TXSeries™ for Multiplatforms

Encina® Transactional Programming Guide

Version 5.0

SC09-4485-01

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 137.

Second Edition (March 2001)

This edition replaces SC09-4485-00.

Order publications through your IBM representative or through the IBM branch office serving your locality.

Contents

Figures

Tables

About this book

This document explains how to develop distributed, transactional applications using the transactional interfaces provided by Encina®. These interfaces include the Transactional-C (Tran-C) programming language, which is a set of extensions to the C language, and the Encina TX Interface, which is an implementation of the X/Open TX Interface. The Tran-C and Encina TX interfaces simplify the development of transactional applications by providing high-level interfaces to the Encina Toolkit. Transactional applications can use low-level Toolkit interfaces (such as the Encina Transaction Service) when special circumstances require functionality not available in the higher-level interfaces.

Who should read this book

This document is written for programmers who are writing applications using the Encina transactional interfaces. It is assumed that users are familiar with program development in the C programming language and the Open Software Foundation (OSF®) Distributed Computing Environment (DCE).

Document organization

This document has the following organization:

Part 1: *Distributed Transaction Processing*

- "Chapter 1. Introduction to transaction processing" on page 3 explains basic concepts related to distributed transaction processing and the Encina transaction processing environment.
- "Chapter 2. Transactional Interface Definition Language (TIDL)" on page 13 covers both the syntax for TIDL interface definition files and the use of the TIDL preprocessor.

Part 2: *Transactional-C*

- "Chapter 3. Transactional-C concepts" on page 35 provides an overview of the important features and terminology used in Transactional-C and explains the Transactional-C model.
- "Chapter 4. Managing transactions using Tran-C" on page 43 describes how to develop applications using Transactional-C.
- "Chapter 5. Advanced Tran-C programming" on page 81 covers advanced issues related to developing transactional applications with Tran-C.

- "Chapter 6. Compiling Tran-C applications" on page 111 describes various compilation issues that are important to the development of Tran-C applications and provides information about compiling the sample Tran-C application.

Part 3: *Encina TX Interface*

- "Chapter 7. X/Open TX interface for Encina" on page 117 provides a guide to the functions and data types in Encina's implementation of the TX interface for developing transactional applications.

Part 4: *Encina Transaction Service Interface*

- "Chapter 8. Using TRAN for transactional programming" on page 127 provides an overview of the low-level Encina interfaces used in transaction processing, primarily the Transaction Service (TRAN).

Related information

For further information on the topics discussed in this manual, see the following documents:

- *Writing Encina Applications*
- *Encina Monitor Programming Guide*
- *Encina Toolkit Programming Guide*
- *OSF DCE Application Development Reference* and *OSF DCE Application Development Guide* (for information about developing applications that use DCE RPC)

Conventions used in this book

TXSeries documentation uses the following typographical and keying conventions.

Table 1. Conventions used in this book

Convention	Meaning
Bold	Indicates values you must use literally, such as commands, functions, and resource definition attributes and their values. When referring to graphical user interfaces (GUIs), bold also indicates menus, menu items, labels, buttons, icons, and folders.
`Monospace`	Indicates text you must enter at a command prompt. Monospace also indicates screen text and code examples.
Italics	Indicates variable values you must provide (for example, you supply the name of a file for *file_name*). Italics also indicates emphasis and the titles of books.
< >	Enclose the names of keys on the keyboard.

Table 1. Conventions used in this book (continued)

Convention	Meaning
<**Ctrl-***x*>	Where *x* is the name of a key, indicates a control-character sequence. For example, <**Ctrl-c**> means hold down the **Ctrl** key while you press the **c** key.
<**Return**>	Refers to the key labeled with the word Return, the word Enter, or the left arrow.
`%`	Represents the UNIX command-shell prompt for a command that does not require **root** privileges.
`#`	Represents the UNIX command-shell prompt for a command that requires **root** privileges.
`C:\>`	Represents the Windows® command prompt.
>	When used to describe a menu, shows a series of menu selections. For example, "Select **File > New**" means "From the **File** menu, select the **New** command."
Entering commands	When instructed to "enter" or "issue" a command, type the command and then press <**Return**>. For example, the instruction "Enter the **ls** command" means type **ls** at a command prompt and then press <**Return**>.
[ ]	Enclose optional items in syntax descriptions.
{ }	Enclose lists from which you must choose an item in syntax descriptions.
|	Separates items in a list of choices enclosed in **{ }** (braces) in syntax descriptions.
...	Ellipses in syntax descriptions indicate that you can repeat the preceding item one or more times. Ellipses in examples indicate that information was omitted from the example for the sake of brevity.
IN	In function descriptions, indicates parameters whose values are used to pass data to the function. These parameters are not used to return modified data to the calling routine. (Do *not* include the IN declaration in your code.)
OUT	In function descriptions, indicates parameters whose values are used to return modified data to the calling routine. These parameters are not used to pass data to the function. (Do *not* include the OUT declaration in your code.)
INOUT	In function descriptions, indicates parameters whose values are passed to the function, modified by the function, and returned to the calling routine. These parameters serve as both IN and OUT parameters. (Do *not* include the INOUT declaration in your code.)
$CICS	Indicates the full path name where the CICS product is installed; for example, **C:\opt\cics** on Windows or **/opt/cics** on Solaris. If the environment variable named CICS is set to the product path name, you can use the examples exactly as shown; otherwise, you must replace all instances of $CICS with the CICS product path name.
CICS on Open Systems	Refers collectively to the CICS product for all supported UNIX platforms.

Table 1. Conventions used in this book (continued)

Convention	Meaning
TXSeries CICS	Refers collectively to the CICS for AIX, CICS for Solaris, and CICS for Windows products.
CICS	Refers generically to the CICS on Open Systems and CICS for Windows products. References to a specific version of a CICS on Open Systems product are used to highlight differences between CICS on Open Systems products. Other CICS products in the CICS Family are distinguished by their operating system (for example, CICS for OS/2 or IBM mainframe-based CICS for the ESA, MVS, and VSE platforms).

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other WebSphere Application Server Enterprise Edition documentation, send your comments by e-mail to wasdoc@us.ibm.com. Be sure to include the name of the book, the document number of the book, the version of WebSphere Application Server Enterprise Edition, and, if applicable, the specific location of the information you are commenting on (for example, a page number or table number).

Part 1. Distributed Transaction Processing

Chapter 1. Introduction to transaction processing

This chapter provides background information about transaction processing and transactional applications. This chapter also explains threads and their use in transactional applications.

