Informix Embedded SQL

TP/XA Programmer's Manual

INFORMIX-OnLine Dynamic Server, Version 7.2x
INFORMIX-OnLine Workgroup Server, Version 7.2x
INFORMIX-OnLine XPS, Version 8.1x
INFORMIX-SE, Version 7.2x
INFORMIX-Universal Server, Version 9.1x

Documentation Team: Bob Berry, Twila Booth, Evelyn Eldridge-Diaz, Smita Joshi

Table of Contents

Introduction

Read this introduction for an overview of the information provided in this manual and for an understanding of the documentation conventions used.

About This Manual

This manual explains the use and features of the TP/XA library. As part of INFORMIX-ESQL/C, the TP/XA library facilitates communication between a third-party transaction manager and an Informix database server for the purpose of distributed transaction processing (DTP) in a multivendor database setting. This library allows the database server to operate as a database management system (DBMS) within a Resource Manager (RM) of an X/Open distributed transaction processing environment.

Types of Users

This manual is written for programmers and system administrators who are developing applications for a third-party transaction manager and an Informix database server.

If you have limited experience with relational databases, SQL, or your operating system, see the *Getting Started* manual for your database server for a list of supplementary manuals.

Software Dependencies

This manual assumes that you are using the TP/XA library with INFORMIX-ESQL/C, Version 9.13. In addition, you must use one of the following database servers:

- INFORMIX-OnLine Dynamic Server, Version 7.2x
- INFORMIX-OnLine Workgroup Server, Version 7.2x
- INFORMIX-OnLine XPS, Version 8.1x
- INFORMIX-SE, Version 7.2x
- INFORMIX-Universal Server, Version 9.1x

Assumptions About Your Locale

Informix products can support many languages, cultures, and code sets. All culture-specific information is brought together in a single environment, called a GLS (Global Language Support) locale.

This manual assumes that you are using the default locale, **en_us.8859-1**. This locale supports U.S. English format conventions for dates, times, and currency. In addition, this locale supports the ISO 8859-1 code set, which includes the ASCII code set plus many 8-bit characters such as é, è, and ñ.

If you plan to use nondefault characters in your data or your SQL identifiers, or if you want to conform to the nondefault collation rules of character data, you need to specify the appropriate nondefault locale(s). For instructions on how to specify a nondefault locale, additional syntax, and other considerations related to GLS locales, see the *INFORMIX-GLS Programmer's Manual* and the *Informix Guide to GLS Functionality*.

Demonstration Database

Your Informix software includes a demonstration database called **stores7** that contains information about a fictitious wholesale sporting-goods distributor. Informix includes source files for INFORMIX-ESQL/C demonstration programs that access and manipulate the data stored in **stores7**. Most of the examples in this manual are based on the **stores7** demonstration database.

For information on how to install **stores7** and access the code examples, see the *INFORMIX-ESQL/C Programmer's Manual*.

Documentation Conventions

This section describes the conventions that this manual uses. These conventions make it easier to gather information from this and other Informix manuals.

The following conventions are covered:

- Typographical conventions
- Icon conventions

Typographical Conventions

This manual uses the following standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth.

Convention	Meaning
KEYWORD	All keywords appear in uppercase letters in a serif font.
italics	Within text, new terms and emphasized words appear in italics. Within syntax diagrams, values that you are to specify appear in italics.
boldface	Identifiers (names of classes, objects, constants, events, functions, program variables, forms, labels, and reports), environment variables, database names, filenames, table names, column names, icons, menu items, command names, and other similar terms appear in boldface.
monospace	Information that the product displays and information that you enter appear in a monospace typeface.
KEYSTROKE	Keys that you are to press appear in uppercase letters in a sans serif font.
♦	This symbol indicates the end of feature-, product-, platform-, or compliance-specific information.

Tip: When you are instructed to "enter" characters or to "execute" a command, immediately press RETURN *after the entry. When you are instructed to "type" the text or to "press" other keys, no* RETURN *is required.*

Icon Conventions

Throughout the documentation, you will find text that is identified by several different types of icons. This section describes these icons.

Comment Icons

Comment icons identify warnings, important notes, or tips. This information is always displayed in italics.

Icon	Description
	The *warning* icon identifies vital instructions, cautions, or critical information.
	The *important* icon identifies significant information about the feature or operation that is being described.
	The *tip* icon identifies additional details or shortcuts for the functionality that is being described.

Feature, Product, and Platform Icons

Feature, product, and platform icons identify paragraphs that contain feature-, product-, or platform-specific information.

Icon	Description
GLS	Identifies information that is specific to the Informix Global Language Support (GLS) feature.
IUS	Identifies information that is specific to INFORMIX-Universal Server.
ODS	Identifies information that is specific to INFORMIX-OnLine Dynamic Server.
OWS	Identifies information that is specific to INFORMIX-OnLine Workgroup Server.
SE	Identifies information that is specific to INFORMIX-SE.
UNIX	Identifies information that is specific to the UNIX operating system.
WIN NT	Identifies information that is specific to the Windows NT environment.
WIN 95	Identifies information that is specific to the Windows 95 environment.
WIN NT/95	Identifies information that is specific to both Windows NT and Windows 95 environments.
XPS	Identifies information that is specific to INFORMIX-OnLine XPS.

These icons can apply to a row in a table, one or more paragraphs, or an entire section. A ♦ symbol indicates the end of the feature-, product-, or platform-specific information.

Additional Documentation

For additional information, you might want to refer to the following types of documentation:

- On-line manuals
- Printed manuals
- Error message files
- Documentation notes, release notes, and machine notes
- Related reading

On-Line Manuals

An Answers OnLine CD that contains Informix manuals in electronic format is provided with your Informix products. You can install the documentation or access it directly from the CD. For information about how to install, read, and print on-line manuals, see the installation insert that accompanies Answers OnLine.

Printed Manuals

To order printed manuals, call 1-800-331-1763 or send email to moreinfo@informix.com.

Please provide the following information:

- The documentation that you need
- The quantity that you need
- Your name, address, and telephone number

Error Message Files

Informix software products provide ASCII files that contain all of the Informix error messages and their corrective actions. The **finderr** utility displays these error messages on the screen. See the Introduction to the *Informix Error Messages* manual for a detailed description of these error messages.

To read the error messages in the ASCII file, Informix provides scripts that let you display error messages on the screen (**finderr**) or print formatted error messages (**rofferr**). For a detailed description of these scripts, see the Introduction to the *Informix Error Messages* manual.

Documentation Notes, Release Notes, Machine Notes

In addition to printed documentation, the following on-line files, located in the **$INFORMIXDIR/release/en_us/0333** directory, supplement the information in this manual.

On-Line File	Purpose
XADOC_9.1	The documentation-notes file describes features that are not covered in this manual or that have been modified since publication.
CLIENTS_2.0	The **CLIENTS_2.0** file lists the release-notes files for the 2.0 Client SDK. These release-notes files describe feature differences from earlier versions of Informix products and how these differences might affect current products. These files also contain information about any known problems and their workarounds.
ESQL/C_9.1	The machine notes file describes any special actions that are required to configure and use Informix products on your computer. The machine notes for TP/XA are contained in the machine notes file for INFORMIX-ESQL/C.

Please examine these files because they contain vital information about application and performance issues.

Related Reading

For information on the X/Open XA specification, consult *Distributed Transaction Processing: The XA Specification* by X/Open Company, Limited (February 1992).

Compliance with Industry Standards

The TP/XA library adheres to the X/Open XA interface specification developed by The Open Group to support large-scale, high-performance, on-line transaction-processing applications.

The American National Standards Institute (ANSI) has established a set of industry standards for SQL. Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992, on Informix database servers. In addition, many features of some Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL CAE (common applications environment) standards.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about corrections or clarifications that you would find useful. Include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

> Informix Software, Inc.
> SCT Technical Publications Department
> 4100 Bohannon Drive
> Menlo Park, CA 94025

If you prefer to send email, our address is:

> doc@informix.com

Or send a facsimile to the Informix Technical Publications Department at:

> 650-926-6571

We appreciate your feedback.

Informix and the X/Open Distributed Transaction-Processing Model

Several distinct models for transaction processing have emerged in the evolution of relational databases. Each model attempts to meet the changing needs of the business community. The Informix solution for distributed transaction processing (DTP) is based on the X/Open DTP model.

This chapter covers the following topics:

- A general introduction to a DTP system
- A description of the X/Open DTP model
- How TP/XA and Informix database servers fit into the X/Open DTP model

The TP/XA library is not available with the following Informix database servers:

- INFORMIX-SE ◆

- INFORMIX-OnLine XPS ◆

The TP/XA library is not available on the Windows NT or Windows 95 operating systems. ◆

The TP/XA library is bundled with the following Informix SQL API products:

- INFORMIX-ESQL/C
- INFORMIX-ESQL/COBOL

INFORMIX-ESQL/COBOL is not available with INFORMIX-Universal Server. ◆

Distributed Transaction Processing

This section provides the following information about DTP:

- A brief introduction to transaction processing
- A description of several features needed to support a DTP environment

Transaction Processing

A *transaction* is a unit of work that consists of an application-specific sequence of operations. A typical example of a transaction in the accounting world might be subtracting some amount from the accounts receivable ledger and adding the same amount to the cash ledger. The transaction consists of the subtraction and the addition operation taken together.

A *transaction-processing* system defines and coordinates interactions between multiple users and databases (or other shared resources). When a transaction includes operations in several databases or other shared resources, the goal of a transaction-processing system is to carry out this transaction in an efficient, reliable, and coordinated way.

The success of any transaction-processing system is measured against four critical objectives. Together, these objectives are known as the *ACID test*:

- Atomicity

  Are all operations within a transaction performed on an all-or-nothing basis?

- Consistency

  If a transaction must be aborted, is the data returned to its previous valid state?

- Isolation

  Are the results of a transaction invisible to other transactions until the transaction is committed?

- Durability

  Will the results of a transaction survive subsequent system failures?

In addition to these fundamental objectives, a transaction-processing system should have the following capabilities:

- Performance

  The transaction-processing system must be able to handle a large number of users without a corresponding performance loss.

- Resiliency

  The transaction-processing system must be able to recover in the event of system or computer failure.

Features of a DTP System

DTP is a form of transaction processing that supports transactions whose operations are distributed among different computers or among databases from different vendors. A DTP system must support all features of the general transaction-processing system, including the ACID test properties (see page 1-4). In addition, the DTP system must also support communication and cooperation among shared resources that are installed at *different* physical sites and are connected over a network. Databases from different vendors or on different computers are called *heterogeneous* databases.

On-line transaction-processing (OLTP) applications are often run in a DTP environment. (For a definition of OLTP applications, see the *Administrator's Guide* for your database server.) The following table lists features of a DTP system that are useful to OLTP applications.

DTP Feature	OLTP Application Feature	For More Information
DTP client and server programs	A large volume of well-defined application requests	"Client and Server Programs" on page 1-6
	Small, well-defined interactions between user and database	
Transaction management	Emphasis on system response	"Local and Global Transactions" on page 1-7
Global transactions and two-phase commit	Heavy database use of large, shared databases (or other resources)	"Two-Phase Commit Protocol" on page 1-10

Client and Server Programs

In the DTP model, an application is divided into the following parts:

- A *server* program provides one or more *services.* A *service* is a single function within the server program. It performs one database task for the application.

- *A client* program handles the user interface. It determines which services the user needs performed. To initiate a service, the client program sends a *service request* to the server program that offers that particular service.

Important: In the DTP model, the definitions of the terms "client" and "server" differ from their definitions in the Informix client/server architecture. In the DTP model, a "client" is not an entire application; it is only that part of the application that handles the user interface. In the DTP model, a "server" is not a database server but the second part of the application, the part that handles the database communication.

Figure 1-1 shows the relationship between the client program, the server program, and the service requests. Each instance of the application program contains *one* client program and *at least one* server program.
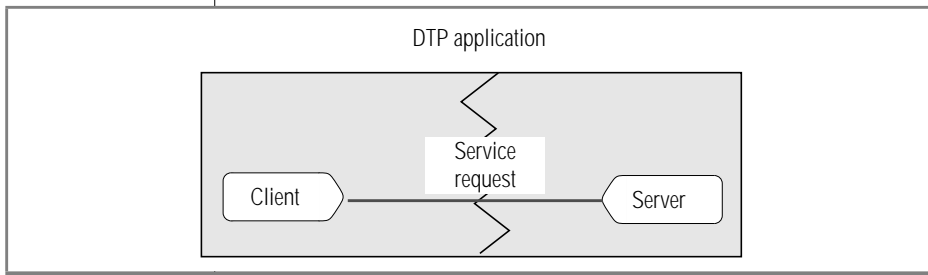
Figure 1-1
Application Program Showing Client and Server Programs

DTP application

Service request

Client

Server

The client and server are distinct programs. The client program does *not* perform the database tasks. Instead, it sends a service request to a server. The server program then initiates the execution of the desired service, a specific operation performed by a code module that is embedded in a server. A server program accepts requests and dispatches them to the appropriate service.

Tip: For an example of client and server programs, see "A Sample DTP ESQL/C Application Program" on page 3-16.

The following list shows additional benefits of dividing an application into client and server programs:

- When one of the client or server programs fails, it does not affect any other client or server processes.
- Server programs can be located on the same computer as the associated database server, providing centralized access for client programs.
- Server programs reduce redundant storage of service-related coding at the user site.
- Client programs can reside at the user site and can be tailored to the needs of the user.
- Modularity allows extension and reorganization of the client and server programs without rewriting existing code.

These benefits allow an OLTP application to support a large volume of well-defined application requests and to provide small, well-defined interactions between the user and the database.

Local and Global Transactions

The DTP model supports the following types of transactions:

- A *local transaction* involves only one service within a single server program, and it accesses only one database.
- A *global transaction* involves several services that might be located in different server programs, perhaps on different computers. A global transaction is also called a *distributed transaction*.

The DTP software must be able to handle both types of transactions in support of the application.

