X/Open CAE Specification

Distributed Transaction Processing: The XA Specification

X/Open Company Ltd.

X/Open CAE Specification

Distributed Transaction Processing: The XA Specification

ISBN: 1 872630 24 3
X/Open Document Number: XO/CAE/91/300

Set in Palatino by X/Open Company Ltd., U.K.
Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

List of Tables

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and allows users to move between systems with a minimum of retraining.

The components of the Common Applications Environment are defined in X/Open CAE Specifications. These contain, among other things, an evolving portfolio of practical application programming interfaces (APIs), which significantly enhance portability of application programs at the source code level, and definitions of, and references to, protocols and protocol profiles, which significantly enhance the interoperability of applications.

The X/Open CAE Specifications are supported by an extensive set of conformance tests and a distinct X/Open trademark - the XPG brand - that is licensed by X/Open and may be carried only on products that comply with the X/Open CAE Specifications.

The XPG brand, when associated with a vendor's product, communicates clearly and unambiguously to a procurer that the software bearing the brand correctly implements the corresponding X/Open CAE Specifications. Users specifying XPG-conformance in their procurements are therefore certain that the branded products they buy conform to the CAE Specifications.

X/Open is primarily concerned with the selection and adoption of standards. The policy is to use formal approved *de jure* standards, where they exist, and to adopt widely supported *de facto* standards in other cases.

Where formal standards do not exist, it is X/Open policy to work closely with standards development organisations to assist in the creation of formal standards covering the needed functions, and to make its own work freely available to such organisations. Additionally, X/Open has a commitment to align its definitions with formal approved standards.

X/Open Specifications

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the long-life specifications that form the basis for conformant and branded X/Open systems. They are intended to be used widely within the industry for product development and procurement purposes.

  Developers who base their products on a current CAE Specification can be sure that either the current specification or an upwards-compatible version of it will be referenced by a future XPG brand (if not referenced already), and that a variety of compatible, XPG-branded systems capable of hosting their products will be available, either immediately or in the near future.

  CAE Specifications are not published to coincide with the launch of a particular XPG brand, but are published as soon as they are developed. By providing access to its specifications in this way, X/Open makes it possible for products that conform to the CAE (and hence are eligible for a future XPG brand) to be developed as soon as practicable, enhancing the value of the XPG brand as a procurement aid to users.

- *Preliminary Specifications*

  These are specifications, usually addressing an emerging area of technology, and consequently not yet supported by a base of conformant product implementations, that are released in a controlled manner for the purpose of validation through practical implementation or prototyping. A Preliminary Specification is not a ''draft'' specification. Indeed, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE Specification.

  Preliminary Specifications are analogous with the ''trial-use'' standards issued by formal standards organisations, and product development teams are intended to develop products on the basis of them. However, because of the nature of the technology that a Preliminary Specification is addressing, it is untried in practice and may therefore change before being published as a CAE Specification. In such a case the CAE Specification will be made as upwards-compatible as possible with the corresponding Preliminary Specification, but complete upwards-compatibility in all cases is not guaranteed.

In addition, X/Open periodically publishes:

- *Snapshots*

  Snapshots are ''draft'' documents, which provide a mechanism for X/Open to disseminate information on its current direction and thinking to an interested audience, in advance of formal publication, with a view to soliciting feedback and comment.

A Snapshot represents the interim results of an X/Open technical activity. Although at the time of publication X/Open intends to progress the activity towards publication of an X/Open Preliminary or CAE Specification, X/Open is a consensus organisation, and makes no commitment regarding publication.

Similarly, a Snapshot does not represent any commitment by any X/Open member to make any specific products available.

X/Open Guides

X/Open Guides provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant.

X/Open Guides are not normative, and should not be referenced for purposes of specifying or claiming X/Open-conformance.

This Document

The January 1987 edition of the **X/Open Portability Guide** committed X/Open to standardise facilities by which commercial applications could achieve distributed transaction processing (DTP) on UNIX systems. This document specifies the bidirectional interface between a transaction manager and resource manager (the XA interface).

This document is a CAE specification (see above), which was initially issued as a Preliminary Specification in April 1990, and reissued as a Snapshot of current thinking in June 1991. This CAE reflects changes to the specification resulting from prototype implementations and committee and industry review.

This specification is structured as follows:

- Chapter 1 is an introduction.
- Chapter 2 provides fundamental definitions for the remainder of the document.
- Chapter 3 is an overview of the XA interface.
- Chapter 4 discusses the data structures that are part of the XA interface.
- Chapter 5 contains reference manual pages for each routine in the XA interface.
- Chapter 6 contains state tables.
- Chapter 7 summarises the implementation requirements and identifies optional features.
- Appendix A is the code of the header file required by XA routines.

There is an index at the end.

Typographical Conventions

The following typographical conventions are used throughout this document:

- `Constant width` strings are code examples or literals and are to be typed just as they appear.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics also denote:
  - — variable names
  - — commands or utilities
  - — functions; these are shown as follows: *name*().
- The notation "**file.h**" indicates a header.
- The notation [ABCD] is the name of a return value.
- Ellipses (...) are used to show that additional arguments are optional.

Trademarks

X/Open™ and the 'X' device are trademarks of X/Open Company Limited in the U.K. and other countries.

UNIX® is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

Palatino™ is a trademark of Linotype AG and/or its subsidiaries.

Referenced Documents

The following documents are referenced in this specification:

ASN.1
: Information Processing Systems — Open Systems Interconnection — Specification of Abstract Syntax Notation 1 (ASN.1), ISO 8824, 1990.

BER
: Information Processing Systems — Open Systems Interconnection — Specification of Basic Encding Rules for Abstract Syntax Notation 1 (ASN.1), ISO 8825, 1990.

C
: ISO/IEC 9899:1990 (which is technically identical to ANS X3.159-1989, Programming Language C)

DTP
: X/Open Guide, Distributed Transaction Processing Reference Model, X/Open Company Ltd., October 1991.

OSI DTP
: The ISO/IEC Open Systems Interconnection (OSI) Distributed Transaction Processing (DTP) standard.

  - ISO/IEC DIS 10026-1 (1991) (model)
  - ISO/IEC DIS 10026-2 (1991) (service)
  - ISO/IEC DIS 10026-3 (1991) (protocol)

OSI CCR
: The ISO/IEC Open Systems Interonnection (OSI) Commitment, Concurrency, and Recovery (CC) standard.

  - ISO/IEC 9804.3 (1989) (service)
  - ISO/IEC 9805.3 (1989) (protocol)

SQL
: X/Open Developers' Specification, Structured Query Language (SQL), X/Open Company Ltd., 1990, or any later revision.

Chapter 1

Introduction

The X/Open Distributed Transaction Processing (DTP) model envisages three software components:

- An application program (AP) defines transaction boundaries and specifies actions that constitute a transaction.
- Resource managers (RMs, such as databases or file access systems) provide access to shared resources.
- A separate component called a transaction manager (TM) assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for failure recovery.

Chapter 2 defines each component in more detail and illustrates the flow of control.

This document specifies the *XA interface*: the bidirectional interface between a transaction manager and a resource manager. The XA interface is not an ordinary Application Programming Interface (API). It is a system-level interface between DTP software components. X/Open is developing other DTP interfaces for direct use by an application program (see Section 2.1 on page 3 for an overview). These interfaces may be the subject of future publications.

This specification is limited to the model presented in Section 2.1 on page 3. This specification does not discuss aspects of the model that pertain to communication. X/Open anticipates that heterogeneous TMs will use the OSI DTP protocols for communication of DTP information and application data. Such communication will involve interfaces in addition to the one described in this specification, and will involve a more detailed DTP model. This is deferred to a later publication.

Relevant definitions and other important concepts are discussed in Chapter 2. This chapter also defines the AP, TM and RM in more detail, and describes their interaction. Chapter 3 is an overview of the XA interface, describing the situations in which each of the services is used. Chapter 4 discusses the data structures that are part of the XA interface. Reference manual pages for each routine in the XA interface are presented in Chapter 5; state tables follow in Chapter 6. Chapter 7 summarises the implications of this specification on the implementors of RMs and TMs; it also identifies features that are optional. Appendix A presents the contents of an "**xa.h**" header file in both ANSI C and Common Usage C.

Chapter 2

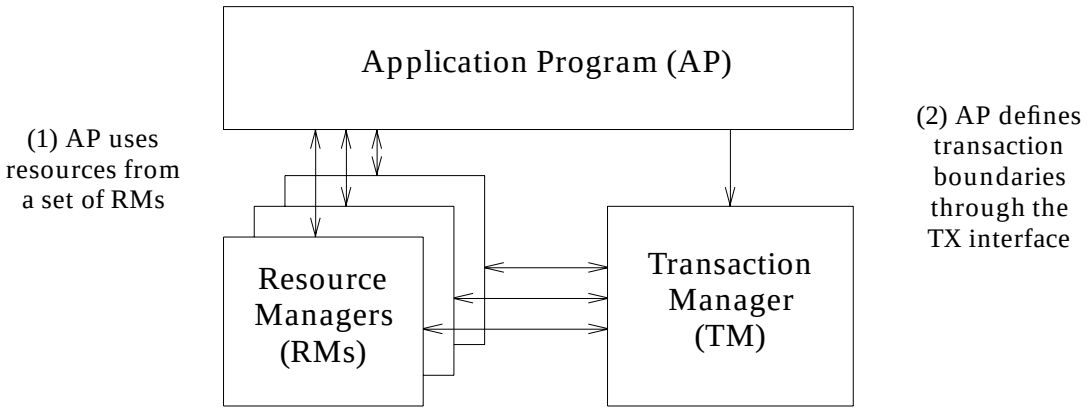
Model and Definitions

This chapter discusses the XA interface in general terms and provides necessary background material for the rest of the specification. The chapter shows the relationship of the interface to the X/Open DTP model. The chapter also states the design assumptions that the interface uses and shows how the interface addresses common DTP concepts.

2.1 X/Open DTP Model

The figure below illustrates a local instance of a DTP system where an AP calls a TM to structure transactions. The boxes indicate software components in the X/Open DTP model (see the definitions in Section 2.2 on page 4). The arrows indicate the directions in which control flows.

There may be several DTP systems coexisting on the same processor. The boxes in the figure below are not necessarily separate processes, nor necessarily a single thread of control (see Section 2.2.8 on page 6). Furthermore, the components of this model do not have invariable roles. For example, an RM might use the TX interface to do work in support of a transaction.



2.1.1 Interfaces between Local TP Components

The subject of this X/Open specification is interface (3) in the diagram above, the XA interface by which TMs and RMs interact.

For more details on this model and diagram, including detailed definitions of each component, see the referenced **DTP** guide.

2.2 Definitions

For additional definitions see the referenced **DTP** guide.

2.2.1 Transaction

A transaction is a complete unit of work. It may comprise many computational tasks, which may include user interface, data retrieval, and communications. A typical transaction modifies shared resources. (The referenced OSI DTP specification (model) defines transactions more precisely.)

Transactions must be able to be *rolled back*. A human user may roll back the transaction in response to a real-world event, such as a customer decision. A program can elect to roll back a transaction. For example, account number verification may fail or the account may fail a test of its balance. Transactions also roll back if a component of the system fails, keeping it from retrieving, communicating, or storing data. Every DTP software component subject to transaction control must be able to undo its work in a transaction that is rolled back at any time.

When the system determines that a transaction can complete without failure of any kind, it *commits* the transaction. This means that changes to shared resources take permanent effect. Either commitment or rollback results in a consistent state. *Completion* means either commitment or rollback.

2.2.2 Distributed Transaction Processing

Within the scope of this document, DTP systems are those where work in support of a single transaction may occur across RMs. This has several implications:

- The system must have a way to refer to a transaction that encompasses all work done anywhere in the system.
- The decision to commit or roll back a transaction must consider the status of work done anywhere on behalf of the transaction. The decision must have uniform effect throughout the DTP system.

Even though an RM may have an X/Open-compliant interface, such as Structured Query Language (SQL), it must also address these two items to be useful in the DTP environment.

2.2.3 Application Program

The AP defines transactions and accesses resources within transaction boundaries. Each AP specifies a sequence of operations that involves resources such as terminals and databases. This specification generally uses the term AP to refer to a single instance of an application program.

2.2.4 Resource Manager

An RM manages a certain part of the computer's shared resources. Many other software entities can request access to the resource from time to time, using services that the RM provides. Here are some examples of RMs:

- A database management system (DBMS) is an RM. Typical DBMSs are capable of defining transactions and committing work atomically.
- A file access method such as the Indexed Sequential Access Method (ISAM) can be the basis for an RM. Typically, an ISAM RM must be enhanced to support transactions as defined herein.
- A print server might be implemented as an RM.

A single RM may service multiple independent resource domains. An RM *instance* services one of these domains. (See also Section 3.2 on page 13.) Unless specified otherwise, operations this specification allows on an RM are allowed on each RM instance.

2.2.5 Global Transactions

Every RM in the DTP environment must support transactions as described in Section 2.2.1 on page 4. Many RMs already structure their work into recoverable units.

In the DTP environment, many RMs may operate in support of the same unit of work. This unit of work is a *global transaction*. For example, an AP might request updates to several different databases. Work occurring anywhere in the system must be committed atomically. Each RM must let the TM coordinate the RM's recoverable units of work that are part of a global transaction.

Commitment of an RM's internal work depends not only on whether its own operations can succeed, but also on operations occurring at other RMs, perhaps remotely. If any operation fails anywhere, every participating RM must roll back all operations it did on behalf of the global transaction. A given RM is typically unaware of the work that other RMs are doing. A TM informs each RM of the existence, and directs the completion, of global transactions. An RM is responsible for mapping its recoverable units of work to the global transaction.