Transactions

A *transaction* is a set of operations that transforms data from one consistent state to another. This set of operations is an indivisible unit of work and in some contexts is referred to as a *logical unit of work* (LUW). The operations that make up a transaction typically consist of requests for existing data, requests to modify existing data, requests to add new data, or any combination of these requests.

Transactions provide several important properties, referred to as the *ACID properties*:

- *Atomicity*: A transaction is either successful or unsuccessful. Either all of the operations that make up a transaction take effect or none take effect. A successful transaction is said to *commit*. An unsuccessful transaction is said to *abort*. Any operations performed by an aborted transaction are undone (*rolled back*) so that its effects are not visible.
- *Consistency*: A transaction transforms distributed data from one consistent state to another. The application program is responsible for ensuring consistency.
- *Isolation*: Each transaction appears to run independently of other transactions that are running concurrently. The effects of a transaction are not visible to others until the transaction completes (commits or aborts). The transactions appear to be *serialized*, with two or more transactions acting as though one completed before the other began, even though they are executed concurrently.
- *Durability*: Also known as *permanence*, this property ensures that once completed, the effects of a transaction are permanent. A subsequent failure (such as abnormal program termination, communications failure, or hardware failure) does not cause the effects to be undone.

Transactions are often used to control and moderate access to a centralized database. Typical uses for transaction processing systems include database-oriented applications such as airline reservation systems and automatic bank-teller machines.

The process of reserving a seat on an airline flight can be used to illustrate the role that transactional properties play in accessing data. For example, a customer requests a seat on an airline flight, and a reservation agent queries the database for the number of seats left on that flight. The answer to the query indicates that one seat is available, and the agent can then reserve that seat.

The isolation property ensures that other transactions that make the same request at the same time cannot reserve the same seat. That seat must be reserved by the transaction until the entire transaction is either completed or canceled—information is not allowed to change while a running transaction depends on it. To provide atomicity, the request for an available seat and the reservation of that seat must both be part of the same transaction. This ensures that if a seat is marked as unavailable, there is a customer reservation associated with that seat. (If the customer decides *not* to reserve the seat, the transaction is canceled, and the seat is marked as available once again.) To provide durability, the centralized database must maintain a permanent record of the reservation even if the database resides on a computer that fails and restarts after the seat is reserved. To provide consistency, the application must update the information in the database correctly to indicate that the seat is unavailable and identify the customer who reserved the seat.

Distributed transactions

A *distributed transaction* is one that runs in multiple processes, possibly on several machines. Each process works for the transaction. Distributed transactions, like local transactions, must adhere to the ACID properties. However, maintaining these properties is greatly complicated for distributed transactions because a failure can occur in any process. Even in the event of such a failure, however, each process must undo any work already done on behalf of the transaction.

A distributed transaction processing system maintains the ACID properties in distributed transactions by using the following features:

- *Recoverable processes*. Recoverable processes are those that log their actions and thus can restore earlier states if a failure occurs.
- *A commit protocol*. A commit protocol enables multiple processes to coordinate the committing or aborting of a transaction. The most common commit protocol, and the one used by Encina, is the two-phase commit protocol.
- *Transactional remote procedure calls*. Transactional remote procedure calls (TRPCs) provide access to distributed information or resources and carry additional information about the identity and state of a transaction.

In a distributed transaction processing environment, both resources and processing capabilities are distributed across different machines. The programs and computer resources that make up a distributed transaction processing system can work together as a single logical unit, sharing access to centralized data resources.

Recoverable processes

Recoverable processes can undo operations on data when a transaction aborts can recover modifications to data in the case of a system failure. To achieve this, recoverable processes can store two types of information: transaction-state information and descriptions of changes to data. This information allows a process to participate in a two-phase commit and ensures isolation and durability.

Transaction-state information must be stored by all recoverable processes, but only processes that manage application data (such as resource managers) must store descriptions of changes to data. Recoverable processes are typically servers that maintain resources and data that client programs use and modify by making transactional RPCs to servers.

Not all processes involved in a distributed transaction need to be recoverable. Encina uses the term *ephemeral* to refer to processes that are not recoverable. In general, clients are ephemeral because they do not interact directly with a resource manager—processes that do not support recoverable data directly are not recoverable.

The two-phase commit protocol

The *two-phase commit protocol*, as the name implies, involves two phases: a prepare phase and a resolution phase. In each transaction, one process (or *participant*) acts as the coordinator. The *coordinator* oversees the activities of the other participants in the transaction to ensure a consistent outcome.

The actions that occur when a transaction completes without aborting are extremely important because this is the point at which a transaction can become permanent. First, one of the participants in the transaction becomes the coordinator of the transaction and confirms whether all other participants are able to complete their work for the transaction. Second, if there are no problems, the transaction is committed. When a transaction commits, the changes associated with that transaction become permanent.

The prepare phase

During the prepare phase, the coordinator requires that all participants in a transaction agree to make the transaction's changes to data permanent. When a participant in a transaction has completed the prepare phase successfully, has logged this fact, and has returned a message to the coordinator stating that it has completed the prepare phase, it is considered *prepared*. Once a participant in a transaction has prepared, that participant can no longer

unilaterally decide to abort the transaction. Its acceptance of the coordinator's prepare request signals that it is able to commit the transaction. If a participant cannot prepare (that is, if it cannot guarantee that it can commit the transaction), the transaction must abort.

When it has received a positive response from each of the participants, the coordinator writes its own prepare record to the log. Logging the prepare phase ensures that even in the event of a system failure, the transaction can complete automatically once all participating systems are available again. This is possible because the log of each application contains both a record of the changes that were made and a record of the fact that the participant was prepared to commit that transaction. Additionally, the existence of the coordinator's prepare record indicates that all the participants were ready to commit. After a system failure, the log files can be played back to restore the recoverable data to a current state. Following the prepare phase, the participants in the transaction must still complete the resolution phase.

The resolution phase

During the resolution phase (sometimes called the *commit phase*), the prepared transaction commits. The coordinator sends a commit notification to all the participants. Each participant writes a log record to indicate that the transaction is committed, releases all resources held on behalf of the transaction, and returns an acknowledgment to the coordinator. Once the changes requested by the transaction have occurred and a commit record has been written for the transaction, the changes made to recoverable storage on behalf of the transaction become permanent. In the event of a system failure, the presence of the commit record in the log indicates that the changes are to be reinstated.

Logging both the prepare and resolution phases enables the participants in the transaction to determine whether the changes made by the transaction are, or must be, made to permanent storage. The changes are thus protected from problems caused by failures in one or more of the participants in the transaction.