Figure 1-2 shows a local transaction that contains only one service request involving only one database management system (DBMS). This DBMS manages two databases, Database A and Database B.
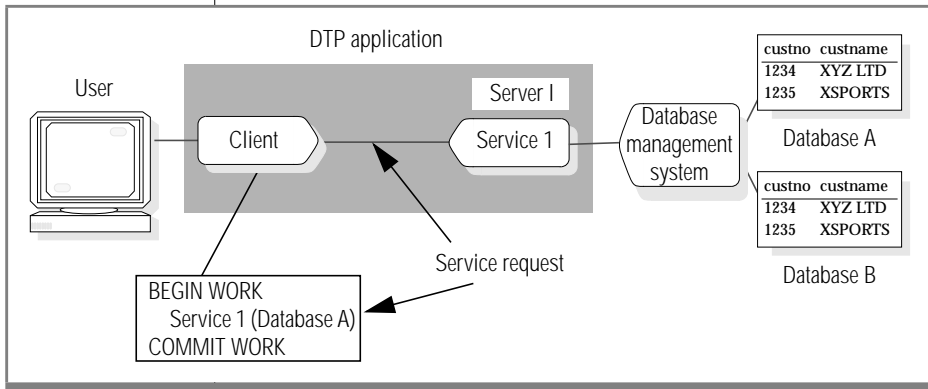
An application marks the start and end of a local transaction using only local database transaction commands such as the SQL statements BEGIN WORK, COMMIT WORK, and ROLLBACK WORK. For more information on these SQL statements, see the *Informix Guide to SQL: Syntax.* For more information on local transactions, see page 1-16.

When the service request involves more than one server program or database, the application must use a global transaction. To support such transactions, the DTP model must keep track of the following types of information:

- Which services are in which server program
- Which service request to send to which server process
- From which client process a given service request originated

Without this information, the DTP model cannot ensure the integrity of the databases in the event of a system or server failure.

To handle this information, the DTP model requires *transaction-management software*, as Figure 1-3 shows.
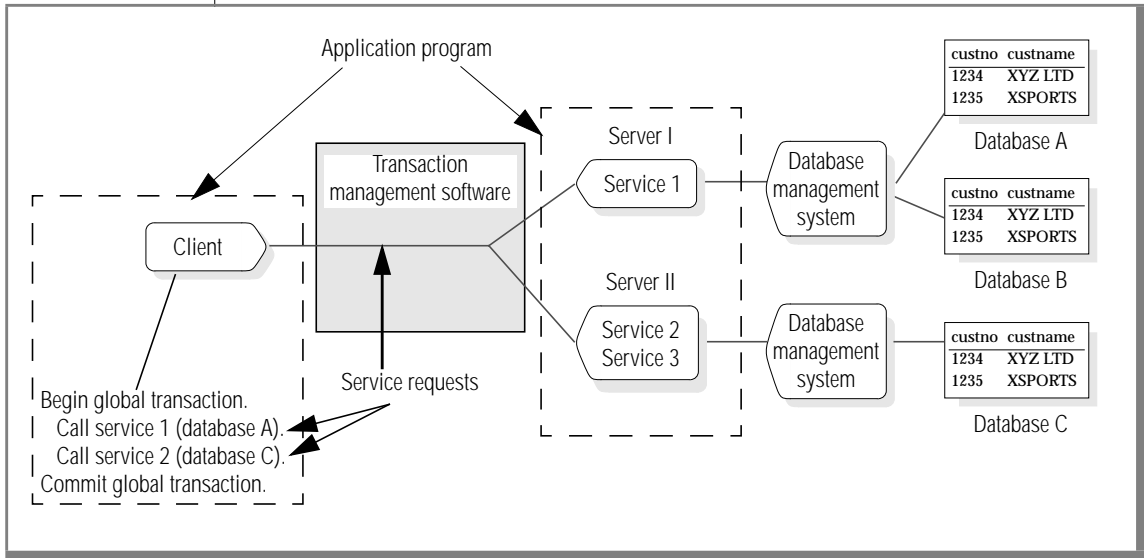
Figure 1-3 shows a general DTP model. For the X/Open DTP model, see Figure 1-8 on page 1-27.

To mark the start and end of a global transaction, an application program must use special transaction-demarcation commands known to the transaction-management software. These commands replace the database server transaction commands, such as the SQL statements BEGIN WORK, COMMIT WORK, and ROLLBACK WORK. For information on global transactions in the X/Open DTP environment, see page 1-16.

Transactions allow an OLTP application to increase the system response by efficiently grouping database operations.

Two-Phase Commit Protocol

The DTP system must support *two-phase commit* to provide reliable transaction data. The *two-phase commit protocol* governs the order in which a global transaction is committed and provides an automatic recovery mechanism in case a system or media failure occurs during transaction execution. Every global transaction has a coordinator and one or more participants, defined as follows:

- The *coordinator* directs the resolution of the global transaction. It decides whether the global transaction should be committed or aborted.

- Each *participant* directs the execution of one *transaction branch*, which is the part of the global transaction involving a single local database. A global transaction includes several transaction branches in the following situations:

    - When an application uses multiple processes to work for a global transaction

    - When multiple remote applications work for the same global transaction

The two-phase commit protocol consists of the following two phases:

■ Phase 1: Precommit phase

 During this phase, the coordinator directs each participant (database server or shared resource manager) to prepare its transaction branch to commit. Every participant notifies the coordinator whether it can commit its transaction branch.

 The coordinator, based on the response from each participant, decides whether to commit or abort the global transaction. It decides to commit only if *all* participants indicate that they can commit their transaction branches. If any participant indicates that it is *not* ready to commit its transaction branch (or if it does not respond), the coordinator decides to abort the global transaction. The coordinator records its decision in its log.

■ Phase 2: Postdecision phase

 Based on the decision in the precommit phase, the coordinator issues the appropriate message to each participant. Each participant then performs the requested action on its transaction branch and notifies the coordinator when it is finished. The transaction branch is either committed or aborted by all participants.

The goal of the two-phase commit protocol is to have the coordinator determine the likelihood of success for a global transaction *before* the participants actually handle their individual transaction branches. Once a participant has actually handled its transaction branch, it is very difficult to undo the work. Some cases can cause a participant to decide how to handle its transaction branch independently of the coordinator. For more information about these cases, see "Heuristic Decisions" on page 1-22.

Tip: For general information on two-phase commit protocol, see the "Administrator's Guide" for your database server.

The two-phase commit protocol for global transactions ensures that an OLTP application with heavy database use does not lose data in the event of system failure.

The X/Open DTP Model

The X/Open DTP model is a DTP system specified by the X/Open Company Limited in the document *Distributed TP: The XA Specification*. This XA specification describes a uniform way to structure a DTP system. Although other DTP systems provide the same set of features as this model, they use proprietary methods and interfaces. The X/Open DTP model is an *open* model, based on the XA specification.

When an application conforms to this model, it can use global transactions that include multivendor database servers. A TP/XA library and the database server enable an application developer to build OLTP applications that conform to the *X/Open* XA specification.

The X/Open DTP model consists of the following parts:

- Application program

  The application program defines the boundaries of a transaction and specifies the actions that constitute a transaction.

- Resource Manager

  The Resource Manager (RM) provides access to a shared resource. Usually, an RM is a database server or file-access system with one or more server programs that access the database server or file-access system.

- Transaction Manager

  The Transaction Manager (TM) manages the routing and transaction-processing control of service requests. It manages global transactions, coordinating their resolution and any failure recovery. TM software also lets you establish the communication links among the client and server programs.

Each part is discussed in more detail in the following sections. For more information on how these parts communicate, see "The Model Interfaces" on page 1-23.

Figure 1-4 shows a conceptual version of the X/Open DTP model.
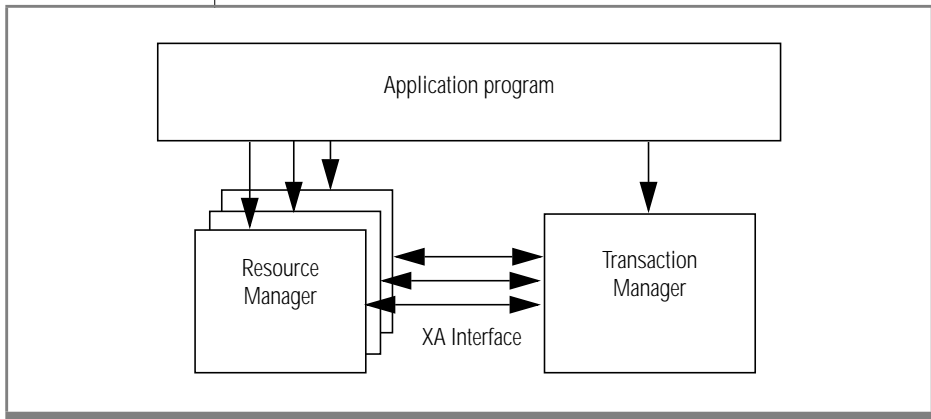
The Application Program

In the X/Open DTP model, the application program contains the *client* program of the DTP client/server structure (see page 1-6). The application program is written by the DTP application developer and performs the following tasks:

- Receives a user request and turns it into the appropriate service request(s)

- Handles the user interface by accepting and displaying data for the user

- Defines the global transactions by making the appropriate transaction-management commands (see page 1-15) to mark the start and end of a transaction

- Performs any local transactions through the native RM API interface to start and end the transaction as well as to execute the actual transaction operations

For more information on building the application program, see Chapter 3, "Programming in an X/Open Environment."

The Resource Manager

In an X/Open DTP environment, the RM manages a set of shared resources. For a database application, the most commonly accessed shared resources are *databases,* and these databases are managed by a DBMS. A single DBMS might manage several independent databases.

The RM includes the following parts:

- A DBMS that supports the following tasks:
  - ❑ Understands how to commit or roll back a transaction
  - ❑ Communicates with the server program in support of a service, through the native RM API
  - ❑ Recognizes a global transaction, accepting a transaction identifier (XID) from the TM and mapping it to an RM-specific XID. (For more information on global transaction identifiers (GTRIDs), see "Assigning Transaction Identifiers" on page 1-17.)
  - ❑ Acts as a participant in the two-phase commit and recovery, acting on the XA requests it receives from the TM. (For more information, see "Controlling the Two-Phase Commit" on page 1-20.)

  The DBMS software is usually provided by a third-party vendor such as Informix. In the Informix implementation of an RM, the Informix database server is the DBMS. To establish the Informix database server as the RM DBMS, you must provide certain information to the TM. For more information, see Chapter 2, "Integrating the Database Server and TP/XA into the X/Open DTP Model."

- The *server* program that performs the following tasks:

  ❑ Defines the services needed by the client application program that are supported by the associated DBMS

  ❑ Communicates with the DBMS in support of a service, through a native RM API

  The server program is written by the DTP application developer in a language that supports the native RM API. In the Informix implementation of an RM, the server program is written in one of the following SQL APIs, working through the TP/XA library:

  ❑ INFORMIX-ESQL/C

  ❑ INFORMIX-ESQL/COBOL

  INFORMIX-ESQL/COBOL is not available with INFORMIX-Universal Server. ♦

IUS

The Transaction Manager

The Transaction Manager (TM) performs the following transaction-management tasks:

- Manages local and global transactions
- Assigns XIDs
- Routes and queues service requests from a client process to the appropriate server process
- Acts as the coordinator in two-phase commit and recovery

The TM must know about all computers, application programs (clients), services, and RMs (servers and DBMS systems) on its network that are involved in global transactions. This knowledge enables the TM to coordinate the activity among these entities.

Managing Transactions

In an X/Open DTP environment, a single transaction can span one or several RMs. The TM can manage both types of transactions, local (one RM) and global (several RMs). For a general description of local and global transactions, see page 1-7.

Local Transactions

A *local transaction* involves one service in a single RM. A local transaction can occur in one of the following ways:

■ Under the control of the TM

The application program calls special transaction-management calls provided as part of the AP-TM interface to begin and commit the work in the local transaction. For more information, see "The AP-to-TM Interface" on page 1-24.

■ Under the control of the RM

The application program uses the appropriate calls in the AP-RM, the native RM API, to start and end the local transaction. When this API is SQL, the application program starts the transaction with a BEGIN WORK statement and ends the transaction with a COMMIT WORK statement, or it uses single-statement transactions (implicit transactions). For more information, see "The AP-to-RM Interface" on page 1-23.

Global Transactions

In the X/Open DTP environment, many RMs can operate in support of the same global transaction. For example, an application program can require updates to several databases in several RMs in a single global transaction. The commitment of work in one transaction branch can be contingent on transaction branches occurring at other RMs. RMs are typically unaware of the work performed by other RMs. A *global transaction* includes more than one RM.

Figure 1-5 shows the X/Open version of Figure 1-3 on page 1-9. Both figures show several service requests that involve updates for three databases.

Figure 1-5
Routing an X/Open Global Transaction
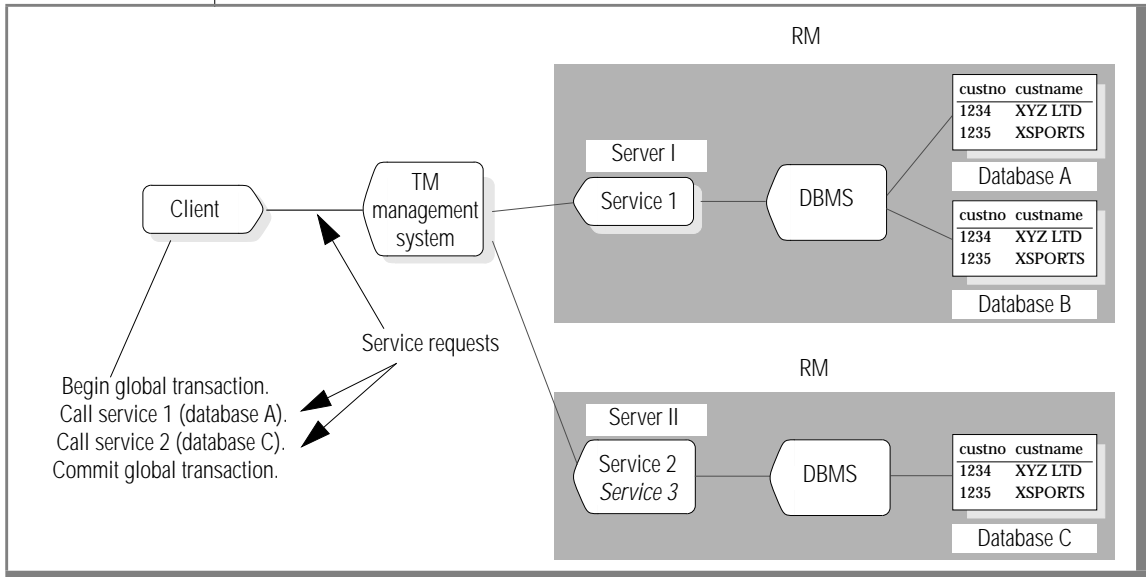
Figure 1-5 shows the roles of the RM and TM in an X/Open global transaction.The TM does the central coordination of global transactions. It coordinates the work performed by the RMs in their transaction branches. Figure 1-5 shows the global transaction has two transaction branches.