2.2.6 Transaction Branches

A global transaction has one or more *transaction branches* (or *branches*). A branch is a part of the work in support of a global transaction for which the TM and the RM engage in a separate but coordinated transaction commitment protocol (see Section 2.3 on page 8). Each of the RM's internal units of work in support of a global transaction is part of exactly one branch.

A global transaction might have more than one branch when, for example, the AP uses multiple processes or is involved in the same global transaction by multiple remote APs.

After the TM begins the transaction commitment protocol, the RM receives no additional work to do on that transaction branch. The RM may receive additional work

on behalf of the same transaction, from different branches. The different branches are related in that they must be completed atomically.

Each transaction branch identifier (or XID — see Section 4.2 on page 19) that the TM gives the RM identifies both a global transaction and a specific branch. The RM may use this information to optimise its use of shared resources and locks.

2.2.7 Transaction Manager

TMs manage global transactions, coordinate the decision to commit them or roll them back, and coordinate failure recovery. The AP defines the start and end of a global transaction by calling a TM. The TM assigns an identifier to the global transaction (see Section 4.2 on page 19). The TM manages global transactions and informs each RM of the XID on behalf of which the RM is doing work. Although RMs can manage their own recoverable work units as they see fit, each RM must accept XIDs and associate them with those work units. In this way, an RM knows what recoverable work units to complete when the TM completes a global transaction.

2.2.8 Thread of Control

A thread of control (or a *thread*) is the entity, with all its context, that is currently in control of a processor. A thread of control is an operating-system process: an address space and single thread of control that executes within that address space, and its required system resources. The context may include the process' locks on shared resources, and the files the process has open. For portability reasons, the notion of thread of control must be common among the AP, TM and RM.

The thread concept is central to the TM's coordination of RMs. APs call RMs to request work, while TMs call RMs to delineate transaction branches. The way the RM knows that a given work request pertains to a given branch is that the AP and the TM both call it from *the same thread of control*. For example, an AP thread calls the TM to declare the start of a global transaction. The TM records this fact and informs RMs. After the AP regains control, it uses the native interface of one or more RMs to do work. The RM receives the calls from the AP and TM in the same thread of control.

Certain XA routines, therefore, must be called from a particular thread. The reference manual pages in Chapter 5 indicate which routines require this.

2.2.9 Tightly- and Loosely-coupled Threads

Many application threads of control can participate in a single global transaction. All the work done in these threads is atomically completed. Within a single global transaction, the relationship between any pair of participating threads is either *tightly-coupled* or *loosely-coupled*:

- A tightly-coupled relationship is one where a pair of threads are designed to share resources. In addition, with respect to an RM's isolation policies, the pair are treated as a single entity. Thus, for a pair of tightly-coupled threads, the RM must guarantee that resource deadlock does not occur within the transaction branch.

- A loosely-coupled relationship provides no such guarantee. With respect to an RM's isolation policies, the pair may be treated as if they were in separate global transactions even though the work is atomically completed.

Within a single global transaction, a set of tightly-coupled threads may consist of more than just a pair. Moreover, many sets of tightly-coupled threads may exist within the same global transaction and each set is loosely coupled with respect to the others. The reference manual pages in Chapter 5 indicate how a TM communicates these relationships to an RM.

2.3 Transaction Completion and Recovery

TMs and RMs use two-phase commit with presumed rollback, as defined by the referenced OSI DTP specification (model).

In Phase 1, the TM asks all RMs to *prepare to commit* (or *prepare*) transaction branches. This asks whether the RM can guarantee its ability to commit the transaction branch. An RM may have to query other entities internal to that RM.

If an RM can commit its work, it records stably the information it needs to do so, then replies affirmatively. A negative reply reports failure for any reason. After making a negative reply and rolling back its work, the RM can discard any knowledge it has of the transaction branch.

In Phase 2, the TM issues all RMs an actual request to commit or roll back the transaction branch, as the case may be. (Before issuing requests to commit, the TM stably records the fact that it decided to commit, as well as a list of all involved RMs.) All RMs commit or roll back changes to shared resources and then return status to the TM. The TM can then discard its knowledge of the global transaction.

2.3.1 Rolling Back the Global Transaction

The TM rolls back the global transaction if any RM responds negatively to the Phase 1 request, or if the AP directs the TM to roll back the global transaction. Therefore, any negative response vetoes the global transaction. A negative response concludes an RM's involvement in the global transaction.

The TM effects Phase 2 by telling all RMs to roll back transaction branches. They must not let any changes to shared resources become permanent. The TM does not issue Phase 2 requests to RMs that responded negatively in Phase 1. The TM does not need to record stably the decision to roll back nor the participants in a rolled back global transaction.

2.3.2 Protocol Optimisations

- **Read-only**
  An RM can respond to the TM's prepare request by asserting that the RM was not asked to update shared resources in this transaction branch. This response concludes the RM's involvement in the transaction; the Phase 2 dialogue between the TM and this RM does not occur. The TM need not stably record, in its list of participating RMs, an RM that asserts a read-only role in the global transaction.

  However, if the RM returns the read-only optimisation before all work on the global transaction is prepared, global *serialisability*[1] cannot be guaranteed. This is because the RM may release transaction context, such as read locks, before all application activity for that global transaction is finished.

1. Serialisability is a property of a set of concurrent transactions. For a serialisable set of transactions, at least one serial sequence of the transactions exists that produces identical results, with respect to shared resources, as does concurrent execution of the transaction.

- **One-phase Commit**
  A TM can use one-phase commit if it knows that there is only one RM anywhere in the DTP system that is making changes to shared resources. In this optimisation, the TM makes its Phase 2 commit request without having made a Phase 1 prepare request. Since the RM decides the outcome of the transaction branch and forgets about the transaction branch before returning to the TM, there is no need for the TM to record stably these global transactions and, in some failure cases, the TM may not know the outcome.

2.3.3 Heuristic Branch Completion

Some RMs may employ heuristic decision-making: an RM that has prepared to commit a transaction branch may decide to commit or roll back its work independently of the TM. It could then unlock shared resources. This may leave them in an inconsistent state. When the TM ultimately directs an RM to complete the branch, the RM may respond that it has already done so. The RM reports whether it committed the branch, rolled it back, or completed it with mixed results (committed some work and rolled back other work).

An RM that reports heuristic completion to the TM must not discard its knowledge of the transaction branch. The TM calls the RM once more to authorise it to forget the branch. This requirement means that the RM must notify the TM of all heuristic decisions, even those that match the decision the TM requested. The referenced OSI DTP specifications (model) and (service) define heuristics more precisely.

2.3.4 Failures and Recovery

A useful DTP system must be able to recover from a variety of failures. A storage device or medium, a communication path, a node, or a program could fail.

Failures that a node can correct internally may not affect a global transaction.

Failures that do not disrupt the commitment protocol let the DTP system respond by rolling back appropriate global transactions. For example, an RM recovering from a failure responds negatively to a prepare request based on the fact that it does not recognise the XID.

More significant failures may disrupt the commitment protocol. The TM typically senses the failure when an expected reply does not arrive.

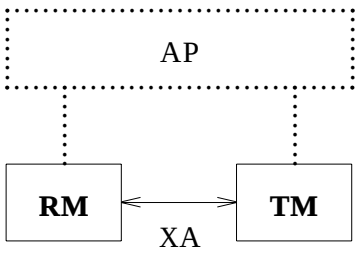
Failure and recovery processing in an X/Open DTP system is compatible with the referenced OSI DTP specifications, which define the presumed-rollback protocol. The X/Open DTP model makes these assumptions:

- TMs and RMs have access to stable storage
- TMs coordinate and control recovery
- RMs provide for their own restart and recovery of their own state. On request, an RM must give a TM a list of XIDs that the RM has prepared for commitment or has heuristically completed.

Chapter 3

Interface Overview

This chapter gives an overview of the XA interface. This is the interface between the TM and the RM in an X/Open DTP system. Chapter 5 contains reference manual pages for each routine in alphabetical order. These pages contain C-language function prototypes.



The X/Open DTP model envisages interfaces between each of the AP, RM, and TM (see Section 2.1 on page 3). Generally, each use of the XA interface is prompted by the AP calling the TM or the RM.

3.1 Index to Services in the XA Interface

Name	Description	See
ax_reg	Register an RM with a TM.	Section 3.3.1 on page 16
ax_unreg	Unregister an RM with a TM.	Section 3.3.1 on page 16
xa_close	Terminate the AP's use of an RM.	Section 3.2 on page 13
xa_commit	Tell the RM to commit a transaction branch.	Section 3.4 on page 17
xa_complete	Test an asynchronous *xa_* operation for completion.	Section 3.5 on page 18
xa_end	Dissociate the thread from a transaction branch.	Section 3.3 on page 14
xa_forget	Permit the RM to discard its knowledge of a heuristically-completed transaction branch.	Section 3.4 on page 17
xa_open	Initialise an RM for use by an AP.	Section 3.2 on page 13
xa_prepare	Ask the RM to prepare to commit a transaction branch.	Section 3.4 on page 17
xa_recover	Get a list of XIDs the RM has prepared or heuristically completed.	Section 3.6 on page 18
xa_rollback	Tell the RM to roll back a transaction branch.	Section 3.4 on page 17
xa_start	Start or resume a transaction branch - associate an XID with future work that the thread requests of the RM.	Section 3.3 on page 14

The *ax_* routines let an RM call a TM. All TMs must provide these routines. These routines let an RM dynamically control its participation in a transaction branch.

The *xa_* routines are supplied by RMs operating in the DTP environment and called by TMs. When an AP calls a TM to start a global transaction, the TM may use the *xa_* interface to inform RMs of the transaction branch. After the AP uses the RM's native interface to do work in support of the global transaction, the TM calls *xa_* routines to commit or roll back branches. One other *xa_* routine helps the TM coordinate failure recovery.

A TM must call the *xa_* routines in a particular sequence (see the state tables in Chapter 6). When a TM invokes more than one RM with the same *xa_* routine, it can do so in an arbitrary sequence.

Note: The routine names in the *xa_* series are only templates.

The actual names of these functions are internal to the RM. The RM publishes the name of a structure (see Section 4.3 on page 21) that specifies the entry points to the RM.

3.2 Opening and Closing Resource Managers

In each thread of control, the TM must call *xa_open*() for each RM directly accessible by that thread before calling any other *xa_* routine. The TM must eventually call *xa_close*() to dissociate the AP from the RM.

If an RM needs to take start-up actions (such as opening files, opening paths to a server, or resynchronising a node on the network), then it could do so when called by *xa_open*(). X/Open does not specify the actual meaning of *xa_open*() and *xa_close*() to an RM, but the effect must be internal to the RM and must not affect transaction processing in either the calling TM or in other RMs.

If an RM requires or accepts parameters to govern its operation (for example, a directive to open files for reading only), or to identify a target resource domain, then a string argument to *xa_open*() conveys this information. If the RM does not require initialisation parameters, the string is typically an empty string. The *xa_close*() call likewise takes a string.

TMs typically read the initialisation string from a configuration file. The *xa_open*() routine, and the string form of its argument, support portability. A TM can give the administrator control over every run-time option that any RM provides through *xa_open*() with no reprogramming or relinking. The administrator must only edit a configuration file or perform a comparable, system-specific procedure.

The TM calls *xa_open*() with an identifier that the TM uses subsequently to identify the RM instance. A single RM may service multiple resource domains using multiple RM instances, if each instance supports independent transaction completion. For example, a single database system might access several data domains, or a single printer spooler might service multiple printers. The TM calls such an RM's *xa_open*() routine several times, once for each instance, using string parameters that identify the respective resource. It must generate a different RM identifier for each call.

To enhance portability, RMs in the DTP environment should rely on the use of *xa_open*() in place of any non-standard *open* service the RM may provide in its native interface. If an RM lets DTP applications call the native *open* routine, the effect must not conflict with the TM's use of *xa_open*().

3.3 Association of Threads with Transaction Branches

Several threads may participate in a single transaction branch, some more than once. The *xa_start*() and *xa_end*() routines pass an XID to an RM to associate or dissociate the calling thread with a branch. The association is not necessarily the thread's initial association with the branch; its dissociation is not necessarily the final one.

A thread's association with a transaction branch can be active or suspended:

- A thread is actively associated with a transaction branch if it has called *xa_start*() and has not made a corresponding call to *xa_end*(). A thread is allowed only one active association with each RM at a time.
- Certain calls to *xa_end*() suspend the thread's association (see **Suspend** below). The call may indicate that the association can *migrate*, that is, that any thread may resume the association. In this case, the calling thread is no longer associated. (An RM may indicate that it does not support association migration.)

If a thread calls *xa_end*() to suspend its association but the association cannot migrate to another thread, the calling thread retains a suspended association with the transaction branch.

Several uses of *xa_start*() and *xa_end*() are considered below:

- **Start**
  The primary use of *xa_start*() is to register a new transaction branch with the RM. This marks the start of the branch. Subsequently, the AP, *using the same thread of control*, uses the RM's native interface to do useful work. All requests for service made by the same thread are part of the same branch until the thread dissociates from the branch (see below).

  The return code from *xa_start*() may indicate that the RM has already vetoed commitment of the transaction branch. This return code is not an error; rolled back global transactions may be routine, while actual errors deserve the administrator's attention.

- **Join**
  Another use of *xa_start*() is to join an existing transaction branch. TMs must use a certain form of *xa_start*() so that RMs can validate that they recognise the passed XID.

  RMs in the DTP environment should anticipate that many threads will try to use them concurrently. If multiple threads use an RM on behalf of the same XID, the RM is free to serialise the threads' work in any way it sees fit. For example, an RM may block a second or subsequent thread while one is active.

- **Resume**
  A special form of *xa_start*() associates a thread with an existing transaction branch that has been suspended (see below).