Transactional remote procedure calls

A common method of providing access to distributed information or resources is through *remote procedure calls* (RPCs) from one application to another. Using RPCs, application programs can invoke functions and procedures that are available in other applications or on remote systems, thereby accessing data that is used by those other applications or available on those remote systems. When communicating between applications located on different machines, RPCs provide an abstract interface to the hardware that actually sends and receives information across the network.

In general, standard RPCs do not carry enough information to support a transactional environment. RPCs typically identify only the host system and port from which the RPC is issued. This information is not sufficient for a transaction processing environment, where many transactions and RPCs can take place simultaneously, and the transaction that issued the RPC must also be identified.

Transactional remote procedure calls provide stronger guarantees of success and failure than RPCs. Transactional RPCs (TRPCs) carry additional information about the transaction on whose behalf they are acting, providing *exactly-once* semantics for remote procedure interaction. That is, if a TRPC completes successfully, the actions specified in that call are guaranteed to have occurred *once and only once* on the target machine.

TRPCs guarantee that the remote work is executed once if the transaction commits and not at all if it aborts. If a TRPC fails or the initiating transaction is notified of an error, the transaction is aborted; if the remote function performed any updates, those updates are undone. In contrast. a standard RPC that does not complete successfully can fail for a number of reasons and can also leave updated information on the target system.

Transaction processing and Encina

The foundation of the Encina environment is the Encina Toolkit, which is a set of low-level components for building distributed transactional applications. Though it is possible to use the Toolkit components directly, Encina provides higher-level interfaces that are built on top of the Toolkit. These interfaces hide many of the complexities of Toolkit-level programming.

The higher-level interfaces used for writing transactional applications include Transactional-C and the Encina TX interface. The information in this guide focuses on these higher-level interfaces.

Encina also supplies a transactional interface (CPI-C/RR) for writing X/Open-compliant applications that use the Peer-To-Peer Communications Services (PPC). For more information on the CPI-C/RR interface, see the *Encina PPC Services Programming Guide*.

In addition, Encina provides the *Transactional Interface Definition Language* (TIDL), which is an extension of Distributed Computing Environment (DCE) Interface Definition Language (IDL). Developers can use TIDL to define interfaces that include the additional information required for TRPCs. See "Chapter 2. Transactional Interface Definition Language (TIDL)" on page 13 for more information.

Overview of the Encina Toolkit

Encina's transaction processing environment relies on the components of the Encina Toolkit to provide the underlying services required for distributed, transactional applications. The following is an overview of the primary modules in the Encina Toolkit. For complete information about each Toolkit module, see the *Encina Toolkit Programming Guide*.

- **Distributed Transaction Service**: The Transaction Service (TRAN) manages distributed transactions, supports a nested transaction model, and uses a two-phase commit algorithm for insuring integrity in distributed transactions. TRAN provides modular interfaces to enable the use of applications that require specialized recovery or communications mechanisms.
- **Lock Service**: The Lock Service (LOCK) guarantees the isolation property of transactions by enabling transactions to lock resources before accessing or modifying them. LOCK supports several different types of locks, including standard read and write locks.
- **Log Service**: The Log Service (LOG) provides efficient and stable storage for recording the actions of programs while updating recoverable data, ensuring that accurate records of transactions are retained across system shutdowns and restarts.
- **Recovery Service**: The Recovery Service (REC) guarantees the consistency of the permanent data used by a distributed transaction service. It uses log records to undo or re-create transactions.
- **Thread-to-Tid Mapping Service**: The Thread-to-Tid Mapping Service (threadTid) associates transactions with threads, supporting a model in which one active thread in an application works on behalf of one transaction at any given time.
- **Transaction Manager-XA Service**: The Transaction Manager-XA Service (TM-XA) enables transactional applications to initiate, coordinate, and resolve distributed transactions with XA-compliant resource managers.
- **Transactional Remote Procedure Call**: The Transactional Remote Procedure Call Service (TRPC) provides the underlying communications mechanisms used by the entire system, synchronizing the interaction of transactions and enabling transactions to be distributed to other programs and nodes.
- **Volume Service**: The Volume Service (VOL) enables applications to address storage in terms of logical units of disk storage. VOL maintains the storage used by the Log Service to store the data required by the Recovery Service when restarting a recoverable application.

Threads in transactional applications

A single server program typically services requests made by multiple client programs. To expedite the handling of multiple requests, most— client/server development environments provide a mechanism for intraprocess multitasking. Encina supports the use of multiple parallel execution

sequences, called *threads*, within the single address space of a process. A thread, also known as a *lightweight process*, is an execution environment within an address space. Threads that belong to the same parent process normally do not interfere with each other. In most cases, it is not necessary to protect portions of a process's address space from various threads of that process.

Within a threaded environment, interthread communications by using shared data structures is so simple that it is necessary to restrict access to some portions of shared memory. This is done by using *mutual exclusion facilities* (*mutexes*). A mutex is a synchronization object used to ensure that only one thread can be executed in a particular section of code or access a particular portion of memory at a single time. A mutex has essentially two states: *locked* (no other thread can execute the particular piece of code or access the agreed upon memory) and *unlocked* (access is available to any thread). Mutexes can be used to prevent incompatible accesses from occurring, such when one thread is reading some data while another thread is modifying it.

A transactional environment has many different types of access to shared memory, not all of which need to be mutually exclusive. For example, multiple threads can require read access to a portion of shared memory. Although such access is not compatible with another thread modifying the data, all threads that only require read access are compatible with each other. In this situation, mutexes fail to provide an appropriate mutual-exclusion mechanism. To overcome this shortcoming, the Encina Toolkit provides a locking mechanism that provides multiple locking modes for portions of shared memory. The Transactional-C interface provides transactional mutexes and locks for use in situations where finer control of access to shared memory is required; see "Chapter 4. Managing transactions using Tran-C" on page 43 for more information on transactional mutexes and "Locking data in Tran-C applications" on page 89 for more information on locking.

Callbacks, upcalls, and return codes

Encina's support and development environment for transactional applications is based on the modules of the Encina Toolkit. Three mechanisms are used to exchange information between Toolkit modules, within application programs, and between clients and servers:

- **Callback** procedures that an application component can register to be invoked at specific points during a specific transaction
- **Upcall** procedures that, once registered, are invoked at specific points during all transactions
- **Return codes** from calls made directly to library functions or to the interfaces of the Toolkit modules themselves

Callbacks are procedures that are associated with individual transactions; they are used to notify applications that certain events have occurred. Callbacks are

invoked only during calls to TRAN. A call to any TRAN function (whether that function is called directly or indirectly by way of another function) can result in a callback; therefore, a callback can occur from any thread that makes a call to a TRAN function.