Assigning Transaction Identifiers

The TM uniquely identifies each transaction that it manages by assigning it an XID. Each XID identifies both a global transaction and a specific transaction branch, as follows:

- The part of the XID that uniquely identifies the global transaction is called a *global transaction identifier* (GTRID).

- The part of the XID that uniquely identifies the transaction branch is called a *branch qualifier*.

In the case of a global transaction, the TM assigns the same GTRID to all transaction branches associated with that transaction. The TM informs each participating RM of the existence, commitment, or abortion of the global transaction. The TM sends the GTRID to the RM so that the RM knows to which global transaction its transaction branch belongs. The RM might, in turn, translate this GTRID to its own internal XID while it works on the transaction branch.

Managing Client/Server Communication

As discussed on , a DTP application consists of a client program and at least one server program, which communicate by service requests. To coordinate the communication of the service requests, an X/Open DTP application can use the TM.

Figure 1-6 shows how the TM manages the routing of service requests.

Figure 1-6
TM Service-Request Routing
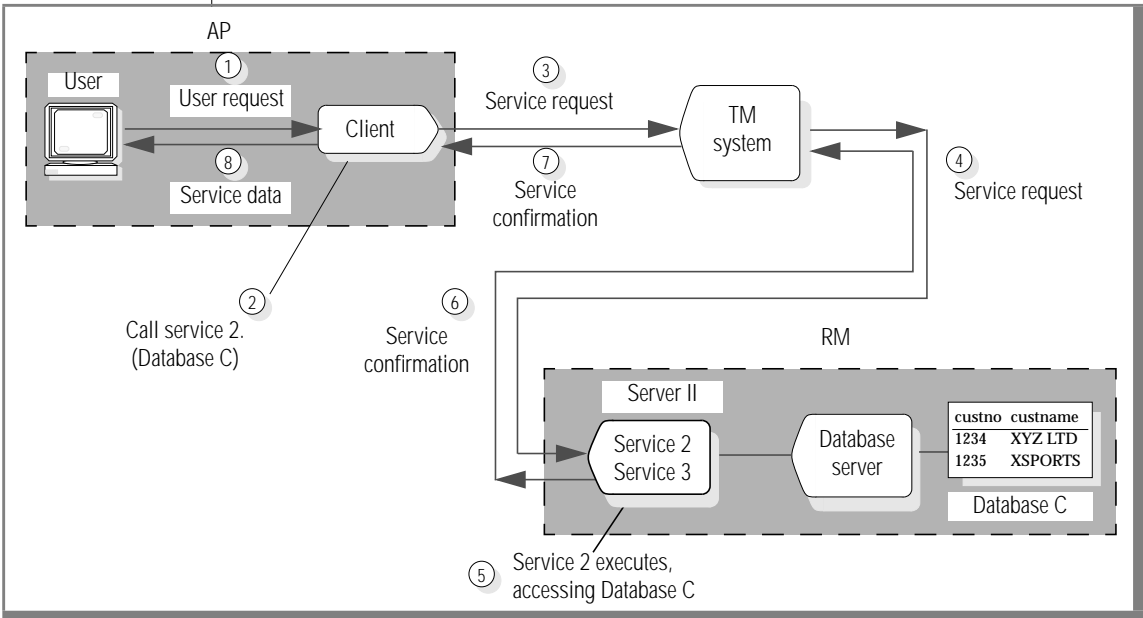
As shown in Figure 1-6 on page 1-18, the TM routes a service request in the following steps:

1. The user enters information needed for the transaction.

2. The client program creates a service request for Service 2.

3. The client program sends this service request to the TM.

4. The TM routes the service request to the server program of the appropriate RM. In this case, the TM routes the request to the RM that contains Server II, where Service 2 is located.

5. The server program, Server II, locates the Service 2 service and executes it. In the course of this execution, the Service 2 program accesses Database C through the associated DBMS.

6. The Server II server program sends the TM a confirmation that the service has executed. It then sends to the TM any data that the DBMS returned.

7. The TM routes the service confirmation to the appropriate client program (the application program). If the service has returned any data, the TM also routes this data to this client.

8. The client program (the application program) displays any data that is appropriate for the user.

To handle these service requests, the TM defines and manages a *request queue*, which stores information about the state of a transaction. The request queue, shown in Figure 1-7, is a piece of shared memory that the TM defines and manages.
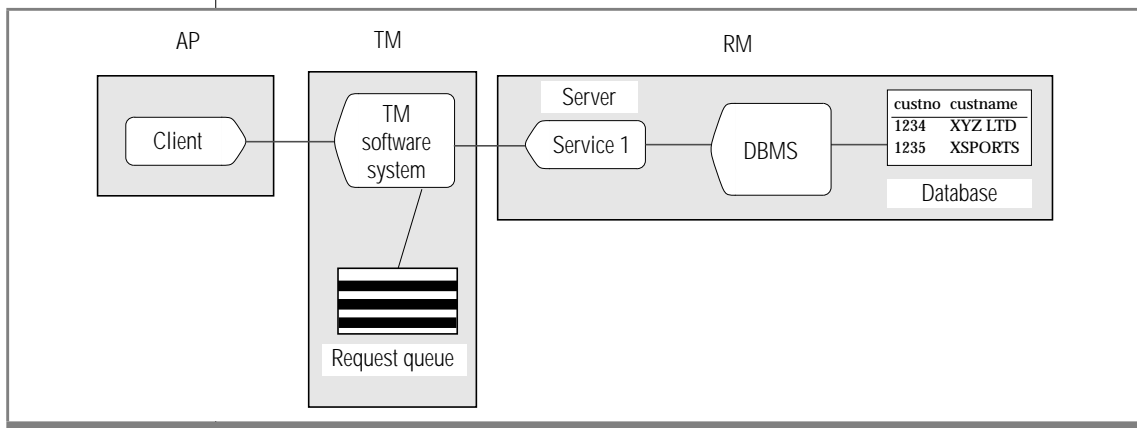
Important: *Because the design and implementation of TMs is not governed by any specification, your TM might not handle server requests as described in Figure 1-7.*

Controlling the Two-Phase Commit

In an X/Open DTP model, the *two-phase commit protocol* is handled by the following two parts:

- The TM is the coordinator, directing the execution of the global transactions.
- Each RM is a participant, directing the execution of one transaction branch.

For definitions of coordinators and participants, see page 1-10.

The TM controls the two-phase commit protocol in the following two phases.

Phase 1: The Precommit Phase

1.   The application program initiates the two-phase commit by notifying the TM that it wants to commit the global transaction.

2.   The TM asks each RM if it is prepared to commit its transaction branch. To do this, the TM calls the **xa_prepare()** XA interface routine.

3.   If the RM determines that it can commit the transaction branch, it records this information in a log and sends an affirmative reply to the TM. If the RM cannot commit the transaction branch, it aborts it and sends a negative reply to the TM.

Phase 2: The Postdecision Phase

1.   If the TM receives any negative replies or no reply, it asks the other RMs to abort their transaction branches by calling the **xa_rollback()** XA interface routine for each one.

2.   If all RM replies are affirmative, the TM first records the fact that it has decided to commit the global transaction along with a list of the involved RMs (excluding those that responded with a *read-only* status). It then asks each RM to commit its transaction branch by calling the **xa_commit()** XA interface routine for each one. The TM forgets about the transaction branch after all **xa_commit()** routines are complete.

For information on how the database server participates in the two-phase commit, see "The Database Server and the Two-Phase Commit Protocol" on page 2-11. For more information on the XA interface, see "The XA Interface" on page 1-24.

Before it can call any other XA routine, the TM must first initialize the RM by calling the **xa_open()** XA interface routine. When the communication finishes, the TM closes the channel by calling **xa_close()**. For more information on opening and closing RMs, see X/Open *Distributed TP: The XA Specification.*

Heuristic Decisions

If an RM aborts the work it is doing for a transaction branch during the precommit phase (phase one), the TM aborts the transaction branch during the postdecision phase (phase two). When the transaction branch is aborted, all RMs are in a consistent state.

However, an RM might make a *heuristic decision* during the postdecision phase. That is, an RM that is prepared to commit a transaction branch can decide to commit or abort its work *independently* of the TM. If this occurs, when the TM tells the RM to complete the transaction branch during the postdecision phase, the RM reports that the transaction branch was either committed or aborted.

When a participating RM makes a heuristic decision and reports its decision to the TM, the TM returns an error message to the application. The actual text of the error message is TM dependent. It is probably similar to one of the following error messages:

- Due to a heuristic decision, the work done for the specified GTRID was aborted.
- Due to some failure, the work done for the specified GTRID might have been heuristically completed.
- Due to a heuristic decision, the work done for the specified GTRID was partially committed and partially aborted.

In the first case, the global transaction was aborted. When the state that the RM reports matches the state that the TM requires, no problem exists because the transaction branch was completed (either aborted or committed by all RMs) successfully. This response means that the global system is still consistent, and no further problem exists.

In the second and third cases, however, the TM error messages indicate that the transaction state is unknown or mixed. (A mixed state can occur if the transaction is partially committed and partially aborted due to a heuristic decision made by an RM.) When the state that the RM reports does *not* match the state the TM wants, the global system is now inconsistent. The system administrator must bring the system back to a consistent state. Because several RMs can be involved in a transaction, the first task for the system administrator is to determine which RM made a heuristic decision. For more information on how to determine whether an RM has made a heuristic decision, see "The Database Server and Heuristic Decisions" on page 2-14.

The Model Interfaces

For two parts of the X/Open DTP model to communicate, they must use an *interface.* An interface is a series of functions linked into the sending and receiving programs so that each program can send and receive data. The X/Open DTP model has three paths of communication, which result in the following three interfaces:

- The *AP-to-RM interface* (shown in Figure 1-8 on page 1-27 as *AP-RM*) allows the application program to call an RM to request work that involves neither coordination of a global transaction nor management of the TM.

- The *AP-to-TM interface* (shown in Figure 1-8 on page 1-27 as *AP-TM*) lets the application program call the TM to request management of the transaction.

- The *XA interface* allows two-way communication between an RM and the TM. The XA interface implements the two-phase commit protocol (see page 1-20) between the RMs and the TM.

Figure 1-8 on page 1-27 shows the interfaces of the X/Open DTP model.

The AP-to-RM Interface

The AP-RM enables the application program to communicate with the RM. A library, called the *native* RM API, contains functions that these two programs use to communicate. Link this native RM API into your server program so that it can send database requests directly to the RM. The native RM API is also part of the database server so it can send and receive requests from the application program.

The server program uses the native RM API when it sends SQL statements to the database server. This protocol is independent of the underlying transport or network protocol. The preprocessor for Informix SQL API products automatically links the appropriate API libraries into the program.

The AP-to-TM Interface

The TM communicates with both clients and servers through the AP-TM interface (see Figure 1-4 on page 1-13). The TM provides these AP-TM routines in the form of a library that is linked to both client and server programs of the application. The AP-TM library supports assignment and prioritization of client service requests. It also manages transactions and buffers used for communication in global transactions.

Ideally, the proprietary AP-TM interface should adhere to a single specification, as TP/XA adheres to the X/Open XA specification. But currently the AP-TM for each TM vendor is unique. Therefore, to program in an X/Open environment, you need to embed the AP-TM calls specific to your vendor's TM in your application.

The XA Interface

The XA interface handles the communication *between* the RM and TM. The interface is a standard library of routines described in the XA specification. The TM uses these routines to manage global transactions. The names of these TP/XA routines, as defined by the XA specification, begin with the string **xa_**. The **xa_** routines must be supported by all RMs and the TM operating in the X/Open DTP environment. When a global transaction occurs, the TM, through the XA interface, ensures that the transaction meets all the requirements of the ACID test. (The ACID test requirements are discussed on page 1-4.)

To successfully meet these requirements, the TM uses the routines of the XA interface to accomplish the following tasks:

- Central coordination of global transactions

  When an application program calls a TM to start a global transaction, the TM uses the XA interface to inform RMs of their transaction branches.
- Transaction commitment and recovery using two-phase commit protocol

  After the application program uses the native RM API, through the AP-to-RM interface, to do work in support of the global transaction, the TM uses the XA interface to commit or abort branches.

The **xa_** routines of the XA interface are listed in Appendix A.

Software Products and the X/Open DTP Model

Although the database server supports a form of DTP, it does *not* provide support for the following types of DTP:

- Distributed transactions across database servers from other DBMS vendors (heterogeneous distributed transactions)

  Informix database servers supports distributed transactions when *all* the database servers are Informix database servers.

- The X/Open model for DTP

  The native DTP of the database server does *not* follow the X/Open DTP model.

Tip: For more information on the DTP *that is native to the database server, see the "Administrator's Guide" for your database server.*

To handle distributed transactions in either of these cases, you can use the following software products:

- Third-party TM software for managing heterogeneous global transactions

- Informix software for creating an RM that handles the service requests based on data managed by a database server

Third-Party TM Software

The TM software supervises global transactions that update databases on multiple systems, including databases from different vendors, as long as they support the XA specification. To be used in an X/Open DTP environment, the third-party TM software product must provide the following features:

- Software to provide TM functionality, as described in "The Transaction Manager" on page 1-15

   This software must also include support for the XA routines needed to support the TM side of the XA interface.
- An AP-TM library to be linked into both the client application program and the server program
- Link scripts that facilitate building client and server programs with access to the AP-TM library

For more information, refer to your TM documentation.