- **End**
  A typical call to *xa_end*() dissociates the calling thread from the transaction branch and lets the branch be completed (see Section 3.4 on page 17). Alternatively, a thread may use *xa_start*() to rejoin the branch.

- **Suspend**
  A form of *xa_end*() suspends, instead of ending, a thread's association with the transaction branch. This indicates that the thread has left the branch in an incomplete state. By using the **resume** form of *xa_start*(), it or another thread resumes its association with the branch. Instead of resuming, the TM may completely end the suspended association by using *xa_end*().
- **Rollback-only**
  An RM need not wait for global transaction completion to report an error. The RM can return **rollback-only** as the result of any *xa_start*() or *xa_end*() call. The TM can use this knowledge to avoid starting additional work on behalf of the global transaction. An RM can also unilaterally roll back and forget a transaction branch any time before it prepares it. A TM detects this when an RM subsequently indicates that it does not recognise the XID.
- **Transaction branch states**
  Several state tables appear in Chapter 6. Each call to *xa_start*() or *xa_end*() may affect the status of the thread's association with a transaction branch (see Table 6-2 on page 59 and Table 6-3 on page 60) and the status of the branch itself (see Table 6-4 on page 62). A TM must use these routines so that each thread of control makes calls in a sequence that complies with both tables.

Transaction Context

Transaction context is RM-specific information visible to the AP. The RM should preserve certain transaction context on *xa_end*() so that the RM can restore context in the join or resume cases (defined above). In the join case, the RM should make available enough transaction context so that tightly-coupled threads are not susceptible to resource deadlock within the transaction branch. In the resume case, the RM should make available at least that RM-specific transaction context present at the time of the suspend, as if the thread had effectively never been suspended, except that other threads in the global transaction may have affected this context.

3.3.1 Registration of Resource Managers

Normally, a TM involves all associated RMs in a transaction branch. (The TM's set of RM switches, described in Section 4.3 on page 21 tells the TM which RMs are associated with it.) The TM calls all these RMs with *xa_start*(), *xa_end*(), and *xa_prepare*(), although an RM that is not active in a branch need not participate further (see Section 2.3.2 on page 8). A technique to reduce overhead for infrequently-used RMs is discussed below.

Dynamic Registration

Certain RMs, especially those involved in relatively few global transactions, may ask the TM to assume they are *not* involved in a transaction. These RMs must register with the TM before they do application work, to see whether the work is part of a global transaction. The TM never calls these RMs with any form of *xa_start*(). An RM declares dynamic registration in its switch (see Section 4.3 on page 21). An RM can make this declaration only on its own behalf, and doing so does not change the TM's behaviour with respect to other RMs.

When an AP requests work from such an RM, before doing any work, the RM contacts the TM by calling *ax_reg*(). The RM must call *ax_reg*() from the same thread of control that the AP would use if it called *ax_reg*() directly. The TM returns to the RM the appropriate XID if the AP is in a global transaction.

If the thread ends its involvement in the transaction branch (using *xa_end*()), then the RM must re-register (using *ax_reg*()) with the TM if the AP calls it for additional work in the global transaction. If the RM does not resume its participation, then the TM does not call the RM again for that branch until the TM completes the branch.

If the RM calls *ax_reg*() and the AP is not in a global transaction, the TM informs the RM, and remembers, that the RM is doing work outside any global transaction. In this case, when the AP completes its work with the RM, the RM must notify the TM by calling *ax_unreg*(). The RM must call *ax_unreg*() from the same thread of control from which it called *ax_reg*(). Until then — that is, as long as the AP thread involves the RM outside a global transaction — the TM neither lets the AP start a global transaction, nor lets any RM register through the same thread to participate in one.

3.4 Branch Completion

A TM calls *xa_prepare*() to ask the RM to prepare to commit a transaction branch. The RM places any resources it holds in a state such that it can either make any changes permanent if the TM subsequently calls *xa_commit*(), or nullify any changes if the TM calls *xa_rollback*(). An affirmative return from *xa_prepare*() guarantees that a subsequent *xa_commit*() or *xa_rollback*() succeeds, even if the RM experiences a failure after responding to *xa_prepare*().

A TM calls *xa_commit*() to direct the RM to commit a transaction branch. The RM applies permanently any changes it has made to shared resources, and releases any resources it held on behalf of the branch. A TM calls *xa_rollback*() to ask the RM to roll back a branch. The RM undoes any changes that it applied to shared resources, and releases any resources it held.

Before a TM can call *xa_prepare*() for a transaction branch, all associations must be completely ended with *xa_end*() (see Section 3.3 on page 14). Any thread can then initiate branch completion. That is, the TM may supervise branch completion with a separate thread from the AP threads that did work on behalf of the global transaction.

Optimisations

This section describes the use of *xa_* routines in the standard two-phase commit protocol. See Section 2.3.2 on page 8 for other permissible sequences of these calls.

Heuristic Decision

The X/Open DTP model lets RMs complete transaction branches heuristically (see Section 2.3.3 on page 9). The RM cannot discard its knowledge of such a branch until the TM permits this by calling *xa_forget*() for each branch.

3.5 Synchronous, Non-blocking and Asynchronous Modes

Synchronous

The *xa_* functions typically operate synchronously: control does not return to the caller until the operation is complete. Some routines, notably *xa_start*() (see Section 3.3 on page 14) may block the calling thread.

Two other calling modes help the TM schedule its work when dealing with several RMs:

Non-blocking

Certain *xa_* calls direct the RM to operate synchronously with the caller but without blocking it. If the RM cannot complete the call without blocking, it reports this immediately.

Asynchronous

Most *xa_* routines have a form by which the caller requests asynchrony. Asynchronous calls should return immediately. The caller can subsequently call *xa_complete*() to test the asynchronous operation for completion.

A TM might give an RM an asynchronous request (particularly a request to prepare to commit a transaction branch) so that the TM could do other work in the meantime. Within the same thread of control, a TM cannot use asynchrony to give additional work to the same RM for the same branch; the only *xa_* call a TM can give to the RM for the same branch is *xa_complete*() to test that operation's completion. However, for the branch-completion routines: *xa_commit*(), *xa_prepare*() and *xa_rollback*(), and for *xa_forget*(), the TM may issue multiple commands to the same RM from within the same thread of control. Each of these commands must be for a different branch.

3.6 Failure Recovery

A TM must ensure orderly completion of all transaction branches. A TM calls *xa_recover*() during failure recovery to get a list of all branches that an RM has prepared or heuristically completed.

Unilateral RM Action

An RM can mark a transaction branch as rollback-only any time except after a successful prepare. A TM detects this when the RM returns a rollback-only return code. An RM can also unilaterally roll back and forget a branch any time except after a successful prepare. A TM detects this when a subsequent call indicates that the RM does not recognise the XID. The former technique gives the TM more information.

If a thread of control terminates, an RM must dissociate and roll back any associated transaction branch. If an RM experiences machine failure or termination, it must also unilaterally roll back all branches that have not successfully prepared.

Chapter 4

The "xa.h" Header

This chapter specifies structure definitions, flags, and error codes to which conforming products must adhere. It also declares the routines by which RMs call a TM. (Entry points to an RM are contained in the RM's switch; see Section 4.3 on page 21.) This is the minimum content of an include file called "**xa.h**". Fully standardising this information lets RMs be written independently of the TMs that use them. It also lets users interchange TMs and RMs without recompiling.

Appendix A contains an "**xa.h**" header file with **#define** statements suitable for ANSI C (see the referenced C standard) and Common Usage C implementations. This chapter contains excerpts from the ANSI C code in "**xa.h**". The synopses in Chapter 5 also use ANSI C.

4.1 Naming Conventions

The XA interface uses certain naming conventions to name its functions, flags and return codes. All names that appear in "**xa.h**" are part of the XA name space. This section describes the XA naming conventions.

- The negative (error) codes returned by the *xa_* routines all begin with XAER_. Their non-negative return codes all begin with XA_.
- The names of all TM functions that RMs call begin with *ax_* (for example, *ax_reg*). Their negative (error) return codes all begin with TMER_. Their non-negative return codes all begin with TM_.
- Names of flags passed to XA routines, and of flags in the RM switch, begin with TM.

4.2 Transaction Identification

The "**xa.h**" header defines a public structure called an XID to identify a transaction branch. RMs and TMs both use the XID structure. This lets an RM work with several TMs without recompilation.

The XID structure is specified in the C code below in **struct xid_t**. The XID contains a format identifier, two length fields and a data field. The data field comprises at most two contiguous components: a global transaction identifier (*gtrid*) and a branch qualifier (*bqual*).

The *gtrid_length* element specifies the number of bytes (1-64) that constitute *gtrid*, starting at the first byte of the data element (that is, at *data*[0]). The *bqual_length* element specifies the number of bytes (1-64) that constitute *bqual*, starting at the first byte after *gtrid* (that is, at *data*[*gtrid_length*]). Neither component in *data* is null-terminated. The TM need not initialise any unused bytes in *data*.

Although "**xa.h**" constrains the length and byte alignment of the data element within an XID, it does not specify the data's contents. The only requirement is that both *gtrid* and *bqual*, taken together, must be globally unique. The recommended way of achieving global uniqueness is to use the naming rules specified for OSI CCR atomic action identifiers (see the referenced OSI CCR specification). If OSI CCR naming is used, then the XID's *formatID* element should be set to 0; if some other format is used, then the *formatID* element should be greater than 0. A value of –1 in *formatID* means that the XID is null.

The RM must be able to map the XID to the recoverable work it did for the corresponding branch. RMs may perform bitwise comparisons on the data components of an XID for the lengths specified in the XID structure. Most XA routines pass a pointer to the XID. These pointers are valid only for the duration of the call. If the RM needs to refer to the XID after it returns from the call, it must make a local copy before returning.

```
/*
 * Transaction branch identification: XID and NULLXID:
 */
#define XIDDATASIZE  128         /* size in bytes */
#define MAXGTRIDSIZE 64          /* maximum size in bytes of gtrid */
#define MAXBQUALSIZE 64          /* maximum size in bytes of bqual */
struct xid_t {
   long formatID;     /* format identifier */
   long gtrid_length; /* value 1-64 */
   long bqual_length; /* value 1-64 */
   char data[XIDDATASIZE];
   };
typedef struct xid_t XID;
/*
 * A value of -1 in formatID means that the XID is null.
 */
/*
 * Declarations of routines by which RMs call TMs:
 */
extern int ax_reg(int, XID *, long);
extern int ax_unreg(int, long);
```

4.3 Resource Manager Switch

The TM administrator can add or remove an RM from the DTP system by simply controlling the set of RMs linked to executable modules. Each RM must provide a *switch* that gives the TM access to the RM's *xa_* routines. This lets the administrator change the set of RMs linked with an executable module without having to recompile the application. A different set of RMs and their switches may be linked into each separate application executable module in the DTP system. Several instances of an RM can share the RM's switch structure.

An RM's switch uses a structure called **xa_switch_t**. The switch contains the RM's name, non-null pointers to the RM's entry points, a flag and a version word. The flags tell whether the RM uses dynamic registration (see Section 3.3.1 on page 15), whether the RM operates asynchronously (see Section 3.5 on page 18) and whether the RM supports the migration of associations (see Section 3.3 on page 14). Section 4.4 on page 22 defines constants used as these flags. The RM cannot change these declarations during the operation of the DTP system.

```
/*
 * XA Switch Data Structure
 */
#define RMNAMESZ 32           /* length of resource manager name, */
                              /* including the null terminator */
#define MAXINFOSIZE 256       /* maximum size in bytes of xa_info strings,*/
                              /* including the null terminator */
struct xa_switch_t {
   char name[RMNAMESZ];       /* name of resource manager */
   long flags;                /* options specific to the resource manager */
   long version;              /* must be 0 */
   int (*xa_open_entry)(char *, int, long);  /* xa_open function pointer */
   int (*xa_close_entry)(char *, int, long); /* xa_close function pointer */
   int (*xa_start_entry)(XID *, int, long);  /* xa_start function pointer */
   int (*xa_end_entry)(XID *, int, long);    /* xa_end function pointer */
   int (*xa_rollback_entry)(XID *, int, long);
                                             /* xa_rollback function pointer*/
   int (*xa_prepare_entry)(XID *, int, long);
                                             /* xa_prepare function pointer */
   int (*xa_commit_entry)(XID *, int, long); /* xa_commit function pointer */
   int (*xa_recover_entry)(XID *, long, int, long);
                                             /* xa_recover function pointer */
   int (*xa_forget_entry)(XID *, int, long); /* xa_forget function pointer */
   int (*xa_complete_entry)(int *, int *, int, long);
                                             /* xa_complete function pointer*/
};
```

4.4 Flag Definitions

The XA interface uses the flag definitions. For a TM to work with different RMs without change or recompilation, each RM uses these flags, defined in the "**xa.h**" header.

The "**xa.h**" header defines a constant, TMNOFLAGS, for use in situations where no other flags are specified. An RM that does not use any flags to specify special features in its switch (see Section 4.3 on page 21) should specify TMNOFLAGS. In addition, TMs and RMs should use the same TMNOFLAGS constant as the flag argument in any *xa_* or *ax_* call in which they do not use explicit options.