Upcalls are procedures that are called to perform the actions required by certain states of a transaction or to progress to the next state of a transaction.

Both callbacks and upcalls are registered with TRAN, which controls and monitors the transitions between transaction states. Registering the upcalls associated with these states is a part of the application initialization process. Registering callbacks is an optional part of the actions of specific transactions.

Return codes are the values returned by a call to a function. Return codes are usually integer values that indicate success or failure.

Outline of an Encina transactional application

Though the functionality provided by different transactional applications can vary greatly, most have a common structure. A typical transactional application consists of three basic stages:

1. **Initialization**: The application initializes the basic services used, such as system and network interfaces, and initializes any local data it uses. In applications using the client/server model, server applications register with a directory service during this stage; client applications query such services to determine the location of an appropriate server. During this stage, the application also initializes any administrative services it requires, such as logging and recovery services, before accepting or initiating any transactions.
2. **Transactions**: The application performs atomic operations on local and remote data in the form of transactions. Any changes made to recoverable data modified by transactions are logged so that these changes are permanent. This means that, once made, these changes survive across transactions, separate runs of the application, and system failures or other restarts.
3. **Cleanup and exit**: The application gracefully terminates administrative modules such as logging and recovery services, completing all pending interaction with the files maintained by those services. The application surrenders its network identifier, and, if the application is a server, unregisters from the network's directory service.

Most applications can use the Encina Monitor programming interface to simplify the initialization stage and the cleanup-and-exit stage. The Encina Monitor automates many of the tasks commonly required in Encina applications. Applications that do not use the Monitor interfaces must often

initialize and terminate components individually. Refer to the *Encina Monitor Programming Guide* for more information about the Encina Monitor.

For the transactions stage of an Encina application, the higher-level interfaces, such as Transactional-C, provide common functionality that simplifies transactional programming. Low-level interfaces like TRAN are more complex but enable flexibility and greater control over transactions. The requirements of an application determine which interfaces are appropriate.

Chapter 2. Transactional Interface Definition Language (TIDL)

This chapter covers both the Transactional Interface Definition Language (TIDL) and the execution of the TIDL compiler. TIDL is an extension to the Distributed Computing Environment (DCE) Interface Definition Language (IDL) that adds transactional semantics to the operations defined for an interface. The information presented in this chapter assumes that you are already familiar with DCE IDL and creating interface definition files; refer to *Writing Encina Applications* for additional background information.

Introduction to TIDL

TIDL is used to define client/server interfaces for Encina transactional applications. Like IDL, TIDL interface definition files contain descriptions of the operations to be performed by using remote procedure calls (RPCs). TIDL, however, also allows you to specify which operations are transactional and which are not. "Using TIDL" on page 14 provides an overview of using TIDL. See "TIDL file format" on page 15 for details on how TIDL differs from IDL and for information about the limitations of TIDL.

The interfaces for a transactional application must be defined in a TIDL file before the client and server application code is written. Once the interfaces are defined, the TIDL file is compiled by using the TIDL compiler. The TIDL compiler is a preprocessor that generates files containing stub code for the client and server applications. See "Files produced by the TIDL compiler" on page 20.

The TIDL compiler is executed by using the **tidl** command. The **tidl** command provides several options that can be used to control the generation of stub files. For example, you can use options to specify include paths, the names of output files, and so on. You can also use them to specify that files are to be generated for DCE clients rather than for Encina clients. "Building clients and servers" on page 25 provides and example that illustrates the steps for generating the necessary files with the TIDL and IDL compilers and how those files are used to build client and server applications. See the reference page for the **tidl** command for more information on the available options.

Using TIDL

This section provides a general overview of how TIDL is used to define interfaces for transactional applications that use the transactional remote procedure call (TRPC) mechanism provided as part of Encina. An example TIDL file is used as an illustration. (For a complete description of TRPC, see the *Encina Toolkit Programming Guide*.)

The TIDL compiler (**tidl**) uses two files to generate the stub and header files for use with DCE RPC:

- A TIDL file containing the interface description to be processed. In addition to stub and header files, the compiler generates an IDL file to be used as input to the DCE IDL compiler. For more details, see "TIDL file format" on page 15 and "Files produced by the TIDL compiler" on page 20.
- A *transactional attribute configuration file* (TACF) that specifies which of the operations defined in the associated TIDL file are to be exported. Using an attribute configuration file enables the same TIDL file to contain multiple definitions for an operation. Operations can be exported selectively for different applications by modifying the TACF file.

Figure 1 shows the interface description from the TIDL file for a sample interface named **merchandise**. The first line contains a universal unique identifier (UUID) for this interface. The DCE RPC **uuidgen** utility can be used to generate a UUID, as well as an empty TIDL/IDL template file that can be used as the basis for your specialized TIDL file.

```
[
uuid(00083d4c-e722-18fd-9ed2-c037cf690000),
version(1.0)
]
interface merchandise
{
[transactional] void merchandise_QueryItem(
                        [in] long stockNum,
                        [out] long *amountP);
[transactional] void merchandise_OrderItem(
                        [in] long stockNum,
                        [in] long amount);
}
```

Figure 1. merchandise.tidl

Following the UUID is the version number of the interface. The version number is used in some of the internal data structure names produced by TIDL and IDL, uniquely identifying the client and server stubs produced from a certain version of the TIDL input file. The form of the version number is

major-version-number.minor-version-number. If a single integer is specified as the version number, TIDL interprets it as the major number and supplies a minor number of `0` (zero) automatically.

Note: TRPC does not support distinct, coexisting versions of the same interface.

The name of the interface (in this case, **merchandise**) and the actual interface definition (that is, the functions that actually make up the interface) follow the UUID and version number information. Only functions requiring that information be exchanged by using the RPC mechanism must be described in the TIDL file. Of the functions described in the TIDL file, those that are executed within the scope of a transaction are prefixed by the **transactional** attribute. (See "Differences between TIDL and IDL" on page 17 for more information on attributes.)

A TACF file can be used to control the way binding occurs and to control the way errors and exceptions are reported. TACF files name the target interface and the operations in the interface that use modified binding and error handling. The TACF file associated with the **merchandise** TIDL file is shown in Figure 2. Explicit binding is used in the **merchandise** example, so the type of binding does not need to be specified in the TACF file. No other operations are specified in the TACF file, meaning that error handling is not modified for any of the operations defined in the corresponding TIDL file.

```
interface merchandise
{
}
```

Figure 2. merchandise.tacf

The TACF files used by **tidl** have the same syntax as the attribute configuration files (ACF) used with IDL files for DCE RPC. For more information about the syntax of ACF files, see the *OSF DCE Application Development Guide* and *OSF DCE Application Development Reference* documents.