Informix Software for the RM

You need the following Informix products to create an RM in the X/Open DTP environment:

- The Informix database server serves as the DBMS system with which the server program communicates.
- The Informix SQL API products, ESQL/C and ESQL/COBOL, and the TP/XA library support creation of the server programs that access database server.

   INFORMIX-ESQL/COBOL is not available with INFORMIX-Universal Server. ♦

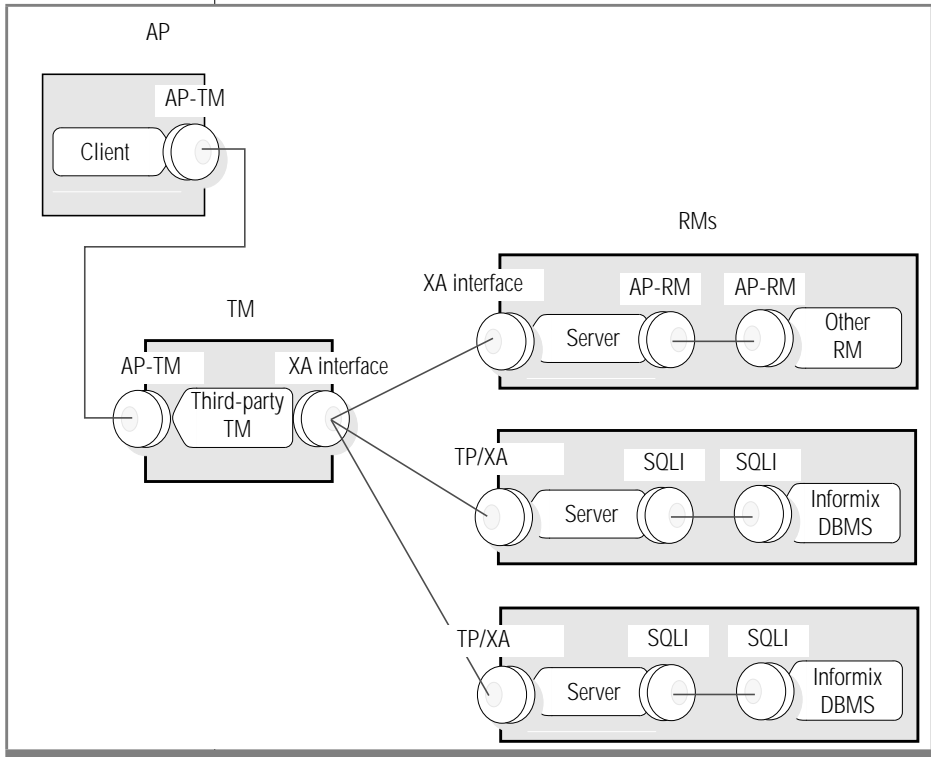**IUS**

Tip: The TP/XA *library is required only if the application uses global transactions.*

Figure 1-8 shows a sample X/Open DTP model that uses TP/XA and the Informix DBMS.

Figure 1-8
The Informix Solution for the X/Open DTP Model

The Informix Database Server as an RM

The Informix database server is a multithreaded database server that provides a flexible threading architecture for the on-line transaction-processing environment. You can use the database server as part of an RM within an X/Open DTP environment by performing the following actions:

- Define the database server to the TM system. For more information, see Chapter 2, "Integrating the Database Server and TP/XA into the X/Open DTP Model."
- Use embedded SQL statements from within the server program. For more information, see Chapter 3, "Programming in an X/Open Environment."
- Link the TP/XA library into your application program. For more information on the TP/XA library, see page 1-27.

The manner in which a TM performs its responsibilities is up to the individual TM software designer. As a result, the Informix database server does not know and cannot predict what tasks the TM is processing. An Informix database server RM knows *only* about the work it does for a transaction branch, whereas the TM might be processing many other tasks. In an XA environment, the database-server RM performs the following tasks:

- Responds to the XA requests that it receives from the TM
- Tracks GTRIDs for the TM

Important: In the Informix implementation of transaction branches, each branch is treated as a separate transaction. No two transaction branches, even if they belong to the same transaction, can share locks. For more information on locking, see the "Administrator's Guide" for your database server.

The TP/XA Library as the Server XA Interface

TP/XA is a library of functions that you link to your SQL API server program. This library has the XA routines that must be present for the server program (in the RM) to communicate with the TM. The TM uses these XA routines to communicate global-transaction information to the server program, which, in turn, communicates the information to the database server.

Through the TP/XA library, a server program and the database server can act as an RM with any TM that conforms to X/Open XA specifications.

The TP/XA library fully supports the required XA interface. However, it does *not* support the following optional XA interface features:

- Asynchronous operations
- Dynamic registration
- Transaction association migration

Tip: For information on these optional features, refer to "Distributed TP: The XA Specification" (February 1992) from the X/Open Company Limited.

What TP/XA Can Do for You

Using the TP/XA library, the Informix SQL API product, third-party TM software, and your Informix database server in your OLTP applications provides the following advantages:

- Distributed heterogeneous transactions

  The TM software supervises global transactions that update databases on multiple systems, including databases from different vendors, as long as they support the XA specification.

- Tunable response times

  With a TM, you can balance the workload among servers. With the database server, you can tune the locking, data buffers, and other performance factors within the database system. For more information in tuning the database server, see the *Performance Guide* for your database server.

- High availability

  The TM software usually has several built-in features to enhance system availability. For example, when the transaction manager TUXEDO System/T detects that a database server aborted abnormally, System/T creates a new instance of the failed database server and sends a message to the client. The high-availability features of the Informix database server include on-line archiving, incremental archiving, mirroring, and automatic fast recovery.

Integrating the Database Server and TP/XA into the X/Open DTP Model

In an X/Open distributed transaction processing (DTP) environment, the Informix database server acts as a database management system (DBMS) in an RM. When you use the database server in an X/Open DTP environment, you must attend to additional setup requirements as well as database administration and configuration issues.

This chapter covers the following topics:

- Installing software for an X/Open DTP environment
- Monitoring global transactions
- Administering the database server in the event of an aborted transaction

This chapter assumes that you are familiar with the information in the *Administrator's Guide* for your database server.

Installing Software for an X/Open DTP Environment

To use the XA interface defined in the TP/XA library as an interface between the database server and your transaction manager, you must install the following products:

- A TM that supports the X/Open specification as described in the document *"Distributed TP: The XA Specification"* (February 1992) from the X/Open Company Limited
- The Informix database server
- One of the following Informix SQL APIs
  - ❏ INFORMIX-ESQL/C
  - ❏ INFORMIX-ESQL/COBOL

    INFORMIX-ESQL/COBOL is not available with INFORMIX-Universal Server. ♦

IUS

Important: *The* TP/XA *library,* **libinfxxa.a**, *is part of Informix SQL API products. When you install one of these products, you automatically install this Informix XA library.*

Installing the Transaction Manager

Although Informix implements the *database portion* of the XA standard, the TM configures, coordinates, and controls the X/Open DTP environment. The individual TM software supplier configures and manages the system. You *cannot* use the database server in an X/Open DTP environment if the TM software is *not* installed and configured properly.

Important: *For information on how to install and set up your TM product, refer to your TM software documentation.*

The TM needs information about the database server to establish its relationship with the database server RM. For a list of the information that you must provide to the TM, see "Integrating the Database Server with the TM" on page 2-5.

Installing the Informix Software

To establish the database server as part of an RM in your X/Open DTP environment, you must install two Informix products:

- An Informix database server
- One of the following Informix SQL API products:
    - ❑ INFORMIX-ESQL/C
    - ❑ INFORMIX-ESQL/COBOL

        INFORMIX-ESQL/COBOL is not available with INFORMIX-Universal Server. ♦

IUS

For instructions on how to install the database server and the SQL API, refer to the *Installation Guide* for those products.

Keep the following compatibility issues in mind when you use the TP/XA library:

- This library is installed as part of the SQL API.

- This library is *not* compatible with versions of TP/XA earlier than 6.0. It *is* compatible with versions later than Version 6.0.

- This library is supported for *both* the Micro Focus (MF) COBOL/2 and Ryan-McFarland (Liant) RM/COBOL-85 compiler.

  INFORMIX-ESQL/COBOL is not available with INFORMIX-Universal Server. ♦

- To run applications built with ESQL/C or ESQL/COBOL, Version 5.0, you need to recompile and link your Version 5.0 application with files that are distributed with the TP/XA library. For details, see "Building Servers for an X/Open DTP Environment" on page 3-12.

IUS

Integrating the Database Server with the TM

Once you install the Informix software, the database server must be integrated as an RM in your X/Open DTP environment. To do this, you must perform the following tasks:

- Provide information describing the RM to the TM
- Set up database logging

Describing the Database Server RM

For the TM to be able to integrate the database server as part of an RM, you must provide the TM with the information shown in Figure 2-1.

Information That the TM Needs to Work with a Database Server RM

Information TM Needs	Information You Must Provide	For More Information
XA Switch Table Name	**infx_xa_switch**	"The Switch Table"
RM Name	Name of Informix database server. For example, INFORMIX-Universal Server	"The RM Name" on page 2-7
XA Routine Library Name	**libinfxxa.a**	"The XA Routine Library" on page 2-7
Open String	Name of database to open	"The Open and Close Strings" on page 2-7
Close String	' ' (null string—a string in which the first character is null)	

For information on *how* to provide this information to the TM, refer to your TM documentation.

The Switch Table

The XA interface defines a structure called the *switch table*, which lists the names of the **xa_** routines as they are implemented in the RM. (For more information on the **xa_** routines, see "The XA Interface" on page 1-24.) In the XA interface, this structure is called **xa_switch_t** and it is defined in the **xa.h** header file. To be integrated into the X/Open DTP model, each RM must identify the name of its switch table so that the TM can find the names of the **xa_** routines.

The name of the database server switch table is **infx_xa_switch**. You must provide this switch table name to the TM so that it can locate the database server **xa_** routines. This switch table is defined in the database server XA library and in the TP/XA library.

The RM Name

The *RM Name* is a string that identifies the RM to the TM. This string is stored within the switch table in the **name** field. When you use the database server as the DBMS of an RM, initialize this field to the name of your Informix database server. For example, you would use the following string to specify that INFORMIX-Universal Server is your database server:

```
INFORMIX-Universal Server
```

The Open and Close Strings

The *open string* and *close string* are the text passed as an argument to the **xa_open()** and **xa_close()** XA interface routines, respectively. The TM calls the **xa_open()** routine to initialize an RM for use in the DTP environment. The open string contains any specific information that the RM needs. The open string is limited to 256 characters and contains no blanks or line feeds. When you use the database server as part of an RM, the open string specifies the name of the database that a particular database server instance can open.

The TM calls the **xa_close()** routine to close a currently open RM. Once closed, the RM cannot participate in global transactions until it is reopened. The close string contains any information that is specific to the RM. However, the database server does not require any close information. Therefore, you must initialize the close string to a null string.

For more information on how to use the **xa_open()** and **xa_close()** routines, refer to *Distributed TP: The XA Specification*.

The XA Routine Library

The *XA Routine Library* is the name of the library that contains the **xa_** routines and the switch table defined by the database server RM. This library is called **libinfxxa.a** and is one of the libraries that the TM must link to the server process of an application program (AP).

Database Logging in an X/Open DTP Environment

Databases in the X/Open DTP environment must use *unbuffered logging*. Unbuffered logging ensures that the database server logical logs are always in a consistent state and can be synchronized with the TM. If a database created with buffered logging is opened in an X/Open DTP environment, the database status automatically changes to unbuffered logging. The database server supports ANSI-compliant databases as well as databases that are not ANSI compliant.

Monitoring Global Transactions

During execution of your AP, you can track the status of global transactions handled by the database server with the **onstat** utility. This utility is described in the *Administrator's Guide*. The information in this section describes specific flag settings that indicate the status of XA-related global transactions.

The following two sections of **onstat** output are useful for monitoring global transactions:

- The Userthreads section is generated by the -**u** option.
- The Transactions section is generated by the -**x** option.

Within an X/Open DTP environment, the following relationships among users, transactions, and locks must exist:

- Locks are owned by a transaction branch.
- Transactions can be associated with an RM (database server) thread.
- A transaction can exist without being associated with an RM (database server) thread.

For more information on how the database server performs locking, refer to the *Administrator's Guide* for your database server.

The Userthreads Section

To generate the Userthreads section output, use the -**u** option of the **onstat** utility. This option produces a profile of user activity, and it refers to the actual database threads. Figure 2-2 shows the headers of output information that appear when you use the -**u** option of **onstat**.

```
RSAM Version 9.10.UC1-- On-Line -- Up 00:06:16 -- 528 Kbytes

Userthreads
address  flags sessid  user  tty   wait   tout locks nreads   nwrites
```
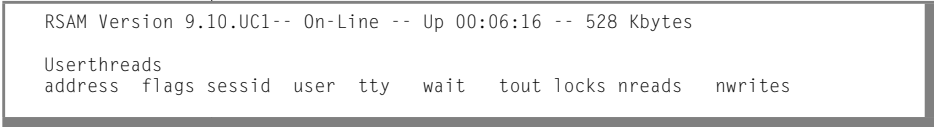
Figure 2-2
onstat -u Output

The flags column in the Userthreads section refers to the status of a thread. The following table describes the XA-related flags.

Position	Code	Description
1	T	Waiting for a transaction.
		In an X/Open DTP environment, multiple database server threads can access the same transaction, but not simultaneously. If a request is made for a service for a global transaction, but the transaction is busy in another service, the first request must finish and detach from the transaction. In the meantime, the second thread blocks, waiting for the transaction.
		This situation could occur if two different services, both using the same database server and database, tried to work on the same global transaction simultaneously.
3	X	Transaction is XA-prepared (the database server is prepared to commit) or is currently in the process of doing so.

If a transaction is associated with a thread, the state of the transaction is shown by the flags column in the Userthreads section. For more information on the -**u** option of **onstat**, refer to the *Administrator's Guide* for your database server.

