note To run these command use the ims wrapper scripts, see "Using wrapper scripts" in chapter 1 of the *e-Biz Impact Command Line Tools Guide*.

Return values

Returns 0 for initialization failure and 1 otherwise.