TIDL file format

This section describes details of the format of TIDL interface definition files. TIDL uses the same interface definition language and base data types as the DCE IDL, with some minor differences and restrictions. For a complete description of DCE RPC and the syntax and format of IDL interface description files, see the DCE RPC information in the *OSF DCE Application Development Guide* and *OSF DCE Application Development Reference*.

IDL data types

Table 2 lists the IDL base data type specifiers and the corresponding C-language data types. Note that the IDL compiler does not permit the specification of type **int** in an RPC definition. The supported integer types in IDL are **small**, **short**, **long**, and **hyper**. You can use the **idl_** type macros emitted by the IDL compiler in your application code, which ensures that your type declarations are consistent with those in the stubs.

Table 2. IDL base data type specifiers

Sign	Specifier	Type	Size	IDL type macro
	small	**int**	8 bits	**idl_small_int**
	short	**int**	16 bits	**idl_short_int**
	long	**int**	32 bits	**idl_long_int**
	hyper	**int**	64 bits	**idl_hyper_int**
unsigned	**small**	**int**	8 bits	**idl_usmall_int**
unsigned	**short**	**int**	16 bits	**idl_ushort_int**
unsigned	**long**	**int**	32 bits	**idl_ulong_int**
unsigned	**hyper**	**int**	64 bits	**idl_uhyper_int**
		float	32 bits	**idl_short_float**
		double	64 bits	**idl_long_float**
		char	8 bits	**idl_char**
		boolean	8 bits	**idl_boolean**
		byte	8 bits	**idl_byte**
		void		**idl_void_p_t**
		handle_t		

The IDL compiler supports two classes of pointers: reference pointers and full pointers. Reference pointers (indicated with the attribute **ref**) are the less complex form of pointer. It is chiefly used for passing a parameter by reference. Reference pointers always point to valid storage, but cannot point to a storage area that is pointed to by any other pointer. Full pointers (indicated with the attribute **ptr**) are more complex. They support all capabilities associated with pointers, such as linked lists, trees, and queues. Unlike reference pointers, full pointers can change their values during a call.

The IDL compiler does not permit ambiguity in pointer declarations. You can either apply the appropriate attribute to the declaration of the parameter, or you can use the **pointer_default** attribute in the interface heading. For example, the following two declarations make **anint** a full pointer:

```
void op ([in, ptr] long *anint);
```

or :

```
[uuid(20ffcf02-d920-11d1-9c98-a9460b11aa77),
version 1.0,
pointer_default(ptr)
]interface full_pointer
{
typedef long *long_ptr;
void op ([in] long_ptr anint);
}
```

Constructed data types are built on the basic data types. The IDL compiler provides the following constructed data types:

- Structures
- Unions
- Enumerations
- Pipes
- Arrays
- Strings

As in C, the arrays and strings are specified by using declarator constructs that specify the bounds of the array or string. The general form of the bounds declarator is [*lower..upper*]. A fixed length array of 12 integers can be declared as in this example:`long a1[0..11]`. The following declaration specifies a string of 80 characters:`[string] char abc[81]`.

The other constructed types are specified by using type specifiers. For more information about IDL data types and pointers, see the *OSF DCE Application Development Guide*.

Differences between TIDL and IDL

TIDL and IDL, although similar, are not exactly alike. The following topics describe the elements of TIDL that differ from IDL to support Encina's transactional semantics.

Transactional attributes

An extra attribute can be included in an operation attribute list to specify whether the operation is transactional or nontransactional. To retain nontransactional semantics, the **nontransactional** attribute must be specified. To provide transactional semantics, the **transactional** attribute can be specified (as shown in Figure 1 on page 14). By default, an operation is transactional if neither attribute is specified.

The **transactional** and **nontransactional** attributes can also be used in an interface attribute list. Unless a different attribute is specified for an operation in the interface, the interface attribute is used for that operation by default.

Error status and DCE RPC exceptions

Operations in TIDL files can be defined to return an error status rather than to use exception handling. If an operation defines an output parameter or a return value of the **error_status_t** type, the TIDL-generated stubs catch any exceptions raised by the DCE RPC runtime and pass them to the application as status codes.

Note: Using type **trpc_status_t** in TIDL files as the type for returning the status of errors is supported for backwards compatibility only—use the **error_status_t** type instead.

For an exception that occurs during a nontransactional operation, TRPC checks whether any status parameters have been defined for the operation before it raises the exception. If a nontransactional operation does not return an error status and no **fault_status** or **comm_status** attribute is specified for it in the TACF, then the exception is reraised in the client.

If an exception occurs during a transactional operation, TRPC aborts the transaction automatically. TRPC always translates a transaction abort at the server into an exception. In applications using Tran-C, exceptions are caught and control transfers to the abort clause. In applications using other transactional interfaces (such as TX), exceptions are *not* caught automatically; the application must catch exceptions (or check the error status) explicitly.

Named exceptions

The TIDL compiler adds a set of named exceptions defined by Encina to the generated IDL and ACF files automatically. Any named exceptions defined in the TIDL file are appended to the Encina system exceptions.

This feature must be disabled if your IDL compiler does not support named exceptions; versions of the compiler prior to DCE 1.0.3 do not provide this support. Include the **-noExceptions** option on the **tidl** command line to prevent the inclusion of named exceptions.

DCE-only RPC interfaces

TIDL allows an interface attribute to be defined for a DCE-only RPC interface UUID. The **dceOnlyRpcUuid** attribute can be specified in the interface attribute list of the TIDL file to supply an interface UUID for a DCE-only RPC interface; the supplied UUID is used in the DCE-only RPC IDL file generated by TIDL. See “Additional files for DCE clients” on page 22 for information on the files generated for DCE-only RPCs.

The following is an example interface attribute list that defines the **dceOnlyRpcUuid** attribute:

```
[uuid(003549e0-a1f5-1f1a-ba3e-9e620912aa77),
      dceOnlyRpcUuid(00224c0a-a1f7-1f1a-a4c1-9e620912aa77),
      version (1)]
```

See the reference page for the **tidl** command for more information on supplying an interface UUID for a DCE-only RPC interface.

Customized handles

When customized handles are used, binding routines that return transactional handles (type **trpc_handle_t**) in place of the regular RPC handles (type **handle_t**) must be provided. Such a binding routine has the following signature:

```
trpc_handle_t <custom_type>_tranBind(
     <custom_type> custom_handle,
     trpc_tranInfo_t *info,
     trpc_ifSpec_t *spec)
```

Unbinding routines that take transactional handles (type **trpc_handle_t**) in place of the regular RPC handles (type **handle_t**) must also be provided. These unbinding routines have the following signature:

```
void <custom_type>_tranUnBind(
     <custom_type> custom_handle,
     trpc_handle_t handle,
     trpc_tranInfo_t *info,
     trpc_ifSpec_t *spec)
```

Refer to the DCE IDL documentation for further information on using the binding and unbinding functions.