The Transactions Section

To generate the `Transactions` section output, use the **-x** option of the **onstat** utility. This option produces information specific to the X/Open environment. The output describes the state of the transaction. Figure 2-3 shows the headers of output information that appear when you use the **-x** option of **onstat**.

```
RSAM Version 9.10.UC1 -- On-Line -- Up 00:06:16 -- 528 Kbytes

Transactions
address  flags user    locks log begin isolation retrys coordinator
```

Figure 2-3
onstat -x Output

The `flags` column in the `Transactions` section refers to the status of a transaction. The following table describes the XA-related flags.

Position	Code	Description
1	A	Transaction is owned by a user thread.
1	S	A global transaction is suspended. A suspension occurs when a user thread is no longer associated with a transaction, though most likely it will be associated again. The suspension ensures that the transaction branch does not reach the precommit phase.
1	C	The TM is waiting for a rollback to be performed.
3	X	Prepare state. This transaction is prepared to commit for XA.
5	G	A global transaction is in effect.

For transactions that are attached to a thread, the transaction state flags (position 2) are the same in the `Userthreads` (**-u** option) and `Transactions` sections (**-x** option). If a transaction appears in the `Transactions` section but not in the `Userthreads` section, the transaction is detached from a thread.

Transaction Commitment and Recovery

In the X/Open DTP environment, global transactions are managed by the TM. Because a global transaction spans more than one RM, you can have database server RMs and other RMs working together on a global transaction. The TM tracks which RMs are involved in the transaction. Although the TM *manages* the start, end, and recovery process of a global transaction, the RM *performs* the actual work commitment by managing its transaction branch. That is, when the TM tells the RM to commit, the RM commits the work that it performed for the transaction branch.

The Database Server and the Two-Phase Commit Protocol

Consider a global transaction that consists of a withdrawal from a savings database and a deposit into another savings database managed by another instance of the database server. Because neither database server RM knows about the other, the TM must ensure that the same decision, either to commit or abort the transaction, applies to both RMs.

RM product vendors decide how their RMs handle transaction commitment and recovery. The following sections focus on how the database server interacts with the TM during transaction commitment and recovery.

When the database server acts as part of an RM in an X/Open DTP environment, it relies on the following elements:

- Communication

   Communication among the TM and participating database server RMs occurs through the XA routines that control the two-phase commit protocol.

- Logical-log records

   Logical-log records of the transaction must be stored on stable media to ensure data integrity and consistency if a failure occurs at a participating database server RM.

- Transaction information

    Transaction information must be stored in shared memory at each
    participating database server RM. This requirement is important for
    tracking open tblspaces, acquired locks, and other resources required
   by the database server RM that works for a global transaction.

When the application sends a commit message to the TM, it initiates the
following actions in the two-phase commit protocol:

- The *precommit* phase begins after all inserts, updates, and deletes
    included in the global transaction are executed. During this phase,
    the TM asks the database server RM to prepare to commit its trans-
    action branch by calling the **xa_prepare()** XA interface routine.

    - If the database server determines that it *can* commit the trans-
        action branch, it records this information in a logical-log record
        (XAPREPARE log record) and notifies the TM that the commit
        succeeded.

    - If the database server *cannot* commit the transaction branch, it
        aborts the branch, records this information in a logical-log record
        (ROLLBACK log record), and notifies the TM that the commit
        failed.

- During the *postdecision* phase, the TM commits or aborts the global
    transaction based on the RM responses from the precommit phase
    (phase one).

    - If all replies are affirmative, the TM asks each RM to commit its
        branch of the global transaction by calling the **xa_commit()** XA
        interface routine for each RM. When this routine is called, the
        database server RM writes a COMMIT log record to a logical log.

    - If at least one negative reply exists, the TM aborts the transaction
        and then asks each RM to abort its transaction branch by calling
        the **xa_rollback()** XA interface routine. When this routine is
        called, the database server RM writes a ROLLBACK log record to
        a logical log.

Figure 2-4 shows the flow of information between the TM and the database server.

Figure 2-4
Flow of Information Between the TM and Database Server During a Two-Phase Commit
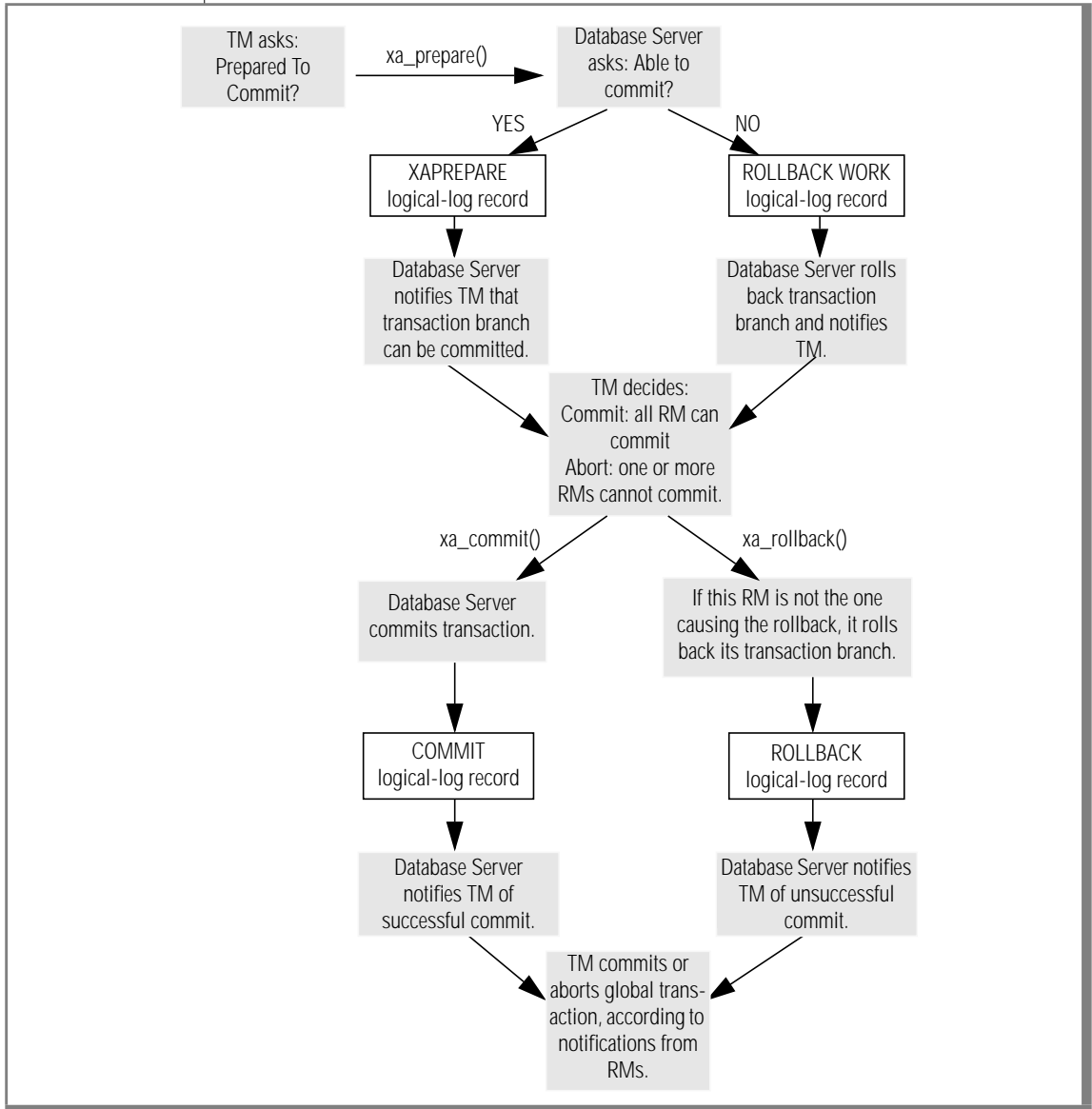
The Database Server and Heuristic Decisions

When the database server RM makes a *heuristic decision* and aborts the transaction branch during the commit phase independently of the TM instructions to the database server RM, your global system might be in an inconsistent state. (For general information on heuristic decisions, see page 1-22.)

A heuristic decision does not, in itself, create a problem for the two-phase commit protocol. The decision by the database server to roll back a transaction branch becomes a problem only when *both* of the following conditions are true:

- The participating database server RM makes the heuristic decision to roll back its transaction branch *after* the TM receives notification that this RM can commit its branch.
- The TM decides to commit the global transaction and instructs all participating RMs to commit their transaction branches.

If a database server RM participating in a global transaction heuristically rolls back a transaction branch and *both* of these conditions are true, the global transaction-processing system is in an inconsistent state. That is, some participating RMs committed their transaction branches while at least one database server RM aborted its transaction branch.

In this case, the TM error messages indicate that the transaction state is unknown or mixed (partially committed and partially aborted). The system administrator must decide what actions should be taken to return the system to a consistent state.

The administrator must first determine which RM made a heuristic decision. If this RM uses a database server, the administrator needs to know the answers to the following questions:

- Why would a database server RM make a heuristic decision?
- How do you determine whether an Informix database contains inconsistent data?
- What actions do you need to perform to bring an Informix database back to a consistent state?

Causing the Database Server to Make a Heuristic Decision

A likely cause of a heuristic decision by a database server RM is a *long transaction* (LTX). An LTX occurs when the logical log fills to the point defined by one of the long-transaction high-water marks (configuration-file parameters LTXHWM or LTXEHWM). The source of an LTX condition is work being performed for a global transaction. For more information about long transactions, see the *Administrator's Guide* for your database server.

Determining Consistency of an Informix Database

If database inconsistency is possible because of a heuristic decision by a database server RM, check the logical logs for the following combination of logical-log records:

- The XAPREPARE logical-log record, seen only in an X/Open DTP environment, indicates the ability of the database server RM to commit the transaction branch, when the TM instructs it to do so.

- The HEURTX logical-log record indicates a heuristic decision that the database server RM made to abort its transaction branch.

- The ROLLBACK logical-log record indicates that the database server RM aborted its transaction branch.

For more information about logical-log records and heuristic decisions, see the *Administrator's Guide* for your database server.

Taking Actions to Handle Database Inconsistency

If you find that your Informix database is in an inconsistent state, you have the following options:

- Leave the database in its inconsistent state

- Recover from the inconsistent state

As you consider your options, remember that no automatic process or utility can perform a rollback of a committed transaction or can commit part of a transaction that is rolled back. Although the database server logical-log records can show the affected transaction, you cannot determine what work transpired from the messages alone. It is your responsibility, based on your knowledge of your application and global system, to determine which option to take.

You might consider the other RMs involved in this particular global transaction and the TM to make this decision. Recovery in an X/Open DTP system not only requires knowledge of the database server RM but also knowledge of how each independent software component is designed to recover from a database inconsistency.

Leaving the Database in an Inconsistent State

You might decide to leave the database in its inconsistent state if the transaction does not significantly affect database data. This situation occurs if you decide that the application can remain inconsistent because the price (in time and effort) of returning the database to a consistent state (by either rolling back or committing the transaction) is too high.

Recovering from a Database Inconsistency

If you cannot leave the database in an inconsistent state, use the database server logical log to determine how you want to recover from the inconsistent state. You must determine which of the following actions to take for the transaction in question:

- Roll back (abort) the effects of the global transaction wherever it was committed
- Commit the effects of the global transaction wherever it was rolled back

To determine which of these actions to take, perform the following steps:

1. Obtain the local XID from the HEURTX logical-log record at the database server RM where the transaction rolled back.

   The transaction that rolled back has a HEURTX record associated with it. The HEURTX record contains the local transaction identification number (local XID). You can use the local XID to locate all associated log records that rolled back as part of this piece of work.

2. Look for an XAPREPARE logical-log record for the local XID and obtain the GTRID.

3. At each possible participating database server RM site, search logical logs for the GTRID associated with the heuristic completion.

4. Use the records and your knowledge of the application to construct a compensating transaction that either rolls back the committed effects of the transaction or commits the work that was rolled back.

Tip: TM logging information is probably the best source of information when determining which RMs participated in a global transaction. For information about how to recover from a mixed or unknown transaction state, refer to your TM documentation.

Programming in an X/Open Environment

To help you create an application program for the X/Open DTP environment, this chapter covers the following topics:

- Preparing to program in an X/Open DTP environment
- Developing client and server programs in an X/Open DTP environment

This chapter assumes that you are familiar with the information in the *Informix Guide to SQL: Syntax* and one of the following SQL API manuals:

- *INFORMIX-ESQL/C Programmer's Manual*
- *INFORMIX-ESQL/COBOL Programmer's Manual*

IUS

    INFORMIX-ESQL/COBOL is not available with INFORMIX-Universal Server. ♦

Preparing to Program in an X/Open DTP Environment

Application programming in an X/Open DTP environment is not too different from application programming in any other environment. Many of the Informix embedded-language statements that you use in programs designed for a non-X/Open DTP environment remain unchanged. In an X/Open DTP environment, however, you must make some programming adjustments.

Designing Programs for an X/Open DTP Environment

When you design programs for an X/Open DTP environment, you must create an application that includes client programs, services, and server programs:

- A *client program* takes user input and sends it in the form of a service request to a server program. The client program accesses the services that a server offers. Most importantly, the client program interfaces with the user. For example, a client program might request data from the user as inputs to a program, or it might return data to the user after inputs are processed.

  In an Informix X/Open DTP environment, a client might be a C program that gets input from the user and marks transaction boundaries using calls to the TM through the AP-TM library. It can also use AP-TM calls to send server requests to a server program.

- A *server* is a program that provides one or more *services.* A server receives requests from a client and initiates execution of the appropriate service. A service could perform a single database task (one service) or many database tasks (many services).

  In an Informix X/Open DTP environment, to create a server you can write a module of services in an SQL API, such as ESQL/C, and group related services into a module, the server program.

- A *service* is a module of SQL API code that performs some database task for the application. A service typically accesses a database to perform queries or update information. The services can communicate with the TM (through calls to the AP-TM) to mark transaction boundaries and to return messages to the client program. They also communicate with the RM (through SQL statements) to access database information.

  In an Informix X/Open DTP environment, to create a service, you can write a function with one of the Informix SQL APIs. For example, the service could be a program that is designed to record deposits into bank accounts. The service could access an Informix database through a database server RM and update a column in an **accounts** table.

For more information on clients, servers, and services, see "Client and Server Programs" on page 1-6 and "Managing Client/Server Communication" on page 1-18. For information on how to build server processes, see "Building Servers for an X/Open DTP Environment" on page 3-12.