Flag definitions for the XA interface are as follows:

```
/*
 * Flag definitions for the RM switch
 */
#define TMNOFLAGS      0x00000000L   /* no resource manager features
                                        selected */
#define TMREGISTER     0x00000001L   /* resource manager dynamically
                                        registers */
#define TMNOMIGRATE    0x00000002L   /* resource manager does not support
                                        association migration */
#define TMUSEASYNC     0x00000004L   /* resource manager supports
                                        asynchronous operations */
/*
 * Flag definitions for xa_ and ax_ routines
 */
/* use TMNOFLAGS, defined above, when not specifying other flags */
#define TMASYNC        0x80000000L   /* perform routine asynchronously */
#define TMONEPHASE     0x40000000L   /* caller is using one-phase commit
                                        optimisation */
#define TMFAIL         0x20000000L   /* dissociates caller and marks
                                        transaction branch rollback-only */
#define TMNOWAIT       0x10000000L   /* return if blocking condition exists */
#define TMRESUME       0x08000000L   /* caller is resuming association
                                        with suspended transaction branch */
#define TMSUCCESS      0x04000000L   /* dissociate caller from transaction
                                        branch */
#define TMSUSPEND      0x02000000L   /* caller is suspending, not ending,
                                        association */
#define TMSTARTRSCAN   0x01000000L   /* start a recovery scan */
#define TMENDRSCAN     0x00800000L   /* end a recovery scan */
#define TMMULTIPLE     0x00400000L   /* wait for any asynchronous operation */
#define TMJOIN         0x00200000L   /* caller is joining existing transaction
                                        branch */
#define TMMIGRATE      0x00100000L   /* caller intends to perform migration */
```

4.5 Return Codes

As with flag definitions, all TMs and RMs must ensure interchangeability by using these return codes, defined in the "**xa.h**" header.

```
/*
 * ax_() return codes (transaction manager reports to resource manager)
 */
#define TM_JOIN       2     /* caller is joining existing transaction
                               branch */
#define TM_RESUME     1     /* caller is resuming association
                               with suspended transaction branch */
#define TM_OK         0     /* normal execution */
#define TMER_TMERR    -1    /* an error occurred in the
                               transaction manager */
#define TMER_INVAL    -2    /* invalid arguments were given */
#define TMER_PROTO    -3    /* routine invoked in an improper context */

/*
 * xa_() return codes (resource manager reports to transaction manager)
 */
#define XA_RBBASE      100            /* the inclusive lower bound of the
                                         rollback codes */
#define XA_RBROLLBACK  XA_RBBASE      /* the rollback was caused by an
                                         unspecified reason */
#define XA_RBCOMMFAIL  XA_RBBASE+1    /* the rollback was caused by a
                                         communication failure */
#define XA_RBDEADLOCK  XA_RBBASE+2    /* a deadlock was detected */
#define XA_RBINTEGRITY XA_RBBASE+3    /* a condition that violates the
                                         integrity of the resources
                                         was detected */
#define XA_RBOTHER     XA_RBBASE+4    /* the resource manager rolled back the
                                         transaction branch for a reason not on
                                         this list */
#define XA_RBPROTO     XA_RBBASE+5    /* a protocol error occurred in the
                                         resource manager */
#define XA_RBTIMEOUT   XA_RBBASE+6    /* a transaction branch took too long*/
#define XA_RBTRANSIENT XA_RBBASE+7    /* may retry the transaction branch */
#define XA_RBEND       XA_RBTRANSIENT /* the inclusive upper bound of the
                                         rollback codes */

#define XA_NOMIGRATE   9   /* resumption must occur where
                              suspension occurred */
#define XA_HEURHAZ     8   /* the transaction branch may have
                              been heuristically completed */
#define XA_HEURCOM     7   /* the transaction branch has been
                              heuristically committed */
#define XA_HEURRB      6   /* the transaction branch has been
                              heuristically rolled back */
#define XA_HEURMIX     5   /* the transaction branch has been
                              heuristically committed and rolled back */
#define XA_RETRY       4   /* routine returned with no effect and
                              may be reissued */
```

```
#define XA_RDONLY      3   /* the transaction branch was read-only and
                           has been committed */
#define XA_OK          0   /* normal execution */

#define XAER_ASYNC     -2   /* asynchronous operation already outstanding */
#define XAER_RMERR     -3   /* a resource manager error occurred in the
                            transaction branch */
#define XAER_NOTA      -4   /* the XID is not valid */
#define XAER_INVAL     -5   /* invalid arguments were given */
#define XAER_PROTO     -6   /* routine invoked in an improper context */
#define XAER_RMFAIL    -7   /* resource manager unavailable */
#define XAER_DUPID     -8   /* the XID already exists */
#define XAER_OUTSIDE   -9   /* resource manager doing work outside */
                            /* global transaction */
```

Chapter 5

Reference Manual Pages

This chapter describes the interfaces to the XA service set. Reference manual pages appear in alphabetical order, for each service in the XA interface. The *ax_* routines are provided by a TM for use by RMs. The *xa_* routines are provided by each RM for use by the TM.

The symbolic constants and error names are described in the "**xa.h**" header (see Chapter 4). The state tables referred to in the reference manual pages appear in Chapter 6.

NAME

ax_reg — dynamically register a resource manager with a transaction manager

SYNOPSIS

```
#include "xa.h"

int
ax_reg(int rmid, XID *xid, long flags)
```

DESCRIPTION

A resource manager calls *ax_reg*() to inform a transaction manager that it is about to perform work on behalf of an application thread of control. The transaction manager, in turn, replies to the resource manager with an indication of whether or not that work should be performed on behalf of a transaction branch. If the transaction manager determines that the calling thread of control is involved in a branch, upon successful return, *xid* points to a valid XID. If the resource manager's work is outside any global transaction, *xid* points to NULLXID. The caller is responsible for allocating the space to which *xid* points.

A resource manager must call this function from the same thread of control from which the application accesses the resource manager. A resource manager taking advantage of this facility must have TMREGISTER set in the *flags* element of its **xa_switch_t** structure (see Chapter 4). Moreover, *ax_reg*() returns failure, [TMER_TMERR], when issued by a resource manager that has not set TMREGISTER.

When the resource manager calls *ax_reg*() for a new thread of control association (that is, when [TM_RESUME] is not returned; see below), the transaction manager may generate a unique branch qualifier within the returned XID.

If the transaction manager elects to reuse within **xid* a branch qualifier previously given to the resource manager, it informs the resource manager of this by returning [TM_JOIN]. If the resource manager receives [TM_JOIN] and does not recognise **xid*, it must return a failure indication to the application.

If the resource manager is resuming work on a suspended transaction branch, it informs the resource manager of this by returning [TM_RESUME]. When [TM_RESUME] is returned, *xid* points to the same XID that was passed to the *xa_end*() call that suspended the association. If the resource manager receives [TM_RESUME] and does not recognise **xid*, it must return a failure indication to the application.

If the transaction manager generated a new branch qualifier within the returned XID, this thread is loosely-coupled in relation to the other threads in this same global transaction. That is, the resource manager may treat this thread's work as a separate transaction with respect to its isolation policies. If the transaction manager reuses a branch qualifier within the returned XID, this thread is tightly-coupled to the other threads in the same transaction branch. A resource manager must guarantee that tightly-coupled threads are treated as a single entity with respect to its isolation policies and that no resource deadlock can occur within the transaction branch among these tightly-coupled threads.

The implications of dynamically registering are as follows: when a thread of control begins working on behalf of a transaction branch, the transaction manager calls *xa_start*() for all resource managers known to the thread except those having

TMREGISTER set in their **xa_switch_t** structure. Thus, those resource managers with this flag set must explicitly join a branch with *ax_reg*(). Secondly, when a thread of control is working on behalf of a branch, a transaction manager calls *xa_end*() for all resource managers known to the thread that either do not have TMREGISTER set in their **xa_switch_t** structure or have dynamically registered with *ax_reg*().

The function's first argument, *rmid*, is the integer that the resource manager received when the transaction manager called *xa_open*(). It identifies the resource manager in the thread of control.

The function's last argument, *flags*, is reserved for future use and must be set to TMNOFLAGS.

RETURN VALUE

The function *ax_reg*() has the following return values:

[TM_JOIN]
: The resource manager is joining the work of an existing transaction branch. The resource manager should make available enough transaction context so that tightly-coupled threads are not susceptible to resource deadlock within the branch. If the resource manager does not recognise ∗*xid*, it must return a failure indication to the application.

[TM_RESUME]
: The resource manager should resume work on a previously-suspended transaction branch. The resource manager should make available at least the transaction context that is specific to the resource manager, present at the time of the suspend, as if the thread had effectively never been suspended, except that other threads in the global transaction may have affected this context.

  If the resource manager does not recognise ∗*xid*, it must return a failure indication to the application. If the resource manager allows an association to be resumed in a different thread from the one that suspended the work, and the transaction manager expressed its intention to migrate the association (via the TMMIGRATE flag on *xa_end*()), the current thread may be different from the one that suspended the work. Otherwise, the current thread must be the same, or the resource manager must return a failure indication to the application.

  If ∗*xid* contains a reused branch qualifier, and the transaction manager has multiple outstanding suspended thread associations for ∗*xid*, the following rules apply:

  - The transaction manager can have only one of them outstanding at any time with TMMIGRATE set in *flags*.
  - Moreover, the transaction manager cannot resume this association in a thread that currently has a non-migratable suspended association.

  These rules prevent ambiguity as to which context is restored.

[TM_OK]
: Normal execution.

[TMER_TMERR]
The transaction manager encountered an error in registering the resource manager.

[TMER_INVAL]
Invalid arguments were specified.

[TMER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

ax_unreg(), *xa_end*(), *xa_open*(), *xa_start*().

WARNINGS

If *xid* does not point to a buffer that is at least as large as the size of an XID, *ax_reg*() may overwrite the caller's data space. In addition, the buffer must be properly aligned (on a long word boundary) in the event that structure assignments are performed.

NAME

ax_unreg — dynamically unregister a resource manager with a transaction manager

SYNOPSIS

```
#include "xa.h"

int
ax_unreg(int rmid, long flags)
```

DESCRIPTION

A resource manager calls *ax_unreg*() to inform a transaction manager that it has completed work, outside any global transaction, that it began after receiving the NULLXID from *ax_reg*(). In addition, the resource manager is informing the transaction manager that the accessing thread of control is free to participate (from the resource manager's perspective) in a global transaction. So long as any resource manager in a thread of control is registered with a transaction manager and is performing work outside any global transaction, that application thread cannot participate in a global transaction.

A resource manager must call this function from the same thread of control that originally called *ax_reg*(). A resource manager taking advantage of this facility must have TMREGISTER set in the *flags* element of its **xa_switch_t** structure (see Chapter 4). Moreover, *ax_unreg*() returns failure [TMER_TMERR] when issued by a resource manager that has not set TMREGISTER.

The function's first argument, *rmid*, is the integer that the resource manager received when the transaction manager called *xa_open*(). It identifies the resource manager in the thread of control.

The function's last argument, *flags*, is reserved for future use and must be set to TMNOFLAGS.

RETURN VALUE

ax_unreg() has the following return values:

[TM_OK]
: Normal execution.

[TMER_TMERR]
: The transaction manager encountered an error in unregistering the resource manager.

[TMER_INVAL]
: Invalid arguments were specified.

[TMER_PROTO]
: The routine was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

ax_reg().

NAME

xa_close — close a resource manager

SYNOPSIS

```
#include "xa.h"

int
xa_close(char *xa_info, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_close*() to close a currently open resource manager in the calling thread of control. Once closed, the resource manager cannot participate in global transactions on behalf of the calling thread until it is re-opened.

The argument *xa_info* points to a null-terminated character string that may contain instance-specific information for the resource manager. The maximum length of this string is 256 bytes (including the null terminator). The argument *xa_info* may point to an empty string if the resource manager does not require instance-specific information to be available. The argument *xa_info* must not be a null pointer.

A transaction manager must call this function from the same thread of control that accesses the resource manager. In addition, attempts to close a resource manager that is already closed have no effect and return success, [XA_OK].

It is an error, [XAER_PROTO], for a transaction manager to call *xa_close*() within a thread of control that is associated with a transaction branch (that is, the transaction manager must call *xa_end*() before calling *xa_close*()). In addition, if a transaction manager calls *xa_close*() while an asynchronous operation is pending at a resource manager, an error, [XAER_PROTO], is returned.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*(), identifies the resource manager called from the thread of control.

The function's last argument, *flags* must be set to one of the following values:

TMASYNC
: This flag indicates that *xa_close*() shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*() to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager, this function fails, returning [XAER_ASYNC].

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

RETURN VALUE

The function *xa_close*() has the following return values:

[XA_OK]
: Normal execution.

[XAER_ASYNC]
: TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred when closing the resource.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_complete(), *xa_end*(), *xa_open*().

WARNINGS

From the resource manager's perspective, the pointer *xa_info* is valid only for the duration of the call to *xa_close*(). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xa_info* points. Resource managers are encouraged to use private copies of ∗*xa_info* after the function completes.

NAME

xa_commit — commit work done on behalf of a transaction branch

SYNOPSIS

```
#include "xa.h"

int
xa_commit(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_commit*() to commit the work associated with **xid*. Any changes made to resources held during the transaction branch are made permanent. A transaction manager may call this function from any thread of control. All associations for **xid* must have been ended by using *xa_end*() with TMSUCCESS set in *flags*.

If a resource manager already completed the work associated with **xid* heuristically, this function merely reports how the resource manager completed the transaction branch. A resource manager cannot forget about a heuristically completed branch until the transaction manager calls *xa_forget*().

A transaction manager must issue a separate *xa_commit*() for each transaction branch that accessed the resource manager.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*(), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags* (note that at most one of TMNOWAIT and TMASYNC may be set):

TMNOWAIT
: When this flag is set and a blocking condition exists, *xa_commit*() returns [XA_RETRY] and does not commit the transaction branch (that is, the call has no effect). The function *xa_commit*() must be called at a later time to commit the branch. TMNOWAIT is ignored if TMONEPHASE is set.

TMASYNC
: This flag indicates that *xa_commit*() shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*() to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, this function fails, returning [XAER_ASYNC].

TMONEPHASE
: The transaction manager must set this flag if it is using the one-phase commit optimisation for the specified transaction branch.