Limitations of TIDL

TIDL imposes the following restrictions in the way IDL and DCE RPC can be used to build distributed applications:

- The native manager functions must have the same names as the corresponding operations in the interface definition file.
- The same interface cannot be multiply registered with different type UUIDs. TIDL allows only one set of user-provided manager functions to be invoked by the shadow manager functions produced by TIDL.
- Declarations of the following format for operation parameters and return values *cannot* be used:

  ```
  struct {
      int a;
      int b;
  ```

```
} * jill_read(
[in] trpc_handle_t handle,
[in] long int index
);
```

Instead, you must use suitable type definitions, as shown in the following example:

```
typedef struct {
    long int a;
    long int b;
    } struct_a_b_t;
    struct_a_b_t * jill_read(
    [in] trpc_handle_t handle,
    [in] long int index
    );
```

The type definition must be inside the brace following the **interface** keyword.

Files produced by the TIDL compiler

Using an example, this section describes each of the output files produced by the TIDL compiler.

Note: The names given for the example output files reflect the default names generated by TIDL on UNIX-type file systems. If you are using a file system that restricts filenames to a particular format, the generated filenames are different. See "TIDL input and output files" on page 30 for the exact form of filenames generated on different file systems.

Default files

TIDL generates several files by default when it compiles a **.tidl** file. The default files produced by TIDL include an IDL file, a header file, and three stub files. You can control which of these default files are produced by specifying the *-stub* argument on the **tidl** command line.

For example, suppose TIDL is presented an input file called **jill.tidl** that defines an interface called **jill** and an operation called **jill_read**. TIDL produces five output files as shown in Figure 3 on page 21.

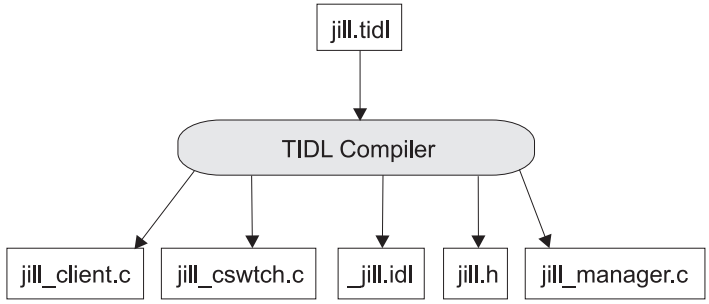

Figure 3. Default files generated by TIDL

The following is a description of each of the files that TIDL generates by default:

jill_client.c
This file contains the shadow client stubs that TIDL produces. It must be compiled and linked with client programs. It contains code that calls the TRPC runtime and invokes the modified operations.

jill_cswtch.c
This file must normally be compiled and linked with the client programs. However, If an application that uses TIDL is to be used as both a client and a server of an interface, it must not link with this file. Such an application must invoke the RPC through the entry-point vector initialized in **jill_client.c**.

_jill.idl
The name of the output interface definition file is **_jill.idl**. TIDL also prefixes an underscore (_) to each operation name in the interface definition file. Each operation in this file also has some additional parameters for transmitting and receiving transaction service data and callback data.

jill.h
This file is described in detail in "Details of TIDL-generated header files" on page 24. The client and server programs must include this file instead of the header file produced by IDL. The **jill.h** file includes the header file produced by IDL (**_jill.h**, in our example).

jill_manager.c
This file contains the shadow manager stubs that TIDL produces. It must be compiled and linked with the server. It contains code that calls TRAN, invokes the callbacks, and calls the user-provided manager function.

If IDL is then presented the input file called **_jill.idl**, which was generated by the TIDL compiler, IDL produces three output files, as shown in Figure 4.

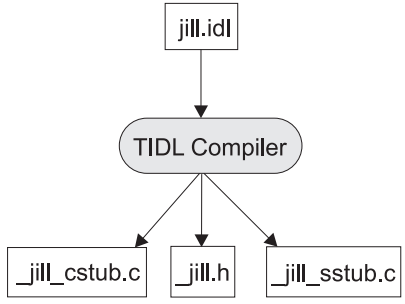

Figure 4. Default files generated by IDL

IDL generates the following files by default:

_jill_cstub.c
This file contains the client stubs that marshal the input parameters and unmarshal the output parameters. It is compiled and linked with the client programs.

_jill.h
This file is expected to be included in the client and the server programs when DCE RPC is used. The header file produced by TIDL (**jill.h**, in our example) includes this file automatically.

_jill_sstub.c
This file contains the server-side stubs that unmarshal the input parameters and marshal the output parameters. It is compiled and linked with the server programs.

Additional files for DCE clients

TIDL can generate several optional files that are used in creating pure DCE client programs that can issue RPCs to Encina interfaces exported by Encina Monitor application servers. These files are generated automatically when the **-dceOnlyRpc** option is specified on the **tidl** command line. The DCE-only RPC files produced by TIDL include an IDL file, a stub file, a manager entry-point vector (EPV) file, and an attribute configuration file (ACF). See the *Encina Monitor Programming Guide* for more information on using pure DCE clients with the Encina Monitor.

For example, suppose TIDL is used to create a DCE-only RPC interface from the input file called **jill.tidl**. Using the **-dceOnlyRpc** option produces the following output files in addition to the default files described in "Default files" on page 20:

jill_dceOnlyRpc.idl
This file contains only the nontransactional operations defined in the TIDL file. The **.idl** file must be run through the IDL compiler on the server to generate the DCE RPC server stubs. The **-no_mepv** option must be specified on the **idl** command line (when compiling both **jill_dceOnlyRpc.idl** and **_jill.idl**) to ensure that the IDL compiler does not generate the manager EPV for this interface automatically; the manager EPV is contained in the **jill_dceOnlyRpc_mepv.c** file produced by TIDL. To create a DCE-only RPC client, the **.idl** file must be run through the IDL compiler on the client machine, and the stub files generated must be linked with the client application code. IDL accepts the **jill_dceOnlyRpc.idl** file as input and produces the following set of files:

- **jill_dceOnlyRpc.h**–This file contains declarations for the DCE-only RPC interface and the manager EPV, as well as declarations for routines that return the DCE-only RPC interface handle and the manager EPV. These routines enable the application writer to obtain the interface handle and manager EPV required for interface registration at application server startup; neither the **jill.h** nor **jill_dceOnlyRpc.h** header file can be included in the application source file due to declaration conflicts.
- **jill_dceOnlyRpc_cstub.c**–This file contains the client stubs that marshal the input parameters and unmarshal the output parameters. It is compiled and linked with the client programs.
- **jill_dceOnlyRpc_sstub.c**–This file contains the server stubs that unmarshal the input parameters and marshal the output parameters. It is compiled and linked with the server programs.

jill_dceOnlyRpc_manager.c
This file contains all the DCE-only RPC shadow manager routines that correspond to the operations in the IDL file **jill_dceOnlyRpc.idl**. These shadow manager routines are implemented as calls to the corresponding shadow manager routines produced by TIDL (for example, in the file **jill_manager.c**).

jill_dceOnlyRpc_mepv.c
This file contains the manager EPV for the DCE-only RPC interface. The EPV is stored in a separate file to avoid conflicts between the DCE-only RPC interface and the TRPC interface definitions. The file can be compiled and linked with server programs.

jill_dceOnlyRpc.acf
This file is a placeholder that can be used for specifying operation attributes for the DCE-only RPC interface. By default, the **explicit_handle**

attribute is set. This attribute can be changed to match the application's mechanism for obtaining a binding handle to the server. See the *OSF DCE Application Development Guide* for more information on specifying operation attributes in an ACF.

Details of TIDL-generated header files

This section uses the example of the **jill.tidl** file containing the **jill_read** operation to describe the contents of the header file produced by TIDL. The example header file **jill.h** contains the following:

- The **_jill.h** header file produced by IDL. This file is needed by application programs to get the type definitions of the operation parameters and to return values.
- The data type **jill_v0_0_epv_t**. This type is used to declare the client stub entry-point vector initialized by the shadow client stub file. The name of this data type is consistent with the names of the corresponding data types produced by IDL. Note that the name contains the major and minor numbers of the interface version. Using these numbers helps to ensure that the client and server stubs were produced at the same time, from the same TIDL file, and therefore have consistent call and return syntax. When the interface is changed, the version numbers should also be changed. Changing the version numbers helps to ensure that the stubs produced from the new interface description file do not attempt to use the entry point vectors associated with the old stubs. The old stubs continue to support the interface as it was defined when the stubs were generated, through the old entry point vector.

  If the interface uses explicit RPC handles, the client stub entry-point vector is defined as follows:

  ```
  typedef struct jill_v0_0_epv_t {
  void (*jill_read)(
      #ifdef __STDC_
      /* IN */  trpc_handle_t handle,
      /* IN */  unsigned long int index,
      /* OUT */ unsigned long int *value
      #endif
      );
  } jill_v0_0_epv_t;
  ```

- The **extern** declaration of the shadow client stub entry point vector. This vector is initialized in the shadow client stub file (in this case, **jill_client.c**). The file **jill_cswtch.c** includes **jill.h** and uses this entry point vector to invoke the shadow client stubs in **jill_client.c**. If certain applications need to be both clients and servers of the same interface, they need to invoke the remote procedure through the shadow client entry point vector and therefore also need this declaration.

  ```
  extern jill_v0_0_epv_t jill_v0_0_client_epv;
  ```

The naming strategy for the client stub entry point vector is consistent with IDL.

- The **extern** declaration of the manager entry point vector. This vector is initialized in the shadow manager stub file (in this case, **jill_manager.c**). This declaration is needed to register the manager entry point vector with the DCE RPC function **rpc_register_if**. For further information, refer to the *OSF DCE Application Development Guide*.

  ```
  extern _jill_v0_0_epv_t jill_v0_0_manager_epv;
  ```

- The `extern` declaration of a surrogate manager entry point vector. This vector is also initialized in the shadow manager stub file (in this case, **jill_manager.c**). This declaration can be used for the same reasons as the previous declaration; the difference between the two is that this declaration is compatible with the type expected by the **rpc_register_if** function and therefore does not need to be cast. The value of `jill_v0_0_mgr_epv` is initialized to be the address of `jill_v0_0_manager_epv` in **jill_manager.c**.

  ```
  extern rpc_mgr_proc_t *jill_v0_0_mgr_epv;
  ```

- The **extern** declaration of the relevant implicit handle variables if the interface uses implicit handles.
- The following preprocessor definitions:

  ```
  #define jill_v0_0_c_if_spec _jill_v0_0_c_ifspec
  #define jill_v0_0_s_if_spec _jill_v0_0_s_ifspec
  ```

  These enable the application developer to use the interface specification variables whose names comply with the naming rules used by IDL.

- The **extern** declarations of the DCE-only RPC interface and manager entry-point vector if nontransactional DCE clients are used. The application server must register the DCE-only RPC interface along with the TRPC interface when the server is started. The EPV is initialized in the manager EPV file (in this case, **jill_dceOnlyRpc_mepv.c**). See "Additional files for DCE clients" on page 22 for more information.

  ```
  extern rpc_if_handle_t jill__d_v0_0_s_ifspecepv;
  extern rpc_mgr_epv_t jill__d_v0_0_mgr_epv;
  ```

The header file must be included in both the client and server programs.

Building clients and servers

This section illustrates the process of using the output files produced by TIDL to build client and server applications. TIDL produces files that enable you to build Encina clients and servers or to build DCE client and Encina servers.

For the examples in this section, the name of the TIDL interface definition file is **jill.tidl**; the client program that initiates the RPC is called **client.c**, the server program is called **server.c**, and the file that contains the sources

implementing the manager functions is called **manager.c**. The TRPC functions are contained in the library called **Encina**.

Note: The example filenames referred to throughout this section reflect the default names used on UNIX-type file systems. Substitute the actual names of your files for the example names when executing the **tidl** command. See "TIDL input and output files" on page 30 for information on the filenames used by TIDL on various file systems.

Building Encina clients and servers

The following example illustrates the process of building client and server applications by using the default output files produced by TIDL. The process is illustrated Figure 5 on page 27.

1. The application developer runs **tidl** and passes the **jill.tidl** file as a command-line argument. This process produces the following files:

   jill_client.c

   jill_manager.c

   jill_cswtch.c

   jill.h

   _jill.idl

2. The developer executes **idl** and passes it the IDL file **_jill.idl**. This produces the following files:

   _jill_cstub.c

   _jill_sstub.c

   _jill.h

3. The developer builds the client by compiling and linking the following files:

   client.c

   jill_client.c

   jill_cswtch.c

   _jill_cstub.c

   Encina library

4. The developer builds the server by compiling and linking the following files:

   server.c

   manager.c

   jill_manager.c

   _jill_sstub.c

   Encina library

When the client initiates the **jill_read** RPC, control is transferred to the **jill_read** function in the file **jill_cswtch.c**. This function calls the appropriate shadow client stub in **jill_client.c** by using the shadow client stub entry-point vector initialized in **jill_client.c**. The shadow client stub for **jill_read** then uses the client stub entry-point vector initialized in the **_jill_cstub.c** file to call the client stub produced by IDL and invoke the actual RPC.