Identifying the Transaction Mode

Your programming adjustments depend largely on the transaction mode in which your program or service is designed to execute. A transaction mode can be either global or local. Once you know the transaction mode, you can write a program that is designed to execute within a global transaction, a local transaction, or both.

Tip: Regardless of the transaction mode used, the service program does not need to include the XA interface routines. These routines are used only by the TM to communicate with the RMs.

The next sections describe each type of transaction.

Local Transactions

A *local transaction* accesses a single database and a single service. Local transactions do *not* invoke the XA interface. A program written for an Informix transaction-processing environment uses local transactions, either implicit or explicit. This type of program (if it is to continue using local transactions), involves little change to run in an X/Open DTP environment. If you choose to use the TM to handle client/server communication (page 1-18), you must add the appropriate message routines from the AP-TM library to the server program.

The two types of local transactions are *implicit* and *explicit.*

Implicit Local Transactions

The database server RM defines an *implicit local transaction* for each SQL statement that modifies the database but is not preceded by a BEGIN WORK statement and followed by a COMMIT WORK statement. For example, the following statement is an implicit local transaction:

```
INSERT INTO manufact VALUES ('BBS', 'Big Boy Sports')
```

In this case, if this single SQL statement is successful, the database server RM commits the transaction, saving the new **manufact** row. If this INSERT fails, the database server RM rolls back the transaction and the new **manufact** row is *not* saved.

Explicit Local Transactions

An explicit local transaction can be executed under the control of either of the following managers:

- The RM

  An explicit local transaction is an SQL statement (or set of statements) preceded by a BEGIN WORK statement and followed by a COMMIT WORK statement. These SQL statements mark transaction boundaries. For example, the following SQL statements are considered to be a single explicit local transaction:

  ```
  BEGIN WORK
  LOCK TABLE stock
  UPDATE stock SET unit_price = unit_price * 1.10
      WHERE manu_code = 'KAR'
  DELETE FROM stock WHERE description = 'baseball bat'
  COMMIT WORK
  ```

- The TM

  An explicit local transaction is identical to a transaction under the control of an RM except that the application makes a call to the transaction manager to begin and commit the work. For the specific statements to begin and commit a local transaction under the control of your TM, consult your TM manual.

Global Transactions

A *global transaction* is a transaction that can span more than one service, database server, and RM. Global transactions are identified and controlled by the TM, which uses the XA interface to communicate with each participating RM. When you write a program that has the potential to execute as part of a global transaction, it must adhere to certain guidelines to execute successfully in an X/Open DTP environment.

A program written for an Informix transaction-processing environment does require some modification to use global transactions in an X/Open DTP environment. It must contain the appropriate AP-TM calls to perform the following tasks:

- Mark the start and end of the global transaction
- Handle client/server communication

However, you do not need to modify the program to use the XA interface because the XA routines that allow the RM and servers to communicate are never called directly by the server program.

Writing Server Programs for an X/Open DTP Environment

This section contains the following information about writing server programs:

- Programming considerations when you use SQL within an X/Open DTP application program
- Steps for building a server program so that it links in the appropriate XA, AP-TM, and AP-RM interfaces
- Sample ESQL/C programs that demonstrate the changes needed to convert a non-DTP ESQL/C program to one for an X/Open DTP environment

  You make similar changes to tailor an existing ESQL/COBOL program for an X/Open DTP environment. ♦

  INFORMIX-ESQL/COBOL is not available with INFORMIX-Universal Server. ♦

ODS

OWS

IUS

Programming Considerations for Server Programs

In general, programming a server program involves creating ESQL programs, or services, to access a specific database server. The following sections list the differences of which you should be aware of when you use Informix products in an X/Open DTP environment.

Using Cursors

Database cursors can be used in *any* transaction mode. However, you must declare, open, and close the cursor within a *single service*. In programming terms, a single service can be defined as follows:

- A single service *starts* with a call to the **xa_start()** XA interface routine, which passes a TMJOIN flag or no flag at all (TMNOFLAGS).

- A single service *ends* with a call to the **xa_end()** XA interface routine, which passes either the TMSUCCESS or TMFAIL parameters (but not TMSUSPEND). This allows you to have any number of **xa_end**(TMSUSPEND) and **xa_start**(TMRESUME**)** calls in a program provided you delimit them with an **xa_start()** and **xa_end()**.

    In addition, you are guaranteed that any cursors or temporary tables associated with the single service can survive until you call **xa_end()** (with TMSUCCESS or TMFAIL) to end the service.

Once a service is exited, any cursors in that service cannot be used. If you attempt to use a cursor in a service other than the one in which it was declared, the database server returns an error.

For details about XA interface routines, see *Distributed TP: The XA Specification* (February 1992) from X/Open Company Limited.

Using Temporary Tables

Temporary tables can be used in *any* transaction mode. However, you must create and use the temporary table within a *single service*. The temporary table is dropped when you exit from a service.

Using Locking

You can use locking in *any* transaction mode. However, an important restriction exists on locking. Informix implements *each* transaction branch as a separate transaction. Therefore, no two transaction branches can share locks. This restriction applies even when the transaction branches belong to the same global transaction.

For more information on how the database server performs locking, refer to the *Administrator's Guide* for your database server.

Using SQL Statements

You can use almost all SQL statements in an X/Open DTP environment in either local or global transaction mode. However, some statements might behave differently than if they were in a non-X/Open DTP environment (see Figure 3-1), or they might return an error when they are involved in a global transaction (see Figure 3-2 on page 3-11).

Figure 3-1
Behavior of SQL Statements in an X/Open DTP Environment

Statement	Special Behavior
CLOSE DATABASE	If you issue a CLOSE DATABASE statement in an X/Open DTP environment, you receive an error.
CONNECT	The current database for a group of servers is set by the open string. If you issue a CONNECT statement in an X/Open DTP environment, you receive an error.
CREATE DATABASE	If you issue a CREATE DATABASE statement in an X/Open DTP environment, you receive an error.
DATABASE	The current database for a group of servers is set by the open string. If you issue a DATABASE statement in an X/Open DTP environment, you receive an error.
DISCONNECT	If you issue a DISCONNECT statement in an X/Open DTP environment, you receive an error.
LOCK TABLE	If you issue a LOCK TABLE statement within a global transaction, it remains in effect until the completion of the transaction branch.

(1 of 2)

Statement	Special Behavior
SET CONNECTION	If you issue a SET CONNECTION statement in an X/Open DTP environment, you receive an error.
SET ISOLATION	The current isolation level of a database remains in effect unless you change the isolation level from within a service. When you change an isolation level, the change remains in effect until the next modification or until the transaction branch ends.
SET EXPLAIN	The default setting for SET EXPLAIN is off. Once you issue a SET EXPLAIN ON statement, all subsequent query access procedures are written to a file, **sqexplain.out**, until the service ends or a SET EXPLAIN OFF statement is issued. The **sqexplain.out** file is stored in the current directory where the database server is running.
SET LOCK MODE	The current lock mode and time-out period remain constant throughout a service unless it is modified in the service. If you change the lock mode, the change remains in effect until the next modification or until the service ends. The SET LOCK MODE statement takes effect only when the time-out period is shorter than the period that the TM software specifies.
SET LOG	You must create databases in an X/Open DTP environment with unbuffered logging. If you create a database with buffered logging, the database status automatically changes to unbuffered logging when the database is opened in an X/Open DTP environment. If you issue a SET LOG statement in an X/Open DTP environment, you receive an error.
UNLOCK TABLE	If you issue an UNLOCK TABLE statement from within any transaction branch, you receive an error.
BEGIN WORK	If you issue a BEGIN WORK statement within a global transaction, you receive an error.
COMMIT WORK	If you issue a COMMIT WORK statement within a global transaction, you receive an error.
ROLLBACK WORK	If you issue a ROLLBACK WORK statement within a global transaction, you receive an error.

(2 of 2)

Figure 3-2 summarizes the return behavior of the statements listed in
Figure 3-1 on page 3-9 from an X/Open DTP global or local transaction.

Figure 3-2
Return Behavior of SQL Statements in Local and Global Transactions

Statement Local Transaction	Within a Global Transaction	Within a Local Transaction
BEGIN WORK	Returns an error	Executes
CLOSE DATABASE	Returns an error	Returns an error
COMMIT WORK	Returns an error	Executes
CREATE DATABASE	Returns an error	Returns an error
DATABASE	Returns an error	Returns an error
LOCK TABLE	Statement remains in effect until the end of the global transaction. All locks are released at the end of the transaction.	All table locks are released at the end of the global transaction.
ROLLBACK WORK	Returns an error	Executes
SET EXPLAIN	The file **sqexplain.out** is stored in the current directory on the computer on which the server is running.	The file **sqexplain.out** is stored in your home directory.
SET ISOLATION	If you change the isolation level within a service, the changes remain in effect until the next change or until the service ends.	If you change the isolation level, the change remains in effect until the next change or until the program ends.
SET LOCK MODE	SET LOCK MODE takes effect only when the time-out period is shorter than the period specified by the TM software.	SET LOCK MODE takes effect when issued.
SET LOG	Returns an error	Returns an error
UNLOCK TABLE	Returns an error	Returns an error

For a full description of the SQL statements listed in , refer to the *Informix Guide to SQL: Syntax*.

Building Servers for an X/Open DTP Environment

You can build a server program in one of the following ways:

- Use the commands and options provided by your TM product to create a server for an X/Open DTP environment. For instructions on how to link the TP/XA and AP-TM libraries to your server program, refer to your TM documentation. Also refer to the TP/XA documentation notes.

E/C

- You can build an ESQL/C server program manually using the ESQL/C preprocessor, **esql**.

This section describes the second method of building a server process. To build an ESQL/C server process manually, you must first preprocess the server program and then link the object (**.o**) files with the TP/XA library, as follows:

1. Execute the **esql** command with the -**c** command-line option, as the following example shows:

    ```
    esql -c srvrprog.ec
    ```

    The -**c** option is not recognized by **esql**, so it is passed through to the C compiler (**cc**, by default). This option suppresses the link phase of the compilation.

2. Use the -**libs** option to obtain the list of Informix API libraries that **esql** links to create an ESQL/C program, as the following example shows:

    ```
    esql -libs
    ```

3. Link the object file created in step 1 with the TP/XA library called **$INFORMIXDIR/lib/esql/libinfxxa**, as the following example shows:

    ```
    cc -o srvrprog srvrprog.o \
        $INFORMIXDIR/lib/esql/libinfxxa.a <LIBS>
    ```

    In the preceding **cc** command, *<LIBS>* represents the other ESQL/C libraries required to create an executable ESQL/C program. You obtained this list of libraries in step 2.

On some computers, you can preprocess the server program and link it in one step, as the following example shows:

```
esql srvrprog.ec -o srvrprog $INFORMIXDIR/lib/esql/libinfxxa.a
```

Important: *If you are using ESQL/C, Version 5.0 or earlier, you must link the object file* **ixacursor.o** *(in $INFORMIXDIR/lib/esql) before the* **libinfxxa.a** *TP/XA library.*

The libraries linked with the server program provide the following interfaces for the server:

- The **$INFORMIXDIR/lib/esql/libinfxxa.a** library provides the server program with the Informix implementation of the XA interface.
- The *<LIBS>* libraries provide the server program with the native AP-RM interface so that the server can communicate with the database server RM. ♦

Sample ESQL/C Programs

The actual programming changes that you make when you create programs with INFORMIX-ESQL/C in an X/Open DTP environment are minor. However, the way that you design the program is quite different than the way that you would design it for a non-X/Open DTP environment. For more information, see "Designing Programs for an X/Open DTP Environment" on page 3-4.

This section takes an ESQL/C program and shows one way to redesign the program for an X/Open DTP environment. It contains the following two ESQL/C programs:

- A non-DTP ESQL/C program that performs both user interactions and database interactions
- An ESQL/C AP for the X/Open DTP environment that includes two programs, a client and a server

Each program lets you update the unit price of a product for a chosen manufacturer using the **stores7** database. (For information about the **stores7** database, refer to the *Informix Guide to SQL: Reference*.)

A Non-DTP ESQL/C Program

The **upstock.ec** program is a small *non-DTP application*. It is not divided into a client and server process; instead, it performs user and database interaction in the same ESQL/C program. The program prompts the user for the manufacturer code, the stock number, and the percentage that the user wants to increase the unit price of the product. With this information, it takes the following actions:

- Retrieves the row from the **stock** table that matches the specified manufacturer code and stock number
- Displays the stock number, description, and unit price of the stock item
- Calculates the new unit price for the stock item
- Asks the user whether it should update the unit price

Figure 3-3 shows the **upstock.ec** program.

