

INTERNATIONAL STANDARD

ISO/IEC
14834

First edition
1996-08-15

Information technology — Distributed Transaction Processing — The XA Specification

*Technologies de l'information — Traitement transactionnel réparti —
La spécification XA*



Reference number
ISO/IEC 14834:1996(E)

AVA **Documents**
Document Sharing Hub

Contents

Chapter 1	General	1
1.1	Scope	1
1.2	X/Open DTP Model.....	1
1.3	Document Structure	2
1.4	Normative References	2
Chapter 2	Model and Definitions.....	3
2.1	X/Open DTP Model.....	4
2.2	Definitions.....	5
2.2.1	Transaction	5
2.2.2	Distributed Transaction Processing	5
2.2.3	Application Program.....	5
2.2.4	Resource Manager	5
2.2.5	Global Transactions	6
2.2.6	Transaction Branches	6
2.2.7	Transaction Manager	6
2.2.8	Thread of Control.....	7
2.2.9	Tightly- and Loosely-coupled Threads	7
2.3	Transaction Completion and Recovery	8
2.3.1	Rolling Back the Global Transaction	8
2.3.2	Protocol Optimisations	8
2.3.3	Heuristic Branch Completion.....	9
2.3.4	Failures and Recovery	9
Chapter 3	Interface Overview.....	11
3.1	Index to Services in the XA Interface	12
3.2	Opening and Closing Resource Managers	13
3.3	Association of Threads with Transaction Branches.....	14
3.3.1	Registration of Resource Managers	15
3.4	Branch Completion	16
3.5	Synchronous, Non-blocking, and Asynchronous Modes	17
3.6	Failure Recovery.....	17
Chapter 4	The "xa.h" Header.....	19
4.1	Naming Conventions.....	19
4.2	Transaction Identification.....	19
4.3	Resource Manager Switch	21
4.4	Flag Definitions	22

© ISO/IEC 1996

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

	4.5	Return Codes	24
Chapter	5	Reference Manual Pages	27
		<i>ax_reg()</i>	28
		<i>ax_unreg()</i>	31
		<i>xa_close()</i>	32
		<i>xa_commit()</i>	34
		<i>xa_complete()</i>	37
		<i>xa_end()</i>	38
		<i>xa_forget()</i>	41
		<i>xa_open()</i>	43
		<i>xa_prepare()</i>	45
		<i>xa_recover()</i>	48
		<i>xa_rollback()</i>	50
		<i>xa_start()</i>	53
Chapter	6	State Tables	57
	6.1	Resource Manager Initialisation	58
	6.2	Association of Threads of Control with Transactions	59
	6.2.1	Dynamic Registration of Threads	59
	6.3	Transaction States	61
	6.4	Asynchronous Operations	63
Chapter	7	Implementation Requirements	65
	7.1	Application Program Requirements	65
	7.2	Resource Manager Requirements	66
	7.2.1	The Application Program (Native) Interface	68
	7.3	Transaction Manager Requirements	69
Appendix	A	Complete Text of "xa.h"	71
Appendix	B	DTP Model — Introduction	75
	B.1	Overview	75
	B.2	Benefits of X/Open DTP	76
	B.3	Areas Not Addressed	76
	B.4	Relationship to International Standards	76
Appendix	C	DTP Model — Definitions	77
	C.1	Transaction Definitions	77
	C.2	Model Definitions	79
Appendix	D	DTP Model — The Model	81
	D.1	Functional Model	82
	D.2	Functional Components	83
	D.2.1	Application Program (AP)	83
	D.2.2	Transaction Manager (TM)	83
	D.2.3	Resource Manager (RM)	83
	D.2.4	Communication Resource Manager (CRM)	84

D.3	Interfaces between Functional Components	85
D.3.1	Functional Component Interfaces	85
D.3.2	Data Interfaces	87
D.4	Activity Involving a Single AP	88
D.4.1	Transaction Initiation	88
D.4.2	Transaction Association	88
D.4.3	Transaction Commitment	88
D.4.4	Transaction Rollback.....	89
D.4.5	Heuristic Transaction Completion	89
D.4.6	Recovery after Failure.....	90
D.5	Distributed Communication Facilities	91
D.5.1	Communication within TM Domains	91
D.5.2	Communication across TM Domains	91
D.5.3	Sharing Resources across TM Domains	91
D.5.4	Global Transaction Demarcation	91
D.5.5	Global Transaction Tree Structure	91
D.5.6	Global Transactions and the Transaction Tree	92
D.5.7	Tightly- and Loosely-coupled Threads	93
D.5.8	Commitment Coordination.....	93
D.6	Activity Involving Two or More APs.....	94
D.6.1	Transaction Initiation	94
D.6.2	Transaction Association	94
D.6.3	Transaction Commitment	94
D.6.4	Transaction Rollback.....	95
D.6.5	Heuristic Transaction Completion	95
D.6.6	Recovery after Failure.....	95
D.7	CRM Communication Paradigms with APs	96
D.7.1	The TxRPC Interface	96
D.7.2	The XATMI Interface	96
D.7.3	The CPI-C Version 2 Interface	97
D.7.4	Relationships between the Communication Paradigms.....	97
D.8	High-level TP Language.....	98
Appendix E	DTP Model — Frequently Asked Questions.....	99
Appendix F	Bibliography.....	103
	Index.....	105
 List of Figures		
2-1	Functional Components and Interfaces	4
3-1	The XA Interface.....	11
C-1	A TM Domain with Four Instances.....	79
D-1	Functional Components and Interfaces	82
D-2	Global Transaction Tree Structure	92
E-1	Projection of Model onto Processes	101