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

RETURN VALUE

The function *xa_commit*() has the following return values:

[XA_HEURHAZ]
Due to some failure, the work done on behalf of the specified transaction branch may have been heuristically completed.

[XA_HEURCOM]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed.

[XA_HEURRB]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was rolled back.

[XA_HEURMIX]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back.

[XA_RETRY]
The resource manager is not able to commit the transaction branch at this time. This value may be returned when a blocking condition exists and TMNOWAIT was set. Note, however, that this value may also be returned even when TMNOWAIT is not set (for example, if the necessary stable storage is currently unavailable). This value cannot be returned if TMONEPHASE is set in *flags*. All resources held on behalf of ∗*xid* remain in a prepared state until commitment is possible. The transaction manager should reissue *xa_commit*() at a later time.

[XA_OK]
Normal execution.

[XA_RB∗]
The resource manager did not commit the work done on behalf of the transaction branch. Upon return, the resource manager has rolled back the branch's work and has released all held resources. These values may be returned only if TMONEPHASE is set in *flags*:

[XA_RBROLLBACK]
The resource manager rolled back the transaction branch for an unspecified reason.

[XA_RBCOMMFAIL]
A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager rolled back the transaction branch for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred in committing the work performed on behalf of the transaction branch and the branch's work has been rolled back. Note that returning this error signals a catastrophic event to a transaction manager since other resource managers may successfully commit their work on behalf of this branch. This error should be returned only when a resource manager concludes that it can never commit the branch and that it cannot hold the branch's resources in a prepared state. Otherwise, [XA_RETRY] should be returned.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable.

[XAER_NOTA]
The specified XID is not known by the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_complete(), *xa_forget*(), *xa_open*(), *xa_prepare*(), *xa_rollback*().

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_commit*(). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of ∗*xid* after the function completes.

NAME

xa_complete — wait for an asynchronous operation to complete

SYNOPSIS

```
#include "xa.h"

int
xa_complete(int *handle, int *retval, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_complete*() to wait for the completion of an asynchronous operation. By default, this function waits for the operation pointed to by *handle* to complete. The argument **handle* must have previously been returned by a function that had TMASYNC set. In addition, a transaction manager must call *xa_complete*() from the same thread of control that received **handle*.

Upon successful return, [XA_OK], *retval* points to the return value of the asynchronous operation and **handle* is no longer valid. If *xa_complete*() returns any other value, **handle*, **retval*, and any outstanding asynchronous operation are not affected. The caller is responsible for allocating the space to which *handle* and *retval* point.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*(), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags*:

TMMULTIPLE
: When this flag is set, *xa_complete*() tests for the completion of any outstanding asynchronous operation. Upon success, the resource manager places the handle of the completed asynchronous operation in the area pointed to by **handle*.

TMNOWAIT
: When this flag is set, *xa_complete*() tests for the completion of an operation without blocking. That is, if the operation denoted by **handle* (or any operation, if TMMULTIPLE is also set) has not completed, *xa_complete*() returns [XA_RETRY] and does not wait for the operation to complete.

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

RETURN VALUE

The function *xa_complete*() has the following return values:

[XA_RETRY]
: TMNOWAIT was set in *flags* and no asynchronous operation has completed.

[XA_OK]
: Normal execution.

[XAER_PROTO]
: The routine was invoked in an improper context. See Chapter 6 for details.

[XAER_INVAL]
: Invalid arguments were specified.

SEE ALSO

xa_close(), *xa_commit*(), *xa_end*(), *xa_forget*(), *xa_open*(), *xa_prepare*(), *xa_rollback*(), *xa_start*().

NAME

xa_end — end work performed on behalf of a transaction branch

SYNOPSIS

```
#include "xa.h"

int
xa_end(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_end*() when a thread of control finishes, or needs to suspend work on, a transaction branch. This occurs when the application completes a portion of its work, either partially or in its entirety (for example, before blocking on some event in order to let other threads of control work on the branch). When *xa_end*() successfully returns, the calling thread of control is no longer actively associated with the branch but the branch still exists. A transaction manager must call this function from the same thread of control that accesses the resource manager.

A transaction manager always calls *xa_end*() for those resource managers that do not have TMREGISTER set in the *flags* element of their **xa_switch_t** structure. Unlike *xa_start*(), *xa_end*() is also issued to those resource managers that have previously registered with *ax_reg*(). After the transaction manager calls *xa_end*(), it should no longer consider the calling thread associated with that resource manager (although it must consider the resource manager part of the transaction branch when it prepares the branch.) Thus, a resource manager that dynamically registers must re-register after an *xa_end*() that suspends its association (that is, after an *xa_end*() with TMSUSPEND set in *flags*) but before the application thread of control continues to access the resource manager.

The first argument, *xid*, is a pointer to an XID. The argument *xid* must point to the same XID that was either passed to the *xa_start*() call or returned from the *ax_reg*() call that established the thread's association; otherwise, an error, [XAER_NOTA], is returned.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*(), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags* (note that one and only one of TMSUSPEND, TMSUCCESS or TMFAIL must be set):

TMSUSPEND

Suspend a transaction branch on behalf of the calling thread of control. For a resource manager that allows multiple threads of control, but only one at a time working on a specific branch, it might choose to allow another thread of control to work on the branch at this point. If this flag is not accompanied by the TMMIGRATE flag, the transaction manager must resume or end the suspended association in the current thread. TMSUSPEND cannot be used in conjunction with either TMSUCCESS or TMFAIL.

TMMIGRATE

The transaction manager intends (but is not required) to resume the association in a thread different from the calling one. This flag may be used only in conjunction with TMSUSPEND and only if a resource manager does not have

TMNOMIGRATE set in the flags element of its **xa_switch_t** structure. Setting TMMIGRATE in flags, while another thread's association for ∗*xid* is currently suspended with TMMIGRATE, makes *xa_end*() fail, returning [XAER_PROTO]. This is because a transaction manager can have at any given time at most one suspended thread association migrating for a particular transaction branch. If this flag is not used, a transaction manager is required to resume or end the association in the current thread.

TMSUCCESS
The portion of work has succeeded. This flag cannot be used in conjunction with either TMSUSPEND or TMFAIL.

TMFAIL
The portion of work has failed. A resource manager might choose to mark a transaction branch as rollback-only at this point. In fact, a transaction manager does so for the global transaction. If a resource manager chooses to do so also, *xa_end*() returns one of the [XA_RB∗] values. TMFAIL cannot be used in conjunction with either TMSUSPEND or TMSUCCESS.

TMASYNC
This flag indicates that *xa_end*() shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*() to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, this function fails, returning [XAER_ASYNC].

RETURN VALUE

The function *xa_end*() has the following return values:

[XA_NOMIGRATE]
The resource manager was unable to prepare the transaction context for migration. However, the resource manager has suspended the association. The transaction manager can resume the association only in the current thread. This code may be returned only when both TMSUSPEND and TMMIGRATE are set in *flags*. A resource manager that sets TMNOMIGRATE in the *flags* element of its **xa_switch_t** structure need not return [XA_NOMIGRATE].

[XA_OK]
Normal execution.

[XA_RB∗]
The resource manager has dissociated the transaction branch from the thread of control and has marked rollback-only the work performed on behalf of ∗*xid*. The following values may be returned regardless of the setting of *flags*:

[XA_RBROLLBACK]
The resource manager marked the transaction branch rollback-only for an unspecified reason.

[XA_RBCOMMFAIL]
A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager marked the transaction branch rollback-only for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred in dissociating the transaction branch from the thread of control.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable.

[XAER_NOTA]
The specified XID is not known by the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

ax_reg(), *xa_complete*(), *xa_open*(), *xa_start*().

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_end*(). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of **xid* after the function completes.

NAME

xa_forget — forget about a heuristically completed transaction branch

SYNOPSIS

```
#include "xa.h"

int
xa_forget(XID *xid, int rmid, long flags)
```

DESCRIPTION

A resource manager that heuristically completes work done on behalf of a transaction branch must keep track of that branch along with the decision (that is, heuristically committed, rolled back, or mixed) until told otherwise. The transaction manager calls *xa_forget*() to permit the resource manager to erase its knowledge of **xid*. Upon successful return, [XA_OK], **xid* is no longer valid (from the resource manager's point of view). A transaction manager may call this function from any thread of control.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*(), identifies the resource manager called from the thread of control.

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC
: This flag indicates that *xa_forget*() shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*() to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, this function fails, returning [XAER_ASYNC].

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

RETURN VALUE

The function *xa_forget*() has the following return values:

[XA_OK]
: Normal execution.

[XAER_ASYNC]
: TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
: An error occurred in the resource manager and the resource manager has not forgotten the transaction branch.

[XAER_RMFAIL]
: An error occurred that makes the resource manager unavailable.

[XAER_NOTA]
: The specified XID is not known by the resource manager as a heuristically completed XID.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_commit(), *xa_complete*(), *xa_open*(), *xa_recover*(), *xa_rollback*().

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_forget*(). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of ∗*xid* after the function completes.

NAME

xa_open — open a resource manager

SYNOPSIS

```
#include "xa.h"

int
xa_open(char *xa_info, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_open*() to initialise a resource manager and prepare it for use in a distributed transaction processing environment. It applies to resource managers that support the notion of *open* and must be called before any other resource manager (*xa_*) calls are made.

The argument *xa_info* points to a null-terminated character string that may contain instance-specific information for the resource manager. The maximum length of this string is 256 bytes (including the null terminator). The argument *xa_info* may point to an empty string if the resource manager does not require instance-specific information to be available. The argument *xa_info* must not be a null pointer.

The argument *rmid*, an integer assigned by the transaction manager, uniquely identifies the called resource manager instance within the thread of control. The transaction manager passes the *rmid* on subsequent calls to XA routines to identify the resource manager. This identifier remains constant until the transaction manager in this thread closes the resource manager.

If the resource manager supports multiple instances, the transaction manager can call *xa_open*() more than once for the same resource manager. The transaction manager generates a new *rmid* value for each call, and must use different values for **xa_info* on each call, typically to identify the respective resource domain.

A transaction manager must call this function from the same thread of control that accesses the resource manager. In addition, attempts to open a resource manager instance that is already open have no effect and return success, [XA_OK].

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC
: This flag indicates that *xa_open*() shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*() to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager, this function fails, returning [XAER_ASYNC].

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

RETURN VALUE

The function *xa_open*() has the following return values:

[XA_OK]
: Normal execution.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred when opening the resource.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_close(), *xa_complete*().

WARNINGS

From the resource manager's perspective, the pointer *xa_info* is valid only for the duration of the call to *xa_open*(). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xa_info* points. Resource managers are encouraged to use private copies of ∗*xa_info* after the function completes.

NAME

xa_prepare — prepare to commit work done on behalf of a transaction branch

SYNOPSIS

```
#include "xa.h"

int
xa_prepare(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_prepare*() to request a resource manager to prepare for commitment any work performed on behalf of **xid*. The resource manager places any resources it held or modified in such a state that it can make the results permanent when it receives a commit request (that is, when the transaction manager calls *xa_commit*()). If the transaction branch has already been prepared with *xa_prepare*(), subsequent calls to *xa_prepare*() return [XAER_PROTO]. A transaction manager may call this function from any thread of control. All associations for **xid* must have been ended by using *xa_end*() with TMSUCCESS set in *flags*.

Once this function successfully returns, the resource manager must guarantee that the transaction branch may be either committed or rolled back regardless of failures. A resource manager cannot erase its knowledge of a branch until the transaction manager calls either *xa_commit*() or *xa_rollback*() to complete the branch.

As an optimisation, a resource manager may indicate either that the work performed on behalf of a transaction branch was read-only or that the resource manager was not accessed on behalf of a branch (that is, *xa_prepare*() may return [XA_RDONLY]). In either case, the resource manager may release all resources and forget about the branch.

A transaction manager must issue a separate *xa_prepare*() for each transaction branch that accessed the resource manager on behalf of the global transaction.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*(), identifies the resource manager called from the thread of control.

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC
: This flag indicates that *xa_prepare*() shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*() to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, this function fails, returning [XAER_ASYNC].

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

RETURN VALUE

The function *xa_prepare*() has the following return values:

[XA_RDONLY]
The transaction branch was read-only and has been committed.

[XA_OK]
Normal execution.

[XA_RB∗]
The resource manager did not prepare to commit the work done on behalf of the transaction branch. Upon return, the resource manager has rolled back the branch's work and has released all held resources. The following values may be returned:

[XA_RBROLLBACK]
The resource manager rolled back the transaction branch for an unspecified reason.

[XA_RBCOMMFAIL]
A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager rolled back the transaction branch for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
The resource manager encountered an error in preparing to commit the transaction branch's work. The specified XID may or may not have been prepared.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable. The specified XID may or may not have been prepared.

[XAER_NOTA]
The specified XID is not known by the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_commit( ), *xa_complete*( ), *xa_open*( ), *xa_rollback*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_prepare*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of ∗*xid* after the function completes.