Figure 3-3
Sample ESQL/C Program That Updates the Unit Price in the stock Table

```
/*

 * upstock.ec *

 The following program fetches rows from the stock table for a chosen
 manufacturer and allows the user to selectively update the unit_price by a
 specified percent. The program prompts the user for the manufacturer code
 and stock number, and then the percent of the price increase.
*/

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

EXEC SQL include sqltypes.h;

#define LCASE(c) (isalpha(c) ? (isupper(c) ? tolower(c) : c) : c)

char decdsply[20];
char format[] = '($$,$$$,$$$.&&)';

EXEC SQL BEGIN DECLARE SECTION;
    short stock_num;
    char description[16];
    dec_t unit_price;
EXEC SQL END DECLARE SECTION;

char errmsg[400];

main()
{
```

```
EXEC SQL BEGIN DECLARE SECTION;
    char manu_code[4];
EXEC SQL END DECLARE SECTION;

char stockin[4];
dec_t dprcnt;
float prcnt;
char ans;

EXEC SQL connect to 'stores7';         /* open the stores7 database */
err_chk('"CONNECT TO stores7");
EXEC SQL declare upcurs cursor for     /* setup cursor for update */
    select stock_num, description, unit_price from maryl.stock
    where manu_code = :manu_code and stock_num = :stock_num
    for update of unit_price;
err_chk("DECLARE upcurs");

/*
  Accept user inputs: Mfr code, stock_num, & percent
 */
printf("\n\tEnter Mfr. code: ");       /* prompt for Mfr. code  */
gets(manu_code);                       /* enter Mfr. code */
rupshift(manu_code);                   /* Mfr. code to upper case */
printf("\n\tEnter stock_num: ");       /* prompt for stock_num  */
gets(stockin);                         /* enter stock_num */
stock_num = atoi(stockin);             /* convert to int   */

printf("\n\tEnter Percent (whole number): "); /* prompt for % of increase */
scanf("%f", &prcnt);                   /* enter % of price increase */
prcnt = 1 + prcnt / 100.0;             /* convert to multiplier */
deccvdbl(prcnt, &dprcnt);              /* convert to DECIMAL type */
EXEC SQL open upcurs;                   /* open cursor */
err_chk("OPEN upcurs");

/*
  Display column headings
 */
printf("\nStock # \tDescription \tUnit Price");
EXEC SQL fetch upcurs into :stock_num, :description, :unit_price;
if(!err_chk("FETCH upcurs"))
    {
    printf("\n\n\t*** Row not found ***\n");
    exit();
    }
if(risnull(CDECIMALTYPE, &unit_price))  /* Skip if price NULL */
    {
    printf("unit_price is NULL");
    exit();
    }
rfmtdec(&unit_price, format, decdsply); /* Format unit_price */

/*
  Display item's stock_num, description and unit_price
 */
printf("\n\t%d\t%15s\t%s", stock_num, description, decdsply);

/*
  Calculate the new unit_price
 */
decmul(&unit_price, &dprcnt, &unit_price);
rfmtdec(&unit_price, format, decdsply); /* format for display */
```

```
        ans = ' ';
        /*
         Update unit_price? y(es) or n(o)
         */
        while((ans = LCASE(ans)) != 'y' && ans != 'n')
            {
            printf("\n\t. . . Update unit_price to %s ? (y/n) ", decdsply);
            scanf("%1s", &ans);
            }
        if(ans == 'y')/* if yes, update current row */
            {
            EXEC SQL update stock set unit_price = :unit_price
                where current of upcurs;
            err_chk("UPDATE stock");
            }
    }

    /*
       err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
       rgetmsg() to display the message for the error number in sqlca.sqlcode.
    */


    err_chk(name)
    char *name;
    {
        if(sqlca.sqlcode < 0)
            {
            rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
            printf("\n\tError %d during %s: %s\n",sqlca.sqlcode, name, errmsg);
            exit(1);
            }
        return((sqlca.sqlcode == SQLNOTFOUND) ? 0 : 1);
    }
```

A Sample DTP ESQL/C Application Program

The **upstock.ec** program, as shows, performs the client and server tasks. It interacts with the user, taking the data that the user provides and performs an update on a table in a database. To perform the same actions in an X/Open DTP environment, you need to divide the **upstock.ec** program into the following two programs.

■ The *client* program handles the user interface.

■ The *server* program handles the database interface.

The following sections show a sample client and server program for the **upstock** program. Most of the code for these two programs is pseudocode because the actual AP-TM calls and the way client/server connections are implemented are TM specific. Lines of code prefixed with OLTP are intended for a TM. To enable your AP to communicate with a TM, embed in your application the AP-TM calls provided by the TM software.

Important: *These sample client and server programs handle a local implicit transaction and therefore do not make use of the XA interface.*

For specific information on how to write client, service, and server programs, refer to your TM manual.

Sample Client Program

The client program, **client.c**, is a C program that performs the user interface portion of the **upstock.ec** program. For example, the Enter Mfr. Code prompt or the Update unit_price? prompt is user-supplied information that is included in the client program.

As shows, **client.c** performs the following tasks:

1. It prompts the user for the manufacturer code, stock number, and percentage of price increase.

2. It connects to the server program by calling the **OLTPCALL()** function, requesting the **QUERYSTO** service to query the stock unit price.

3. It receives the information from the server program and uses the **OLTPGET**() function to retrieve this information from the message buffer. The information returned by **QUERYSTO** appears, and the user sees the prompt Do you want to update this record?

4. If the user chooses to update, the client program calls **OPTPCALL()** again, this time to request the **UPDATESTO** service to update the stock unit price.

Important: *In this example, the OLTPCALL(), OLTPGET(), and OLTPFREE() functions are pseudofunctions. In an actual application, use the appropriate AP-TM calls of your TM interface to perform these tasks.*

Figure 3-4 shows the sample client program, **client.c**.

Figure 3-4
Sample Pseudocode for Client Program Called client.c

```
/*
 **    client.c **
 The following client program prompts the user for the manufacturer code,
 stock number, and the percent of the price increase and then passes the data
 to the server program that requests the QUERYSTO and UPDATESTO services.
*/

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

/* include decimal type structure */
struct decimal
    {
    short dec_exp;   /* exponent base 100 */
    short dec_pos;   /* sign: 1=pos, 0=neg, -1=null) */

    short dec_ndgts; /* number of significant digit */
    short dec_dgts;  /* actual digit base 100 */
    };

/* Pseudocode to include any OLTP header files that are needed. */

#include <OLTPXA.h>
.
.
.
#include 'upstock.h'

struct app_buffer
    {
    char u_manu_code[4];
    int  u_stock_code;
    char  u_stock_des[100];
    float u_percent;
    char  u_errmsg[100];
    decimal u_unit_price;
    }

.
.
.
/* End of OLTP header files */

#define LCASE(c) (isalpha(c) ? (isupper(c) ? tolower(c) : c) : c)

char decdsply[20];
char format[] =  '($$,$$$,$$$.&&)';
decimal unit_price;
char description[16];

char errmsg[400];


main()
{
```

```
char manu_code[4];
char stockin[4];
float prcnt;
char ans;
struct app_buffer *audv; /* Pointer points to message buffer */
int audv_len;

/* Pseudocode to join application  */
if (OLTPINIT() == -1)
    {
    fprintf(stderr,"Failure to join application\n");
    exit(-1);
    }
/* End of pseudocode that joins application */
/* Pseudocode that allocates message buffer  */
audv = (struct app_buffer *);
OLTPALLOCATE(TYPE, "app_buffer", sizeof(struct audv));
/* End of pseudocode that allocates message buffer */


/* Prepare and initialize structure */
void strcpy(audv->u_manu_code," ");
audv->u_stock_code = 0;
void strcpy(audv->u_stock_des,"\0");


audv->u_percent = 0;
void strcpy(audv->u_errmsg,"\0");
/* End of Prepare and initialize structure */


system("clear");                  /* Clean up the screen */
printf("\n\tEnter Mfr. code: "); /* prompt for manu_code  */
gets(manu_code);                  /* get manu_code */
rupshift(manu_code);              /* change manu_code to uppercase */

printf("\n\tEnter Stock number: ");    /* prompt for stock_num  */
gets(stockin);                         /* get stock_num */
stock_num = atoi(stockin);             /* convert stock_num to integer */

/* prompt for % of  increase */
printf("\tEnter Percent (whole number): ");
scanf("%f", &prcnt);        /* get % of price increase */
prcnt = 1 + prcnt / 100.0; /* convert % to multiplier */

/* Place data in message buffer */
void strcpy(audv->u_manu_code,manu_code);
audv->u_stock_code = stock_num;
audv->u_percent = prcnt;
/* End of placing data in message buffer */

/* Display column headings  */
printf("\nStock # \tDescription     \tUnit Price");

/* Pseudocode that connects with server program and requests
   the QUERYSTO service to query the stock unit price.
   The returned results will be put in audv message buffer */

if (OLTPCALL(QUERYSTO, (char *)audv, sizeof(struct audv),
            (char **) &audv,&audv_len,0) == -1)
```

```
        {
        /* Invoke OLTPGET CALL to get info from audv message buffer */
        /* If request fails, print the error and free up the buffer */
        fprintf(stderr, "%s service failed\n%s:    %s\n",
                        "QUERYSTO", audv->u_errmsg);

        OLTPFREE((char *) audv);
        exit(-1);
        }

    /* If QUERYSTO service successful, invoke routines to get info from
       message buffer */

    stock_num = audv->u_stock_code;
    void strcpy(description, audv->u_stock_des);
    void strcpy(&unit_price, &audv->u_unit_price);
    rfmtdec(&unit_price, format, decdsply); /* Format for display */
    void(printf("%d      %s        %s\n", stock_num, description, decdsply);

    /* Prompt user to update the item */
    printf("\nDo you want to update this record? \n");
    ans = ' ';
    /*
    Update unit_price? y(es) or n(o)
    */
    while((ans = LCASE(ans)) != 'y' && ans != 'n')
        {
        /* Calculate the new unit_price */
        decmul(&unit_price,&dpcrnt, &unit_price);
        rfmtdec(&unit_price, format, decdsply); /* Format for display */


        printf("\n\t. . . Update unit_price to %s ? (y/n) ",  decdsply);
        scanf('%1s', &ans);
        }
    /* Yes I want to update the current row */
    if(ans == 'y')/* if yes, update current row */
        {
        /* Make UPDATESTO service call to update the record */
        if (OLTPCALL(UPDATESTO, (char *)audv,
            sizeof(struct audv),(char **) &audv,&audv_len,0) == -1)
            {
            /* If service request failed, print out the error and free
               up the buffer */
            fprintf(stderr, "%s service failed\n%s:   %s\n",
                        "UPSTOCKSTO", audv->u_errmsg);
            OLTPFREE((char *) audv);
            exit(-1);
            }
        /* If update successful, return the success call, free up the
           buffer, leave the application */

        OLTPFREE((char *) audv);
        if (OLTPTERM() == -1)
            fprintf(stderr,"Can not terminate the application\n");
        else
            printf("Application completed\n");
        }
}
```

Sample Server Program

The server program, **server.ec**, is an ESQL/C program that performs the database interface portion of the **upstock.ec** program. For example, the SQL statements that access the database become the following services in the server program:

- The **QUERYSTO** service selects the data in the **stock** table for the specified manufacturer code and stock number (sent in the message buffer).
- The **UPDATESTO** service performs an update of the **unit_price** column of the **stock** table.

As shows, **server.ec** contains the services **QUERYSTO** and **UPDATESTO**. To retrieve a row from the **stock** table, the **QUERYSTO** service performs the following tasks:

1. It calls the **OLTPGETBUF()** function to obtain the specific manufacturer and stock number of the **stock** row to select. These values are sent into the function in the **transb** message buffer.

2. It performs an embedded SELECT statement that retrieves the stock number, description, and unit price for the specified row.

3. It uses the ESQL/C **risnull()** function to verify that the retrieved **unit_price** value is not null.

4. It stores the retrieved values in the message buffer and then calls the **OLTPRETURN()** function to send back to the client the data for the selected **stock** row.

To update the row, the **UPDATESTO** service performs the following tasks:

1. It calls the **OLTPGETBUF()** function to obtain the original **stock** row information from the message buffer.

2. It uses an update cursor to reselect the row for update, locking the row.

3. It compares the contents of the row with the original that was passed back to the client during **QUERYSTO**. If changes were made to that row, the update is disallowed, and an error is returned to the client.

4.   If no changes were made, **UPDATESTO** updates the unit price for the
     row by the user-specified percentage.

5.   Finally, **UPDATESTO** uses the **OLTPRETURN()** function to return the
     update status to the client program.

*Important: In this example, the **OLTPGETBUF()** and **OLTPRETURN()** functions are
pseudo functions. In an actual application, use the appropriate AP-TM calls of your
TM interface to perform these tasks.*

Figure 3-5 shows the sample server program, **server.ec**.