List of Tables

4-1	Flags used in Particular Function Calls	23
6-1	State Table for Resource Manager Initialisation.....	58
6-2	State Table for Transaction Branch Association.....	59
6-3	State Table for Transaction Branch Association (Dynamic Registration)	60
6-4	State Table for Transaction Branches	62
6-5	State Table for Asynchronous Operations.....	63

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 14834 was prepared by X/Open Company Ltd. (as XO/CAE/91/300) and was adopted, under a special "fast-track procedure", by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

Appendix A forms an integral part of this International Standard. Appendices B to F are for information only.

Introduction

(This introduction is not a normative part of ISO/IEC 14834, Information technology—Distributed Transaction Processing—The XA Specification, but is included for information only.)

This International Standard specifies the bidirectional interface between a transaction manager and resource manager (the XA interface) in an X/Open Distributed Transaction Processing (DTP) environment. It is based on X/Open CAE Specification, Distributed Transaction Processing: The XA Specification (December 1991). This International Standard is technically identical to the X/Open version. For informative purposes, this International Standard also contains the text of the X/Open DTP Reference Model Version 3 which X/Open has published as a separate Guide.

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[®] is a registered trade mark, and the "X" device is a trade mark, of X/Open Company Limited.

Information technology — Distributed Transaction Processing — The XA Specification

Chapter 1: General

1.1 Scope

This International Standard specifies the *XA interface*: the bidirectional interface between a transaction manager and a resource manager in an X/Open Distributed Transaction Processing (DTP) environment. The XA interface is not an ordinary Application Programming Interface (API); it is a system-level interface between DTP software components.

This International Standard is technically identical to X/Open CAE Specification, Distributed Transaction Processing: The XA Specification (December 1991). Like that specification, this International Standard does not define the full aspects of the DTP model that pertain to communication.

1.2 X/Open DTP Model

The X/Open Distributed Transaction Processing (DTP) model is a software architecture that allows multiple application programs to share resources provided by multiple resource managers, and allows their work to be coordinated into global transactions.

The full X/Open DTP model comprises five basic functional components:

- an Application Program (AP), which defines transaction boundaries and specifies actions that constitute a transaction
- Resource Managers (RMs) such as databases or file access systems, which provide access to resources
- a Transaction Manager (TM), which assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for coordinating failure recovery.

- Communication Resource Managers (CRMs), which control communication between distributed applications within or across TM domains.
- a communication protocol, which provides the underlying communication services used by distributed applications and supported by CRMs.

1.3 Document Structure

Relevant definitions and other important concepts that pertain to this International Standard are discussed in Chapter 2. That 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 International Standard 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. Appendix F contains a bibliography.

For informative purposes, this International Standard also contains the text of the X/Open DTP Reference Model Version 2 (November 1993) which X/Open publishes as a separate Guide. (See Appendix B, Appendix C, Appendix D, and Appendix E.)

1.4 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

1. ISO/IEC 8824:1990, *Information technology—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*.
2. ISO/IEC 8825:1990, *Information technology—Open Systems Interconnection—Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*.
3. ISO/IEC 9804:1994, *Information technology—Open Systems Interconnection—Service definition for the commitment, concurrency and recovery service element*.
4. ISO/IEC 9805-1:1994, *Information technology—Open Systems Interconnection—Protocol for the Commitment, Concurrency and Recovery service element: Protocol Specification*.
5. ISO/IEC 9899:1990, *Programming languages—C*.
6. ISO/IEC 10026-1:1992, *Information technology—Open Systems Interconnection—Distributed Transaction Processing—Part 1: OSI TP Model*.
7. ISO/IEC 10026-2:1996, *Information technology—Open Systems Interconnection—Distributed Transaction Processing—Part 2: OSI TP Service*.
8. ISO/IEC 10026-3:1996, *Information technology—Open Systems Interconnection—Distributed Transaction Processing—Part 3: Protocol Specification*.