NAME

xa_recover — obtain a list of prepared transaction branches from a resource manager

SYNOPSIS

```
#include "xa.h"

int
xa_recover(XID *xids, long count, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_recover*( ) during recovery to obtain a list of transaction branches that are currently in a prepared or heuristically completed state. The caller points *xids* to an array into which the resource manager places XIDs for these transactions, and sets *count* to the maximum number of XIDs that fit into that array.

So that all XIDs may be returned irrespective of the size of the array *xids*, one or more *xa_recover*( ) calls may be used within a single recovery scan. The *flags* parameter to *xa_recover*( ) defines when a recovery scan should start or end, or start and end. The start of a recovery scan moves a cursor to the start of a list of prepared and heuristically completed transactions. Throughout the recovery scan the cursor marks the current position in that list. Each call advances the cursor past the set of XIDs it returns.

Two consecutive complete recovery scans return the same list of transaction branches unless a transaction manager calls *xa_commit*( ), *xa_forget*( ), *xa_prepare*( ), or *xa_rollback*( ) for that resource manager, or unless that resource manager heuristically completes some branches, between the two recovery scans.

A transaction manager may call this function from any thread of control, but all calls in a given recovery scan must be made by the same thread.

Upon success, *xa_recover*( ) places zero or more XIDs in the space pointed to by *xids*. The function returns the number of XIDs it has placed there. If this value is less than *count*, there are no more XIDs to recover and the current scan ends. (That is, the transaction manager need not call *xa_recover*( ) again with TMENDRSCAN set in *flags*.) Multiple invocations of *xa_recover*( ) retrieve all the prepared and heuristically completed transaction branches.

It is the transaction manager's responsibility to ignore XIDs that do not belong to it.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*( ), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags*:

TMSTARTRSCAN
: This flag indicates that *xa_recover*( ) should start a recovery scan for the thread of control and position the cursor to the start of the list. XIDs are returned from that point. If a recovery scan is already open, the effect is as if the recovery scan were ended and then restarted.

TMENDRSCAN
: This flag indicates that *xa_recover*( ) should end the recovery scan after returning the XIDs. If this flag is used in conjunction with TMSTARTRSCAN, the single *xa_recover*( ) call starts and then ends a scan.

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*. A recovery scan must already be started. XIDs are returned starting at the current cursor position.

RETURN VALUE

The function *xa_recover*() has the following return values:

[≥ 0]
: As a return value, *xa_recover*() normally returns the total number of XIDs it returned in ∗*xids*.

[XAER_RMERR]
: An error occurred in determining the XIDs to return.

[XAER_RMFAIL]
: An error occurred that makes the resource manager unavailable.

[XAER_INVAL]
: The pointer *xids* is NULL and *count* is greater than 0, *count* is negative, an invalid *flags* was specified, or the thread of control does not have a recovery scan open and did not specify TMSTARTRSCAN in *flags*.

[XAER_PROTO]
: The routine was invoked in an improper context. See Chapter 6 for details.

SEE ALSO

xa_commit(), *xa_forget*(), *xa_open*(), *xa_rollback*().

WARNINGS

If *xids* points to a buffer that cannot hold all of the XIDs requested, *xa_recover*() may overwrite the caller's data space.

NAME

xa_rollback — roll back work done on behalf of a transaction branch

SYNOPSIS

```
#include "xa.h"

int
xa_rollback(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_rollback*() to roll back the work performed at a resource manager on behalf of the transaction branch. A branch must be capable of being rolled back until it has successfully committed. Any resources held by the resource manager for the branch are released and those modified are restored to their values at the start of the branch. A transaction manager may call this function from any thread of control.

A resource manager can forget a rolled back transaction branch either after it has notified all associated threads of control of the branch's failure (by returning [XAER_NOTA] or [XA_RB*] on a call to *xa_end*()), or after the transaction manager calls it using *xa_start*() with TMRESUME set in *flags*. The transaction manager must ensure that no new threads of control are allowed to access a resource manager with a rolled back (or marked rollback-only) XID.

In addition, *xa_rollback*() must guarantee that forward progress can be made in releasing resources and restoring them to their initial values. That is, because this function may be used by a transaction manager to resolve deadlocks (defined in a manner dependent on the transaction manager), *xa_rollback*() must not itself be susceptible to indefinite blocking.

If a resource manager already completed the work associated with **xid* heuristically, this function merely reports how the resource manager completed the transaction branch. A resource manager cannot forget about a heuristically completed branch until the transaction manager calls *xa_forget*().

A transaction manager must issue a separate *xa_rollback*() for each transaction branch that accessed the resource manager on behalf of the global transaction.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*(), identifies the resource manager called from the thread of control.

The function's last argument, *flags*, must be set to one of the following values:

TMASYNC
: This flag indicates that *xa_rollback*() shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*() to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager for the same XID, this function fails, returning [XAER_ASYNC].

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

RETURN VALUE

The function *xa_rollback*() has the following return values:

[XA_HEURHAZ]
Due to some failure, the work done on behalf of the specified transaction branch may have been heuristically completed. A resource manager may return this value only if it has successfully prepared ∗*xid*.

[XA_HEURCOM]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was committed. A resource manager may return this value only if it has successfully prepared ∗*xid*.

[XA_HEURRB]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was rolled back. A resource manager may return this value only if it has successfully prepared ∗*xid*.

[XA_HEURMIX]
Due to a heuristic decision, the work done on behalf of the specified transaction branch was partially committed and partially rolled back. A resource manager may return this value only if it has successfully prepared ∗*xid*.

[XA_OK]
Normal execution.

[XA_RB∗]
The resource manager has rolled back the transaction branch's work and has released all held resources. These values are typically returned when the branch was already marked rollback-only. The following values may be returned:

[XA_RBROLLBACK]
The resource manager rolled back the transaction branch for an unspecified reason.

[XA_RBCOMMFAIL]
A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager rolled back the transaction branch for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred in rolling back the transaction branch. The resource manager is free to forget about the branch when returning this error so long as all accessing threads of control have been notified of the branch's state.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable.

[XAER_NOTA]
The specified XID is not known by the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

xa_commit( ), *xa_complete*( ), *xa_forget*( ), *xa_open*( ), *xa_prepare*( ).

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_rollback*( ). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of ∗*xid* after the function completes.

NAME

xa_start — start work on behalf of a transaction branch

SYNOPSIS

```
#include "xa.h"

int
xa_start(XID *xid, int rmid, long flags)
```

DESCRIPTION

A transaction manager calls *xa_start*() to inform a resource manager that an application may do work on behalf of a transaction branch. Since many threads of control can participate in a branch and each one may be invoked more than once, *xa_start*() must recognise whether or not the XID exists. If another thread is accessing the calling thread's resource manager for the same branch, *xa_start*() may block and wait for the active thread to release control of the branch (via *xa_end*()). A transaction manager must call this function from the same thread of control that accesses the resource manager. If the resource manager is doing work outside any global transaction on behalf of the application, *xa_start*() returns [XAER_OUTSIDE].

A transaction manager calls *xa_start*() only for those resource managers that do not have TMREGISTER set in the *flags* element of their **xa_switch_t** structure. Resource managers with TMREGISTER set must use *ax_reg*() to join a transaction branch (see *ax_reg*() for details).

The first argument, *xid*, is a pointer to the XID that a resource manager must associate with the calling thread of control. The transaction manager guarantees the XID to be unique for different transaction branches. The transaction manager may generate a new branch qualifier within the XID when it calls *xa_start*() for a new thread of control association (that is, when TMRESUME is not set in *flags*; see TMRESUME below). If the transaction manager elects to reuse a branch qualifier previously given to the resource manager for the XID, the transaction manager must inform the resource manager that it is doing so (by setting TMJOIN in *flags*; see TMJOIN below).

If the transaction manager generates a new branch qualifier in the XID, this thread is loosely-coupled to the other threads in the same branch. That is, the resource manager may treat this thread's work as a separate global transaction with respect to its isolation policies. If the transaction manager reuses a branch qualifier in the XID, this thread is tightly-coupled to the other threads that share the branch. An RM must guarantee that tightly-coupled threads are treated as a single entity with respect to its isolation policies and that no deadlock occurs within the branch among these tightly-coupled threads.

The argument *rmid*, the same integer that the transaction manager generated when calling *xa_open*(), identifies the resource manager called from the thread of control.

Following are the valid settings of *flags*:

TMJOIN

This flag indicates that the thread of control is joining the work of an existing transaction branch. The resource manager should make available enough transaction context so that tightly-coupled threads are not susceptible to resource deadlock within the branch.

If a resource manager does not recognise **xid*, the function fails, returning [XAER_NOTA]. Note that this flag cannot be used in conjunction with TMRESUME.

TMRESUME
: This flag indicates that a thread of control is resuming work on the specified transaction branch. The resource manager should make available at least the transaction context that is specific to the resource manager, present at the time of the suspend, as if the thread had effectively never been suspended, except that other threads in the global transaction may have affected this context.

  If a resource manager does not recognise **xid*, the function fails, returning [XAER_NOTA]. If the resource manager allows an association to be resumed in a different thread from the one that suspended the work, and the transaction manager expressed its intention to migrate the association (via the TMMIGRATE flag on *xa_end*()), the current thread may be different from the one that suspended the work. Otherwise, the current thread must be the same, or the resource manager returns [XAER_PROTO]. When TMRESUME is set, the transaction manager uses the same XID it used in the *xa_end*() call that suspended the association.

  If **xid* contains a reused branch qualifier, and the transaction manager has multiple outstanding suspended thread associations for **xid*, the following rules apply:

  - The transaction manager can have only one of them outstanding at any time with TMMIGRATE set in *flags*.
  - Moreover, the transaction manager cannot resume this association in a thread that currently has a non-migratable suspended association.

  These rules prevent ambiguity as to which context is restored.

TMNOWAIT
: When this flag is set and a blocking condition exists, *xa_start*() returns [XA_RETRY] and the resource manager does not associate the calling thread of control with **xid* (that is, the call has no effect). Note that this flag cannot be used in conjunction with TMASYNC.

TMASYNC
: This flag indicates that *xa_start*() shall be performed asynchronously. Upon success, the function returns a positive value (called a handle) that the caller can use as an argument to *xa_complete*() to wait for the operation to complete. If the calling thread of control already has an asynchronous operation pending at the same resource manager, this function fails, returning [XAER_ASYNC].

TMNOFLAGS
: This flag must be used when no other flags are set in *flags*.

RETURN VALUE

The function *xa_start*() has the following return values:

[XA_RETRY]
: TMNOWAIT was set in *flags* and a blocking condition exists.

[XA_OK]
Normal execution.

[XA_RB*]
The resource manager has not associated the transaction branch with the thread of control and has marked **xid* rollback-only. The following values may be returned regardless of the setting of *flags*:

[XA_RBROLLBACK]
The resource manager marked the transaction branch rollback-only for an unspecified reason.

[XA_RBCOMMFAIL]
A communication failure occurred within the resource manager.

[XA_RBDEADLOCK]
The resource manager detected a deadlock.

[XA_RBINTEGRITY]
The resource manager detected a violation of the integrity of its resources.

[XA_RBOTHER]
The resource manager marked the transaction branch rollback-only for a reason not on this list.

[XA_RBPROTO]
A protocol error occurred within the resource manager.

[XA_RBTIMEOUT]
The work represented by this transaction branch took too long.

[XA_RBTRANSIENT]
The resource manager detected a transient error.

[XAER_ASYNC]
TMASYNC was set in *flags*, and either the maximum number of outstanding asynchronous operations has been exceeded, or TMUSEASYNC is not set in the *flags* element of the resource manager's **xa_switch_t** structure.

[XAER_RMERR]
An error occurred in associating the transaction branch with the thread of control.

[XAER_RMFAIL]
An error occurred that makes the resource manager unavailable.

[XAER_DUPID]
If neither TMRESUME nor TMJOIN was set in *flags* (indicating the initial use of **xid*) and the XID already exists within the resource manager, the resource manager must return [XAER_DUPID]. The resource manager failed to associate the transaction branch with the thread of control.

[XAER_OUTSIDE]
The resource manager is doing work outside any global transaction on behalf of the application.

[XAER_NOTA]
Either TMRESUME or TMJOIN was set in *flags*, and the specified XID is not known by the resource manager.

[XAER_INVAL]
Invalid arguments were specified.

[XAER_PROTO]
The routine was invoked in an improper context. See Chapter 6 for details.

The function returns a positive value upon success if the caller set TMASYNC (see above) in *flags*.

SEE ALSO

ax_reg(), *xa_complete*(), *xa_end*(), *xa_open*().

WARNINGS

From the resource manager's perspective, the pointer *xid* is valid only for the duration of the call to *xa_start*(). That is, once the function completes, either synchronously or asynchronously, the transaction manager is allowed to invalidate where *xid* points. Resource managers are encouraged to use private copies of ∗*xid* after the function completes.

Chapter 6

State Tables

This chapter contains state tables that show legal calling sequences for the XA routines. TMs must sequence their use of the XA routines so that the calling thread of control makes legal transitions through each applicable table in this chapter. That is, any TM must, on behalf of each AP thread of control:

- open and close each RM as shown in Table 6-1 on page 58
- associate itself with, and dissociate itself from, transaction branches as shown in Table 6-2 on page 59 or Table 6-3 on page 60, whichever applies
- advance any transaction branch toward completion through legal transitions as shown in Table 6-4 on page 62
- make legal transitions through Table 6-5 on page 63, whenever the TM calls XA routines in the asynchronous mode.

Table 6-5 on page 63 is the only table that addresses the asynchronous mode of XA routines. The other tables also describe routines that can be called asynchronously. In this case, the tables view the XA call that initiates an operation, and the *xa_complete*() call that shows that the operation is complete, as a single event.

Interpretation of the Tables

A single call may make transitions in more than one of the state tables. Services that are not pertinent to a given state table are omitted from that table.

All the tables describe the state of a thread of control with respect to a particular RM. That is, the tables indicate the validity of a sequence of XA calls only if the calls all pertain to the same RM. The thread of control could be dealing with other RMs at the same time, which might be in entirely different states. Each state table indicates the valid initial state or states for such a sequence; it is not always the leftmost state (the state with the zero subscript).

Table 6-2 on page 59, Table 6-3 on page 60 and Table 6-4 on page 62 describe the sequence of calls with respect to the progress of a particular XID. Other XIDs within the same RM thread of control may be in different states as they progress from initial creation through completion, except that a thread can have only one active association at a time. Thus, while one XID may be actively associated in a thread of control, the same thread of control may make branch completion calls for other XIDs.