Figure 3-5
Sample Pseudocode for Server Program

```
/*
 ** server.ec  **
 The server program gets the request from the client program
 and performs the  QUERYSTO and UPDATESTO services.
  QUERYSTO:  Fetches rows from the stock table for a chosen
             manufacturer and stock_num.
  UPDATESTO: Updates the record with the percent of the price increase.
*/

#include <stdio.h>
#include <ctype.h>
#include <decimal.h>
EXEC SQL include sqltypes.h;

/* Pseudocode to include any OLTP header files that are needed. */

#include <OLTPXA.h>
...
...
#include 'upstock.h'

struct app_buffer
    {
    char u_manu_code[4];
    int  u_stock_code;
    char  u_stock_des[100];
    float u_percent;
    char  u_errmsg[100];
    decimal u_unit_price;
    }
.
.
.
/* End of OLTP header files */

EXEC SQL BEGIN DECLARE SECTION;
    char manu_code[4];
    short stock_num;
    char description[16];
    dec_t unit_price;
EXEC SQL END DECLARE SECTION;

char manu_code1[4];
```

```
short stock_no;
char errmsg[400];


/* The QUERYSTO service selects the data in the stock table for the
   specified manufacturer code and stock number. Row information is
   received in the message buffer.
*/
QUERYSTO(transb)
OLTPSVCINFO *transb;
{
    /*
      Setup transv pointer to point to the message buffer
    */
    struct app_buffer *transv;

    /*
      Setup the pointer to point to OLTPSVCINFO data buffer
    */
    transv = (struct app_buffer *)transb->data);

    /* Pseudo Code to get the manu_code from the message buffer */
    if (OLTPGETBUF(transb, manu_code1....) == -1)
        {

        /* call routines to check the data 'TYPE' */
        OLTPRETURN(OLTPFAIL,0,transb->data,0L,0); /* Return error message */
        }

    /* Pseudo Code to get the stock_num from the message buffer */
    if (OLTPGETBUF(transb, stock_no....) == -1)
        {
        /* call routines to check the data 'TYPE' */
        OLTPRETURN(OLTPFAIL,0,transb->data,0L,0); /* Return error message */
        }

    /*
      Retrieve one row of data, no cursor is needed
    */
    EXEC SQL select stock_num, description, unit_price
        into :stock_num, :description,:unit_price
        from stock
        where manu_code = :manu_code1 and stock_num = :stock_no;

    if(risnull(CDECIMALTYPE, &unit_price))  /* Skip if price is NULL */
        {
        fprintf(stderr, "unit_price is NULL\n");
        exit(1);
        }

    /*
      Move one row of data into the message buffer
    */
    audv->u_stock_code = stock_no;
    void strcpy(audv->u_stock_des, description);
    void strcpy(&audv->u_unit_price,&unit_price);

    /*
      Return item's stock_num, description and unit_price to client
      program
```

```
        */
        OLTPRETURN(OLTPSUCESS,.. transb->data ...);
    }


/* The UPDATESTO service performs an update of the unit_price column of the
   stock table.
*/
UPDATESTO(transb)
OLTPSVCINFO *transb;
{
    struct app_buffer *transv; /* Setup transv to point to the message buffer */
    char pre_manu_code1[4];
    short pre_stock_no;
    char pre_description[16];
    decimal pre_unit_price;

    EXEC SQL BEGIN DECLARE SECTION;
        dec_t unit_price;
    EXEC SQL BEGIN DECLARE SECTION;

    /*
      Setup the pointer to point to OLTPSVCINFO data buffer
     */
    transv = ((struct app_buffer *)transb->data);

    /*
      Get the stock row info from message buffer and store the info
      for later comparsion. This ensures that the data the client
      wants to update is current.
     */

    pre_stock_no = audv->u_stock_code;
    void strcpy(description,audv->u_stock_des);
    void strcpy(&pre_unit_price, &audv->u_unit_price);

    EXEC SQL declare upcurs cursor for           /* setup cursor for update */
       select stock_num, description, unit_price from stock
       where manu_code = :manu_code and stock_num = :stock_num
       for update of unit_price;

    if (OLTPGETBUF(transb, unit_price....) == -1)
        {
        /* call routines to check the data 'TYPE' */

        OLTPRETURN(OLTPFAIL,0,transb->data,0L,0); /* Return error message */
        }


    EXEC SQL fetch upcurs into :stock_num,:description, :unit_price;
    err_chk("FETCH udcurs");

    /* Compare the record retrieved from QUERYSTO and the current
       query to check for data consistency. If data is not consistent,
       stop updating and return error to client, else update the record;
    */

    /* Invoke routines to compare data from QUERYSTO and fetch
       data where pre* is data previously retrieved and $* is most current data.
    */
```

```
        if (COMPARE(pre*, $*) == -1)
            {
            fprintf(stderr,"Data inconsistent update request ignored \n");
            OLTPRETURN(OLTPFAIL,0,transb->data,0L,0);   /* Return error message */
            }

        EXEC SQL update stock set unit_price = :unit_price
            where current of upcurs;
        err_chk("UPDATE upcurs");

        /*
          Return update success message to client program
        */
        OLTPRETURN(OLTPSUCESS,.. transb->data ...);
}


/*
    err_chk() checks sqlca.sqlcode and if an error has occurred, it uses
    rgetmsg() to display the message for the error number in sqlca.sqlcode.
*/

err_chk(name)
char *name;
{
    if(sqlca.sqlcode < 0)
    {
      rgetmsg((short)sqlca.sqlcode, errmsg, sizeof(errmsg));
      fprintf(stder,"\n\tError %d during %s: %s\n",sqlca.sqlcode, name, errmsg);
      exit(1);
    }
    return((sqlca.sqlcode == SQLNOTFOUND) ? 0 : 1);
}
```

♦

XA Routine Return Codes

This appendix contains a listing of the XA functions and their return codes. The XA functions make up the XA interface, which allows the TM and the RM to communicate information about global transactions. These functions are defined by the X/Open DTP interface to begin with the string **xa_**. These routines are only templates. The actual function names are internal to the RM. They are defined in the XA library. For more information on the XA interface, see Chapter 1, "Informix and the X/Open Distributed Transaction-Processing Model."

Important: This appendix assumes that you are familiar with the "Informix Guide to SQL: Syntax," the "INFORMIX-ESQL/C Programmer's Manual," or the "INFORMIX-ESQL/COBOL Programmer's Manual," and "Distributed TP: The XA Specification."

The following table shows the **xa_** routines listed in this appendix.

XA Function	Purpose
xa_close()	Terminates use of an RM by an AP
xa_commit()	Tells the RM to commit a transaction branch
xa_complete()	Tests for completion of an **xa_** operation
xa_end()	Dissociates a thread from a transaction branch
xa_forget()	Tells the RM to discard its knowledge of a heuristically completed transaction branch
xa_open()	Initializes an RM for an AP to use
xa_prepare()	Asks the RM to prepare to commit a transaction branch
xa_recover()	Gets a list of XIDs that the RM has prepared or heuristically completed
xa_rollback()	Tells the RM to roll back a transaction branch
xa_start()	Starts a transaction branch

Figure A-1 on page A-3 lists the **xa_** routines and their return values. Each return code occurs because of one or more circumstances listed in the **Reason for Return Codes** column. The XA interface establishes the following naming conventions used in these return values:

- Error codes (negative values) are constants whose names begin with the string **XAER_**.
- Status codes (non-negative values) are constants whose names begin with the string **XA_**.

In addition to the error information listed in this appendix, the database server always returns a result code when you execute an SQL statement. This result code, along with other information about the operation, is returned in a global variable called **SQLSTATE** and in a data structure known as the SQL Communication Area (SQLCA). For further information about SQLCA, consult the *Informix Guide to SQL: Tutorial*.

XA Function	Return Code	Reason for Return Code
xa_close()	XAER_INVAL	Invalid arguments were specified for this routine.
		The *xa_info* argument is a null pointer.
	XAER_PROTO	Execution was between an **xa_start()** function and an **xa_end()** function for a global transaction branch when this routine was called.
		The *rmid* argument is not the same as that passed to **xa_open()**.
	XAER_RMERR	The database server failed.
		An internal communication error occurred between the application development tool and the database server.
		The database server encountered failure on close of database.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous operations when the TMASYNC flag was passed in the *flags* argument.
xa_commit()	XAER_INVAL	Invalid arguments were specified for this routine.
	XAER_PROTO	This routine called before a call to **xa_open()**.
		The transaction branch was active or suspended.
		The *rmid* argument is not the same as that passed to **xa_open()**.
		The TMONEPHASE flag was passed in the *flags* argument, and the transaction branch was already prepared for commit.

(1 of 10)

XA Function	Return Code	Reason for Return Code
		The TMONEPHASE flag was not passed in the *flags* argument, and the transaction branch has not yet been prepared for commit.
	XAER_RMFAIL	The database server failed.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous operations when the TMASYNC flag was passed in the *flags* argument.
	XAER_RMERR	An internal communication error occurred between the application development tool and the database server.
		The database server could not find deferred constraint information, probably because of memory corruption.
		The database server encountered an error while freeing the shared-memory transaction entry.
		The database server encountered a failure during write of commit record (during two-phase commit).
	XA_RBTRANSIENT	Execution was between an **xa_start()** function and an **xa_end()** function on another transaction branch when this call occurred. The database server was unable to save the state of the current transaction branch to perform the commit.
	XA_RBINTEGRITY	The database server encountered a constraint error while checking deferred constraints.
	XAER_NOTA	The database server could not find the transaction branch indicated by the *xid* argument.
	XA_HEURRB	Due to a heuristic decision, the database server has already rolled back the transaction branch.

(2 of 10)

XA Function	Return Code	Reason for Return Code
	XA_RBOTHER	The TMONEPHASE flag was passed in the *flags* argument, and the transaction branch was marked for rollback only, probably due to a long transaction.
	XA_RBROLLBACK	The TMONEPHASE flag was passed in the *flags* argument, and the commit failed. The database server has rolled back the trans-action branch.
xa_complete()	XAER_PROTO	The database server does not support asynchronous operations.
xa_end()	XAER_INVAL	Invalid arguments were specified for this routine.
	XAER_PROTO	This routine was called before a call to **xa_open()**.
		The TMMIGRATE flag is passed in the *flags* argument, but transaction migration is not supported.
		The *rmid* argument is not the same as that passed to **xa_open()**.
		This routine has been called in an improper context.
	XAER_RMFAIL	The database server failed.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous opera-tions when the TMASYNC flag was passed in the *flags* argument.
	XAER_RMERR	The database server encountered failure on suspension of transaction branch, probably because of a memory-allocation error.
		An internal communication error occurred between the application development tool and the database server.

(3 of 10)

XA Function	Return Code	Reason for Return Code
		The database server encountered a failure while attempting to save deferred constraint information—a resource-allocation error.
	XA_RBOTHER	The TMSUCCESS flag was passed in the *flags* argument, and this is a suspended-transaction branch. The implicit resumption of the transaction branch failed.
		The database server has marked the transaction branch for rollback only, probably due to a long transaction.
	XA_RBROLLBACK	The TMFAIL flag was passed in the *flags* argument.
	XAER_NOTA	The database server could not find the transaction branch indicated by the *xid* argument.
xa_forget()	XAER_INVAL	Invalid arguments were specified for this routine.
	XAER_PROTO	This routine was called before a call to **xa_open()**.
		The *rmid* argument is not the same as that passed to **xa_open()**.
		This transaction branch was not heuristically completed.
	XAER_RMFAIL	The database server failed.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous operations when the TMASYNC flag was passed in the *flags* argument.
	XAER_RMERR	An internal communication error occurred between the application development tool and the database server.
		The database server encountered a failure on write of end-transaction record.

(4 of 10)

XA Function	Return Code	Reason for Return Code
	XAER_NOTA	The database server could not find the trans-action branch indicated by the *xid* argument.
xa_open()	XAER_INVAL	Invalid arguments were specified for this routine.
		The *xa_info* argument is a null pointer.
		The *xa_info* string is too long (greater than MAXINFOSIZE).
		No database name was given or database name is too long.
		A non-XA connection to the database server already exists.
		Database does not exist.
		Database does not have logging.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous opera-tions when the TMASYNC flag was passed in the *flags* argument.
	XAER_RMERR	This routine was unable to connect to the database server.
		An internal communication error occurred between the application development tool and the database server.
		The database server was unable to open the database (for a reason other than it does not exist or it has no logging).
xa_prepare()	XAER_INVAL	Invalid arguments were specified for this routine.
	XAER_PROTO	This routine was called before a call to **xa_open()**.
		A transaction branch was active or suspended.

XA Function	Return Code	Reason for Return Code
		The *rmid* argument is not the same as that passed to **xa_open()**.
		The transaction branch was already prepared for commit.
	XAER_RMFAIL	The database server failed.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous operations with the TMASYNC flag passed in the *flags* argument.
	XAER_RMERR	An internal communication error occurred between the application development tool and the database server.
		The database server could not find deferred constraint information, probably because of memory corruption.
		The database server encountered an error while freeing the shared-memory transaction entry. (The entry is only freed if the prepare fails and the transaction branch is rolled back or the branch was read only.)
	XA_RBTRANSIENT	Execution was between an **xa_start()** function and an **xa_end()** function on another transaction branch when this call occurred. The database server was unable to save the state of the current transaction branch to perform the prepare.
	XA_RBINTEGRITY	The database server encountered a constraint error while checking deferred constraints.
	XAER_NOTA	The database server could not find the transaction branch indicated by the *xid* argument.
	XA_RBOTHER	The database server has marked this transaction branch as rollback only, probably because of a long transaction.

(6 of 10)

XA Function	Return Code	Reason for Return Code
	XA_RDONLY	This transaction branch was read only and has been committed.
	XA_RBROLLBACK	The database server rolled back the transaction branch because of a failure during write of the prepare log record.
xa_recover()	XAER_INVAL	Invalid arguments were specified for this routine.
		The *xids* argument is null, and the *count* argument is greater than 0.
		The *count* argument is less than 0.
	XAER_PROTO	This routine was called before a call to **xa_open()**.
		The *rmid* argument is not the same as that passed to **xa_open()**.
		This routine has been called in an improper context.
	XAER_RMFAIL	The database server failed.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous operations when the TMASYNC flag was passed in the *flags* argument.
	XAER_RMERR	The database server encountered a memory-allocation error.
		The number of XIDs returned is greater than or equal to 0.
xa_rollback()	XAER_INVAL	Invalid arguments were specified for this routine.
	XAER_PROTO	This routine was called before a call to **xa_open()**.
		The *rmid* argument is not the same as that passed to **xa_open()**.

(7 of 10)

XA Function	Return Code	Reason for Return Code
	XAER_RMFAIL	The database server failed.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous operations when the TMASYNC flag was passed in the *flags* argument.
	XAER_RMERR	An internal communication error occurred between the application development tool and the database server.
		The database server could not find deferred constraint information, probably because of memory corruption.
		The database server encountered an error while freeing the shared-memory transaction entry.
		The database server failed to roll back the transaction branch.
	XA_RBTRANSIENT	Execution was between an **xa_start()** function and an **xa_end()** function on another transaction branch when this call occurred. The database server was unable to save the state of the current transaction branch to perform the rollback.
	XAER_NOTA	The database server could not find the transaction branch indicated by the *xid* argument.
	XA_HEURRB	Because of a heuristic decision, the database server has already rolled back the transaction branch.
xa_start()	XAER_INVAL	Invalid arguments were specified for this routine.
	XAER_PROTO	This routine was called before a call to **xa_open()**.

XA Function	Return Code	Reason for Return Code
		TMRESUME was passed in the *flags* argument, and the transaction branch was not suspended by this user.
		TMRESUME was not passed in the *flags* argument, and the transaction branch was suspended by this user.
		The *rmid* argument is not the same as that passed to **xa_open()**.
		This routine has been called in an improper context.
	XAER_RMFAIL	The database server failed.
	XAER_ASYNC	The database server encountered an error while implementing asynchronous operations when the TMASYNC flag was passed in the *flags* argument.
	XAER_RMERR	Resume of the transaction branch failed. The database server could not find the suspended context, probably because of memory corruption.
		An internal communication error occurred between the application development tool and the database server.
		The database server could not find deferred constraint information, probably because of memory corruption.
		The database server could not allocate a new transaction branch because it ran out of entries in the transaction table.
		The *begin work* operation for this transaction branch failed.
	XAER_OUTSIDE	A local transaction is open and must be committed or rolled back before calling **xa_start()**.

(9 of 10)

XA Function	Return Code	Reason for Return Code
	XAER_DUPID	The database server was expecting to allocate a new transaction branch, but the XID already exists. Neither TMJOIN nor TMRESUME flag has been passed in as part of the *flags* argument.
	XAER_NOTA	The database server could not find the transaction branch indicated by the *xid* argument.
	XA_RBOTHER	The database server has marked this transaction branch as rollback only, probably due to a long transaction.

(10 of 10)

Index