See Appendix F for bibliographic references.

Chapter 2: Model and Definitions

This chapter discusses the XA interface in general terms and provides necessary background material for the rest of this International Standard. 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 boxes in Figure 2-1 are the functional components and the connecting lines are the interfaces between them. The arrows indicate the directions in which control may flow.

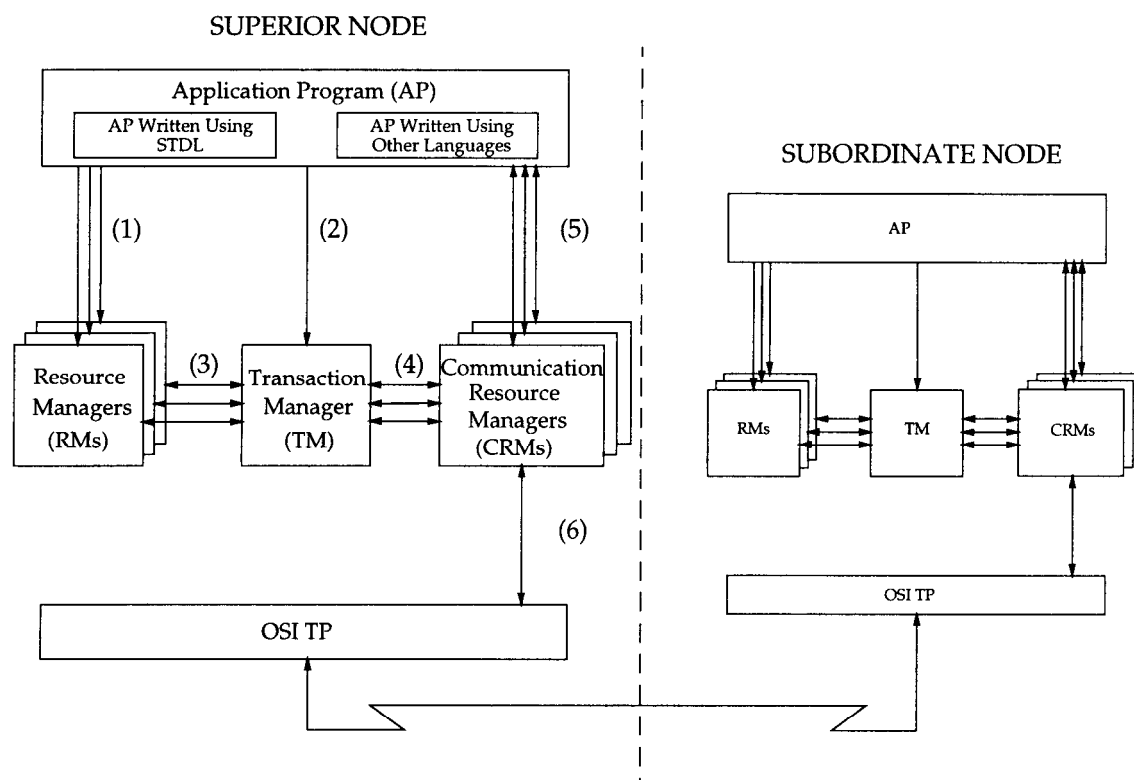


Figure 2-1 Functional Components and Interfaces

The numbers in brackets in Figure 2-1 represent the different X/Open interfaces that are used in the DTP model. The subject of this International Standard is interface (3): the XA interface by which TMs and RMs interact. Descriptions of the functional components relevant to this International Standard can be found in Section 2.2 on page 5. For more details of the the DTP model as shown in Figure 2-1, including definitions of all components and interfaces, see Appendix B, Appendix C, Appendix D, and Appendix E.

2.2 Definitions

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 TP standard (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 International Standard, 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 a standard 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 International Standard 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 International Standard 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 5. 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* (which is normally equivalent to an OSI TP transaction when OSI TP is used as the communication protocol; see the referenced OSI TP standards). 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. (The term *transaction branch* does not necessarily have the same meaning as in OSI TP. The concept is the same, but the parties may be different; see the referenced OSI TP standards.)

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 TP standard (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 TP standards (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 might 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 TP standards, 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. That chapter contains C-language function prototypes.