An entry under a particular state in the table asserts that an XA routine can be called in that state, and shows the resulting state. A blank entry asserts that it is an error to call the routine in that state. The routine should return the protocol error [XAER_PROTO], unless another error code that gives more specific information also applies.

Notation

Sometimes a routine makes a state transition only when the caller gives it certain input, or only when the routine returns certain output (such as a return code). Specific state table entries describe these cases. The tables describe input to the routine in parentheses, even though that may not be the exact syntax used; for example, *xa_end*(TMFAIL) describes a call to *xa_end*() in which the caller sets the TMFAIL bit in the *flags* argument. The tables denote output from the routine, including return status, using an arrow (→) followed by the specific output.

For example, the legend

xa_end → [XA_RB]

describes the case where a call to *xa_end*() returns one of the [XA_RB∗] codes.

A general state table entry (one that does not show flags or output values) describes all remaining cases of calls to that routine. These general entries assume the routine returns success. (The *xa_* routines return the [XA_OK] code; the *ax_* routines return [TM_OK].) Calls that return failure do not make state transitions, except where described by specific state table entries.

The notation *xa_∗* refers to all applicable *xa_* routines.

6.1 Resource Manager Initialisation

For each thread of control, each RM is either open or closed. The *initial state* is closed (R_0). The *xa_open*() and *xa_close*() routines move an RM between these states. Redundant uses of these routines are valid, as Table 6-1 shows:

XA Routines	**Resource Manager States**	
	Un-initialised	**Initialised**
	R_0	R_1
xa_open( )	R_1	R_1
xa_close	R_0	R_0

Table **6-1** State Table for Resource Manager Initialisation

A transition to R_1 enables the use of Table 6-2 on page 59 to Table 6-4 on page 62 inclusive. The state R_0 appears in these tables to illustrate that closing an RM precludes its use in global transactions. At this point, Table 6-1 governs legal sequences.

In Table 6-2 on page 59 to Table 6-4 on page 62 a return of [XAER_RMFAIL] on any routine causes a state transition in that thread to state R_0.

6.2 Association of Threads of Control with Transactions

Table 6-2 shows the state of an association between a thread of control and a transaction branch. (See Table 6-3 on page 60 for RMs that dynamically register with a TM.) Valid initial states of association for a thread of control are T_0 and T_2. (The *Association Suspended* state, T_2, includes the case where a thread has suspended an association migratably. After returning to T_0, any thread can re-enter this table in column T_2 to resume or end that other association.)

Table 6-2 makes the following assumptions:

- The calling thread remains in state R_1.
- The RM does not have TMREGISTER set in its switch.
- The caller passes the same *rmid* and XID as arguments to each applicable routine listed below.

Table 6-2 shows the effect of *xa_start*() and *xa_end*() on the thread of control's association with a single transaction branch. These routines may also change the state of the branch itself. Therefore, Table 6-4 on page 62 further constrains their use.

If a thread suspends its association, it can perform work on behalf of other transaction branches before resuming the suspended association.

XA Routines	**Transaction Branch Association States**		
	Not Associated T_0	**Associated** T_1	**Association Suspended** T_2
xa_start( )	T_1		
xa_start(TMRESUME)			T_1
xa_start(TMRESUME) → [XA_RB]			T_0
xa_end(TMSUSPEND)		T_2	
xa_end(TMSUSPEND) → [XA_RB]		T_0	
xa_end(TMSUCCESS)		T_0	T_0
xa_end(TMFAIL)		T_0	T_0
xa_open( )	T_0	T_1	T_2
xa_recover( )	T_0	T_1	T_2
xa_close( )	R_0		R_0
*xa_**( ) → [XAER_RMFAIL]	R_0	R_0	R_0

Table **6-2** State Table for Transaction Branch Association

6.2.1 Dynamic Registration of Threads

Table 6-3 shows the state of an association between a thread of control and a transaction branch. This table is for RMs that dynamically register with a TM. Valid initial states in Table 6-3 are D_0 and D_2. (The Association Suspended state, D_2, includes the case where a thread has suspended an association migratably. After returning to D_0, any thread can re-enter this table in column D_2 to resume or end that other association.)

The top half of the table shows the legal sequence of calls for an RM thread of control. The bottom half of the table shows the legal sequence of calls for a TM thread of control. The thread of control calling these routines must comply with the applicable half of the table.

Table 6-3 makes the following assumptions:

- The TM calling thread remains in state R_1.
- The RM has TMREGISTER set in its switch.
- The caller passes the same *rmid* and XID as arguments to each applicable routine listed below.
- The same RM and XID are used for the dynamic registration functions.

Table 6-3 defines the behaviour of a single transaction branch in a thread. If a thread suspends its association, it can perform work on behalf of other branches before resuming the suspended association.

XA Routines	**Transaction Branch Association States**			
	Not Registered D_0	**Registered with Valid XID** D_1	**Registration Suspended** D_2	**Registered with NULLXID** D_3
Resource Manager Calls				
ax_reg → valid XID	D_1			
ax_reg → NULLXID	D_3			
ax_reg → [TM_RESUME]			D_1	
ax_unreg				D_0
Transaction Manager Calls				
xa_end(TMSUSPEND)		D_2		
xa_end(TMSUSPEND) → [XA_RB]		D_0		
xa_end(TMSUCCESS)		D_0	D_0	
xa_end(TMFAIL)		D_0	D_0	
xa_open( )	D_0	D_1	D_2	D_3
xa_recover( )	D_0	D_1	D_2	D_3
xa_close( )	R_0		R_0	
*xa_**( ) → [XAER_RMFAIL]	R_0	R_0	R_0	R_0

Table **6-3** State Table for Transaction Branch Association (Dynamic Registration)

6.3 Transaction States

Table 6-4 on page 62 shows the commitment protocol for a transaction branch. Any state listed in Table 6-4 on page 62 except state S_1 is a valid initial state for a thread of control. The table applies to sequential calls by a thread of control that:

- remains in state R_1
- passes the same *XID* in each *xa_* call that requires an *XID*.

Table 6-4 on page 62 shows the effect of *xa_start*() and *xa_end*() on the state of a transaction branch. These routines may also change the thread's association with the branch. Therefore, Table 6-2 on page 59 and Table 6-3 on page 60 further constrain their use.

Table 6-4 on page 62 does not apply to uses of *xa_end*(TMSUSPEND), *xa_start*(TMRESUME) or *ax_reg* in which [TM_RESUME] is returned; the uses of these are constrained by Table 6-2 on page 59 and Table 6-3 on page 60 and the following rules:

- *xa_end*(TMSUSPEND|TMMIGRATE) may be used only if no other thread association for this branch was suspended with the TMMIGRATE flag. (This rule ensures that there exists at most one migratable suspended association for a branch.)
- *xa_start*(TMRESUME) may be used, and *ax_reg* may return [TM_RESUME], only on a branch that has at least one suspended association. That suspended association must either have been suspended non-migratably by the acting thread or suspended migratably by any thread. If both conditions are true, the association which was suspended non-migratably by the acting thread is the one resumed.
- *xa_end*(TMSUCCESS) may be used only on a branch that is associated with the current thread or that has at least one suspended association. If the branch is associated with the current thread, it is that association which is ended. Otherwise, a suspended association ends, as though an implicit *xa_start*(TMRESUME) were performed (see above) before the *xa_end*(TMSUCCESS).

XA Routines	Transaction Branch States					
	Non-existent Transaction S_0	**Active** S_1	**Idle** S_2	**Prepared** S_3	**Rollback Only** S_4	**Heuristically Completed** S_5
† *xa_start*( )	S_1		S_1			
‡ *xa_start*( ) → [XA_RB]			S_4			
xa_end( )		S_2				
‡ *xa_end*( ) → [XA_RB]		S_4				
xa_prepare( )			S_3			
xa_prepare( ) → ‡ [XA_RDONLY] or [XA_RB]			S_0			
xa_prepare( ) → [XAER_RMERR]			S_2			
xa_commit( ) → [XA_OK] or [XAER_RMERR]			S_0	S_0		
‡ *xa_commit*( ) → [XA_RB]			S_0			
xa_commit( ) → [XA_RETRY]				S_3		
xa_commit( ) → ‡ [XA_HEUR]			S_5	S_5		S_5
xa_rollback( ) → ‡ [XA_OK] or [XA_RB] or [XAER_RMERR]			S_0	S_0	S_0	
xa_rollback( ) → ‡ [XA_HEUR]				S_5	S_5	S_5
xa_forget( )						S_0
xa_forget( ) → [XAER_RMERR]						S_5
xa_open( )	S_0	S_1	S_2	S_3	S_4	S_5
xa_recover( )	S_0	S_1	S_2	S_3	S_4	S_5
xa_close( )	R_0		R_0	R_0	R_0	R_0
*xa_**( ) → [XAER_RMFAIL]	R_0	R_0	R_0	R_0	R_0	R_0

Table **6-4** State Table for Transaction Branches

Notes:

† This row also applies when an applicable RM calls *ax_reg* and the TM informs the RM that its work is on behalf of a transaction branch.

‡ [XA_HEUR] denotes any of [XA_HEURCOM], [XA_HEURRB], [XA_HEURMIX], or [XA_HEURHAZ]. [XA_RB] denotes any return value with a prefix [XA_RB.

6.4 Asynchronous Operations

Table 6-5 describes asynchronous operations. The preceding tables do not take into account asynchronous operations.

Table 6-5 illustrates that once a TM thread makes an asynchronous request to an RM on behalf of an XID, the only valid request the TM thread can give the same RM for that XID is the corresponding *xa_complete*(). For those routines that do not take an XID, the thread must wait for the operation's completion before issuing another request to that same RM. The only routines by which the TM can give additional work to the same RM are *xa_commit*(), *xa_forget*(), *xa_prepare*(), and *xa_rollback*().

The valid **initial state** for a thread of control is A_0.

The asynchronous calls (with TMASYNC) achieve a state transition in Table 6-5 when the function returns a valid handle (a positive value). The table entries describing *xa_complete*() assume that the caller passes this same, valid handle to *xa_complete*().

XA Routines	**Asynchronous Operation States**	
	Initial Call	**Operation Pending**
	A_0	A_1
Any synchronous *xa_*∗ call	A_0	
xa_rollback(TMASYNC)	A_1	
xa_close(TMASYNC)	A_1	
xa_commit(TMASYNC)	A_1	
xa_end(TMASYNC)	A_1	
xa_forget(TMASYNC)	A_1	
xa_open(TMASYNC)	A_1	
xa_prepare(TMASYNC)	A_1	
xa_start(TMASYNC)	A_1	
xa_complete( )		A_0
xa_complete(TMNOWAIT)		A_0
xa_complete(TMNOWAIT) → [XA_RETRY]		A_1

Table **6-5** State Table for Asynchronous Operations

Chapter 7

Implementation Requirements

This chapter summarises the implications on implementors of this specification. It also identifies features of this specification that implementors of RMs or TMs can regard as optional.

These requirements are designed to facilitate portability — specifically, the ability to move a software component to a different DTP system without modifying the source code. It is anticipated that DTP products will be delivered as object modules and that the administrator will control the mix and operation of components at a particular site by:

1. re-linking object modules
2. supplying text strings to the software components (or executing a vendor-supplied procedure that generates suitable text strings).

7.1 Application Program Requirements

Any AP in a DTP system must use a TM and delegate to it responsibility to control and coordinate each global transaction. X/Open will specify a TX interface separately.

The AP is not involved in either the commitment protocol or the recovery process. An AP thread can have only one global transaction active at a time.

The AP may ask for work to be done by calling one or more RMs. It uses the RM's native interface exactly as it would if no TM existed, except that it calls the TM to define global transactions (see Section 7.2.1 on page 68).

7.2 Resource Manager Requirements

The X/Open DTP model affects only RMs operating in the DTP environment. The model puts these constraints on the architecture or implementation of the RM:

- **Interfaces**
  RMs must provide all the *xa_* routines specified in Chapter 3 for use by TMs, even if a particular routine requires no real action by that RM. RMs must also provide a native application program interface (see Section 7.2.1 on page 68). As this is the initial version of the XA interface, RMs must also set to 0 the version element of their switches.

  An RM in an executable module is linked to at most one TM.

- **Ability to recognise XIDs**
  RMs must accept XIDs from TMs. They must associate with the XID all the work they do for an AP. For example, if an RM identifies a thread of control using a process identifier, the RM must map the process identifier to the XID.

  An important attribute of the XID is global uniqueness, based on the exact order of the bits in the data portion of the XID for the lengths specified. Therefore, RMs must not alter in any way the bits in the data portion of the XID. For example, if an RM remotely communicates an XID, it must ensure that the data bits of the XID are not altered by the communication process. That is, the data part of the XID should be treated as an OCTET STRING (opaque data) using terminology defined in the referenced ASN.1 standard and the referenced BER standard.

- **Calling protocol**
  A single instance of an RM must allow multiple threads logically to gain access to it on behalf of the same transaction branch, although it has the option of actually implementing single-threaded access (for example, blocking a thread at the call to *xa_start*() or channelling all accesses through a single back-end process). An RM can use whatever it wishes from the AP's environment (for example, process ID, task ID, or user ID) to identify the calling thread.

  An RM must ensure that each calling thread makes *xa_* calls in a legal sequence, and must return [XAER_PROTO] or other suitable error if the caller violates the state tables (see Chapter 6).

- **Commitment protocol**
  The RM must support the commitment protocol specified in Section 2.3 on page 8. This has the following implications:

  — An RM must provide an *xa_prepare*() routine, and must be able to report whether it can guarantee its ability to commit the transaction branch. If it reports that it can, it must reserve all the resources needed to commit the branch. It must hold those resources until the TM directs it either to commit or roll back the branch.