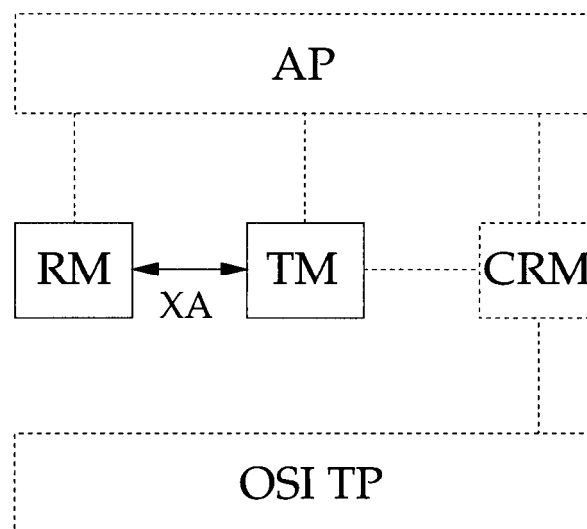


Figure 3-1 The XA Interface

The X/Open DTP model envisages interfaces between each of the AP, RM, TM, CRM, and OSI TP (see Section 2.1 on page 4). 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
<i>ax_reg</i>	Register an RM with a TM.	Section 3.3.1 on page 15
<i>ax_unreg</i>	Unregister an RM with a TM.	Section 3.3.1 on page 15
<i>xa_close</i>	Terminate the AP's use of an RM.	Section 3.2 on page 13
<i>xa_commit</i>	Tell the RM to commit a transaction branch.	Section 3.4 on page 16
<i>xa_complete</i>	Test an asynchronous <i>xa_</i> operation for completion.	Section 3.5 on page 17
<i>xa_end</i>	Dissociate the thread from a transaction branch.	Section 3.3 on page 14
<i>xa_forget</i>	Permit the RM to discard its knowledge of a heuristically-completed transaction branch.	Section 3.4 on page 16
<i>xa_open</i>	Initialise an RM for use by an AP.	Section 3.2 on page 13
<i>xa_prepare</i>	Ask the RM to prepare to commit a transaction branch.	Section 3.4 on page 16
<i>xa_recover</i>	Get a list of XIDs the RM has prepared or heuristically completed.	Section 3.6 on page 17
<i>xa_rollback</i>	Tell the RM to roll back a transaction branch.	Section 3.4 on page 16
<i>xa_start</i>	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 shall 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. (See **Public Information** in Section 7.2 on page 67.)

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()*. This International Standard 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 shall 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 16). 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 shall 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 shall 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 shall 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_`. The 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 standards). 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 — how this is achieved for a specific system is implementation defined (see Chapter 7). 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 17), 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 */
```

The semantics of a flag setting when used on a particular function call are described on the manual page of the call as follows:

Table 4-1 Flags used in Particular Function Calls

Flag	Function
TMASYN	<i>xa_close()</i> on page 32 <i>xa_commit()</i> on page 34 <i>xa_end()</i> on page 38 <i>xa_forget()</i> on page 41 <i>xa_open()</i> on page 43 <i>xa_prepare()</i> on page 45 <i>xa_rollback()</i> on page 50 <i>xa_start()</i> on page 53
TMENDRSCAN	<i>xa_recover()</i> on page 48
TMFAIL	<i>xa_end()</i> on page 38
TMJOIN	<i>xa_start()</i> on page 53
TMMIGRATE	<i>xa_end()</i> on page 38
TMMULTIPLE	<i>xa_complete()</i> on page 37
TMNOWAIT	<i>xa_commit()</i> on page 34 <i>xa_complete()</i> on page 37 <i>xa_start()</i> on page 53
TMONEPHASE	<i>xa_commit()</i> on page 34
TMRESUME	<i>xa_start()</i> on page 53
TMSTARTRSCAN	<i>xa_recover()</i> on page 48
TMSUCCESS	<i>xa_end()</i> on page 38
TMSUSPEND	<i>xa_end()</i> on page 38

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 shall join work of 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, the space to which `xid` points contains a valid XID. If the resource manager's work is outside any global transaction, the space to which `xid` points contains 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 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 the transaction identified by `*xid`, it must return a failure indication to the application. Failure to do so will result in indeterminate behaviour.

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 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]. This directs the resource manager to join the work of the existing transaction branch. If the resource manager receives [TM_JOIN] and does not recognise the transaction identified by `*xid`, it must return a failure indication to the application. Failure to do so will result in indeterminate behaviour. 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. Failure to do so will result in indeterminate behaviour.

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 shall join 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 the transaction identified by **xid*, it must return a failure indication to the application. Failure to do so will result in indeterminate behaviour.

[TM_RESUME]

The resource manager shall resume work on a previously-suspended transaction branch. Failure to do so will result in indeterminate behaviour. 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 the transaction identified by **xid*, it must return a failure indication to the application. Failure to do so will result in indeterminate behaviour. 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; failure to do so will result in indeterminate behaviour.

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()`.