  — An RM must support the one-phase commit optimisation (see Section 2.3.2 on page 8). That is, it must allow *xa_commit*() (specifying TMONEPHASE) even if it has not yet received *xa_prepare*() for the transaction branch in question.

- **Support for Recovery**
  An RM must track the status of all transaction branches in which it is involved. After responding affirmatively to the TM's *xa_prepare*() call, the RM cannot erase its knowledge of the branch, or of the work it has done in support of that branch, until it receives and successfully performs the TM's order to commit or roll back the branch. If an RM heuristically completes a branch, it cannot forget the branch until explicitly permitted to by the TM. On command, an RM must return a list of all its branches that it has prepared to commit or has heuristically completed.

  When an RM recovers from its own failure, it recovers prepared and heuristically completed transaction branches. It forgets all other branches.

- **Public information**
  An RM product must publish the following information:

  — The name of its **xa_switch_t** structure. This switch gives RM entry points and other information; Section 4.3 on page 21 specifies its structure. Multiple RMs may share the same **xa_switch_t** structure. Each pointer must point to an actual routine, even pointers that are theoretically unused on that RM. These routines must return in an orderly way, in case they are mistakenly called. (The RM does not need to publish the names of the individual routines.)

  — The text of the string, within the RM switch, that specifies the name of the RM.

  — The forms of the information strings that its *xa_open*() and *xa_close*() routines accept, and the way that different *xa_open*() strings specify different resource domains for different RM instances.

  — The names of libraries or object files, in the correct sequence, that the administrator must use when linking APs with the RM.

  — Any semantics in the native interface that affect global transaction processing — for example, specifying isolation level, transaction completion, or effects that differ from non-DTP operation.

  X/Open will specify how an RM provider can guarantee that its names do not conflict with the name of any other RM product or version produced by that or another organisation.

- **Implementor options**
  Each RM has the option of implementing these features:

  — **Open/close informational strings**
  Any RM may accept a null string in place of the informational string argument on calls to its *xa_open*() and *xa_close*() routines. (If it does so, it must publish this fact.)

  — **Protocol optimisations**
  The read-only optimisation discussed in Section 2.3.2 on page 8 is optional.

  — **Association migration**
  An RM can allow a TM to resume a suspended association in a thread of control other than the one where the suspension occurred.

— **Branch identification**
An RM can use the *bqual* component of the XID structure to let different branches of the same global transaction prepare to commit at different times, and to avoid deadlock (see Section 4.2 on page 19).

— **Dynamic registration**
This feature, as described in Section 3.3.1 on page 15, is optional.

— **Asynchrony**
Support for the asynchronous mode discussed in Section 3.5 on page 18 is optional. If the TMUSEASYNC flag is set in the RM's switch, the RM must support the asynchronous mode. It may still complete some requests synchronously, either when the TM makes the original asynchronous call or when the TM calls *xa_complete*(). If the TMUSEASYNC flag is not set in the RM's switch and the TM makes an asynchronous call, the RM must return [XAER_ASYNC].

— **Heuristics**
As described in the *xa_commit*() and *xa_rollback*() reference manual pages, an RM can report that it has heuristically completed the transaction branch. This feature is optional.

7.2.1 The Application Program (Native) Interface

RMs must provide a well-defined native interface that APs can use to request work. To maximise portability, all AP-RM interfaces should adhere to any applicable X/Open publication. For example, a relational DBMS RM should use the SQL defined in the referenced **SQL** specification.

In the DTP environment, RMs must rely on the TM to manage global transactions. Some RMs, such as some Indexed Sequential Access Method (ISAM) file managers, have no concept of transactions. So this is a new requirement, but it does not change the native interface. In other RMs, such as SQL RDBMSs, the native interface defines transactions. An AP must not use these services in a DTP context, since TMs have no knowledge of an RM's transaction. For example, the application program interface for an SQL RM in the DTP environment must not let use of these statements affect the global transaction:

```
EXEC SQL COMMIT WORK
EXEC SQL ROLLBACK WORK
```

In addition, any service in the native AP-RM interface that affects its own commitment, or any non-standard service that has an effect on transactions, must not be used within a global transaction.

7.3 Transaction Manager Requirements

- **Service interfaces**
  TMs must use the *xa_* routines the RM provides (see Chapter 3) to coordinate the work of all the local RMs that the AP uses. TMs must call *xa_open*() and *xa_close*() on any local RM associated with the TM. Each TM must ensure that each of its *xa_* calls addresses the correct RM.

  TMs must be written to handle consistently any information or status that an RM can legally return. A TM must assume that it may be linked with RMs that use any RM options that this specification allows, including multiple RMs in a single AP object program, dynamic registration of RMs, RMs that make heuristic decisions, and RMs that use the read-only protocol optimisation.

- **Transaction identifiers**
  A TM must generate XIDs conforming to the structure described in Section 4.2 on page 19. They must be globally unique and must adequately describe the transaction branch. To guarantee global uniqueness, the TM should use an ISO OBJECT IDENTIFIER (see the referenced ASN.1 standard and the referenced BER standard) that the TM knows is unique within the *gtrid* component of the XID. The TM should also use an ISO OBJECT IDENTIFIER within the *bqual* field, but the same OBJECT IDENTIFIER can be used for all XIDs that a given TM generates. The *bqual* fields identify the TM that generated them, and identify the transaction branch with which they are associated.

  Failure to use the ISO OBJECT IDENTIFIER format for XIDs could cause interoperability problems when multiple TMs are either involved in the same global transaction or affect shared resources.

- **Public information**
  A TM product must publish the following information:

  — linking directions for producing an application object program — these directions must describe how to link RM and TM libraries, and how to incorporate the switch from each RM

  — instructions for generating or locating appropriate informational strings that the TM uses when it calls *xa_open*() or *xa_close*() for each RM

- **Implementor options**
  The one-phase commit protocol optimisation (see Section 2.3.2 on page 8) and the asynchronous and non-blocking modes for calling RMs (see Section 3.5 on page 18) are optional.

Appendix A

Complete Text of "xa.h"

This appendix specifies the complete text of an "**xa.h**" file in both ANSI C (see the referenced C standard) and Common Usage C.

```
/*
 * Start of xa.h header
 *
 * Define a symbol to prevent multiple inclusions of this header file
 */
#ifndef XA_H
#define XA_H
/*
 * Transaction branch identification: XID and NULLXID:
 */
#define XIDDATASIZE     128     /* size in bytes */
#define MAXGTRIDSIZE    64      /* maximum size in bytes of gtrid */
#define MAXBQUALSIZE    64      /* maximum size in bytes of bqual */
struct xid_t {
  long formatID;                /* format identifier */
  long gtrid_length;            /* value from 1 through 64 */
  long bqual_length;            /* value from 1 through 64 */
  char data[XIDDATASIZE];
  };
typedef struct xid_t XID;
/*
 * A value of -1 in formatID means that the XID is null.
 */
/*
 * Declarations of routines by which RMs call TMs:
 */
#ifdef __STDC__
extern int ax_reg(int, XID *, long);
extern int ax_unreg(int, long);
#else /* ifndef __STDC__ */
extern int ax_reg();
extern int ax_unreg();
#endif /* ifndef __STDC__ */
/*
 * XA Switch Data Structure
 */
#define RMNAMESZ        32      /* length of resource manager name, */
                                /* including the null terminator */
define MAXINFOSIZE      256     /* maximum size in bytes of xa_info
                                /* strings, including the null terminator*/
struct xa_switch_t {
  char name[RMNAMESZ];          /* name of resource manager */
  long flags;                   /* options specific to the resource manager */
  long version;                 /* must be 0 */
#ifdef __STDC__
  int (*xa_open_entry)(char *, int, long);  /* xa_open function pointer*/
  int (*xa_close_entry)(char *, int, long); /* xa_close function pointer*/
  int (*xa_start_entry)(XID *, int, long);  /* xa_start function pointer*/
  int (*xa_end_entry)(XID *, int, long);    /* xa_end function pointer*/
```

```
  int (*xa_rollback_entry)(XID *, int, long);
                                        /* xa_rollback function pointer*/
  int (*xa_prepare_entry)(XID *, int, long);/* xa_prepare function pointer*/
  int (*xa_commit_entry)(XID *, int, long); /* xa_commit function pointer*/
  int (*xa_recover_entry)(XID *, long, int, long);
                                        /* xa_recover function pointer*/
  int (*xa_forget_entry)(XID *, int, long); /* xa_forget function pointer*/
  int (*xa_complete_entry)(int *, int *, int, long);
                                        /* xa_complete function pointer*/
#else /* ifndef __STDC__ */
  int (*xa_open_entry)();               /* xa_open function pointer */
  int (*xa_close_entry)();              /* xa_close function pointer */
  int (*xa_start_entry)();              /* xa_start function pointer */
  int (*xa_end_entry)();                /* xa_end function pointer */
  int (*xa_rollback_entry)();           /* xa_rollback function pointer*/
  int (*xa_prepare_entry)();            /* xa_prepare function pointer */
  int (*xa_commit_entry)();             /* xa_commit function pointer */
  int (*xa_recover_entry)();            /* xa_recover function pointer */
  int (*xa_forget_entry)();             /* xa_forget function pointer */
  int (*xa_complete_entry)();           /* xa_complete function pointer*/
#endif /* ifndef __STDC__ */
  };
/*
 * Flag definitions for the RM switch
 */
#define TMNOFLAGS    0x00000000L     /* no resource manager features
                                     selected */
#define TMREGISTER   0x00000001L     /* resource manager dynamically
                                     registers */
#define TMNOMIGRATE  0x00000002L     /* resource manager does not support
                                     association migration */
#define TMUSEASYNC   0x00000004L     /* resource manager supports
                                     asynchronous operations */
/*
 * Flag definitions for xa_ and ax_ routines
 */
/* use TMNOFLAGS, defined above, when not specifying other flags */
#define TMASYNC      0x80000000L     /* perform routine asynchronously */
#define TMONEPHASE   0x40000000L     /* caller is using one-phase commit
                                     optimisation */
#define TMFAIL       0x20000000L     /* dissociates caller and marks
                                     transaction branch rollback-only */
#define TMNOWAIT     0x10000000L     /* return if blocking condition exists */
#define TMRESUME     0x08000000L     /* caller is resuming association
                                     with suspended transaction branch */
#define TMSUCCESS    0x04000000L     /* dissociate caller from transaction
                                     branch*/
#define TMSUSPEND    0x02000000L     /* caller is suspending, not ending,
                                     association */
#define TMSTARTRSCAN 0x01000000L     /* start a recovery scan */
#define TMENDRSCAN   0x00800000L     /* end a recovery scan */
#define TMMULTIPLE   0x00400000L     /* wait for any asynchronous operation */
#define TMJOIN       0x00200000L     /* caller is joining existing transaction
                                     branch */
#define TMMIGRATE    0x00100000L     /* caller intends to perform migration */
```

```
/*
 * ax_() return codes (transaction manager reports to resource manager)
 */
#define TM_JOIN         2    /* caller is joining existing transaction
                                branch */
#define TM_RESUME       1    /* caller is resuming association with
                                suspended transaction branch */
#define TM_OK           0    /* normal execution */
#define TMER_TMERR      -1   /* an error occurred in the transaction
                                manager */
#define TMER_INVAL      -2   /* invalid arguments were given */
#define TMER_PROTO      -3   /* routine invoked in an improper context */
/*
 * xa_() return codes (resource manager reports to transaction manager)
 */
#define XA_RBBASE       100             /* the inclusive lower bound of the
                                           rollback codes */
#define XA_RBROLLBACK   XA_RBBASE       /* the rollback was caused by an
                                           unspecified reason */
#define XA_RBCOMMFAIL   XA_RBBASE+1     /* the rollback was caused by a
                                           communication failure */
#define XA_RBDEADLOCK   XA_RBBASE+2     /* a deadlock was detected */
#define XA_RBINTEGRITY  XA_RBBASE+3     /* a condition that violates the
                                           integrity of the resources
                                           was detected */
#define XA_RBOTHER      XA_RBBASE+4     /* the resource manager rolled back
                                           the transaction branch for
                                           a reason not on this list */
#define XA_RBPROTO      XA_RBBASE+5     /* a protocol error occurred in the
                                           resource manager */
#define XA_RBTIMEOUT    XA_RBBASE+6     /* a transaction branch took too
                                           long */
#define XA_RBTRANSIENT  XA_RBBASE+7     /* may retry the transaction branch */
#define XA_RBEND        XA_RBTRANSIENT  /* the inclusive upper bound of the
                                           rollback codes */

#define XA_NOMIGRATE    9    /* resumption must occur where
                                suspension occurred */
#define XA_HEURHAZ      8    /* the transaction branch may have
                                been heuristically completed */
#define XA_HEURCOM      7    /* the transaction branch has been
                                heuristically committed */
#define XA_HEURRB       6    /* the transaction branch has been
                                heuristically rolled back */
#define XA_HEURMIX      5    /* the transaction branch has been
                                heuristically committed and rolled back */
#define XA_RETRY        4    /* routine returned with no effect and
                                may be reissued */
#define XA_RDONLY       3    /* the transaction branch was read-only and
                                has been committed */
#define XA_OK           0    /* normal execution */
```

```
#define XAER_ASYNC      -2   /* asynchronous operation already outstanding */
#define XAER_RMERR      -3   /* a resource manager error occurred in the
                             transaction branch */
#define XAER_NOTA       -4   /* the XID is not valid */
#define XAER_INVAL      -5   /* invalid arguments were given */
#define XAER_PROTO      -6   /* routine invoked in an improper context */
#define XAER_RMFAIL     -7   /* resource manager unavailable */
#define XAER_DUPID      -8   /* the XID already exists */
#define XAER_OUTSIDE    -9   /* resource manager doing work outside */
                             /* global transaction */
#endif /* ifndef XA_H */
/*
 * End of xa.h header
 */
```

Index