On the server side, the RPC runtime calls the native server stub produced by IDL. This server stub calls the shadow manager stub by using the manager entry-point vector registered with the RPC runtime. The shadow manager stub in **jill_manager.c** then calls the user-provided manager function. TIDL includes the declaration of the operations described in the interface definition file in the shadow manager file it produces.

When writing a server, you must not create a new manager entry-point vector. Instead, you must use the manager entry-point vector initialized by the shadow manager stub to register RPC interfaces. A declaration of the manager entry-point vector is available in the header file produced by TIDL, as described in "Details of TIDL-generated header files" on page 24. Refer to the *OSF DCE Application Development Guide* for further information on registering interfaces.

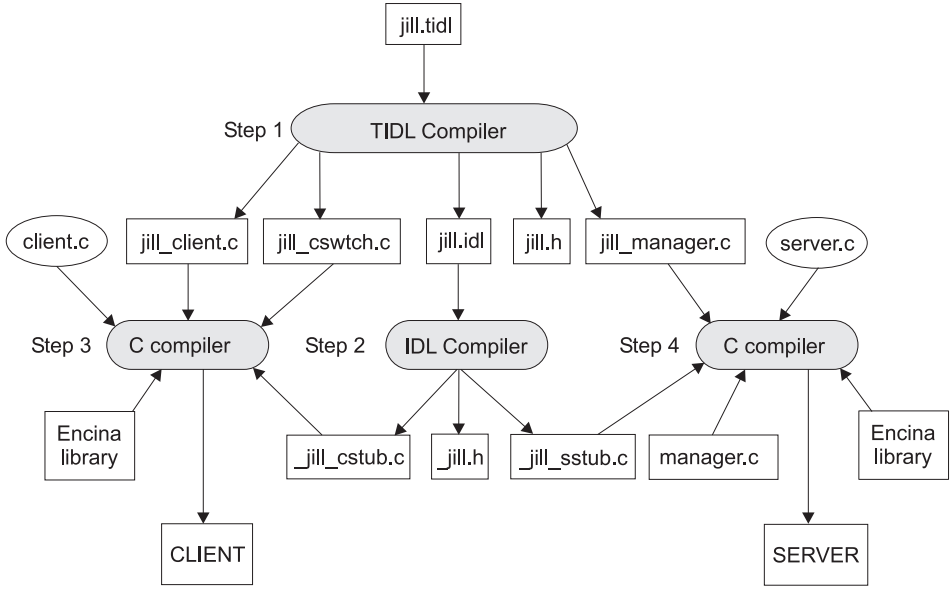

Figure 5. Building Encina clients and servers

Note that TIDL produces two source code files for the client program (**jill_client.c** and **jill_cswtch.c**), but only one is produced for the server

program (**jill_manager.c**). This behavior occurs because some programs must act as both clients and servers, and interface calls for each of these instances must be isolated. One reason that a server uses its own interface is for easy replication.

To act as both a client and server of an interface, a server calls the client stub code to request operations, and it also calls the manager stub code to execute those operations. However, the manager and client stub code contain routines with the same names (to allow the easy flow of information between them). To avoid problems with multiply defined names when attempting to link these modules together, the client switch source code file (**jill_cswtch.c**) contains *public* routines that can be called by the client program directly. The client stub source code file (**_jill_cstub.c**) contains private routines that are called only by the routines in the client switch code. Therefore, the server is compiled with the server stub and client stub routines but not with the client switch routines.

If the application is both the client and server of the interface **jill**, it is built by compiling and linking the following files:

server.c

client.c

jill_client.c

_jill_cstub.c

jill_manager.c

_jill_sstub.c

manager.c

Encina library

Building DCE clients and Encina servers

The following example illustrates the process of building a pure DCE client and an Encina server by using the additional output files produced by TIDL when the **-dceOnlyRpc** option is specified on the command line. In this case, **jill.tidl** defines nontransactional interfaces that the DCE client can use to issue RPCs to the Encina server.

1. The developer runs **tidl** and passes the **jill.tidl** file as a command-line argument. In addition to the default files produced (see Step 1 in Figure 5 on page 27), this process produces the following files:

   jill_dceOnlyRpc_manager.c

   jill_dceOnlyRpc_mepv.c

   jill_dceOnlyRpc.idl

   jill_dceOnlyRpc.acf

2. The developer executes **idl** on each of the IDL files generated by TIDL: **_jill.idl** and **jill_dceOnlyRpc.idl**. The **-no_mepv** option must be specified

on the **idl** command line for both files; this specification prevents IDL from generating manager EPVs (already generated by TIDL in the file **jill_dceOnlyRpc_mepv.c**) for the **jill** interface. In addition to the default files produced (see Step 2 in Figure 5 on page 27), IDL produces the following files:

jill_dceOnlyRpc_cstub.c

jill_dceOnlyRpc_sstub.c

jill_dceOnlyRpc.h

3. The developer builds the server by compiling and linking the following files:

   server.c

   manager.c

   jill_manager.c

   _jill_sstub.c

   jill_dceOnlyRpc_manager.c

   jill_dceOnlyRpc_mepv.c

   jill_dceOnlyRpc_sstub.c

   Encina library

4. The developer copies the files **jill_dceOnlyRpc.idl** and **jill_dceOnlyRpc.acf** to the client machine and executes **idl**, using the **-server none** option, passing **idl** the IDL file **jill_dceOnlyRpc.idl** as an argument. This produces the following files:

   jill_dceOnlyRpc_cstub.c

   jill_dceOnlyRpc.h

5. The developer builds the client by compiling and linking the following files:

   jill_dceOnlyRpc_cstub.c

   client.c

When it is started, the application server must register the DCE-only RPC interface along with the TRPC interface. Declarations for the DCE-only RPC interface specification and manager EPV are contained in the header file generated by TIDL. The interface name has the format *interface_name*_**d_v1_0_s_ifspec**, and the manager EPV has the format *interface_name*_**d_v1_0_mgr_epv**.

An Encina Toolkit server can use the **rpc_server_register_if** DCE function to register the interface. The following is an example of registering the **jill** interface:

```
rpc_server_register_if(jill_d_v1_0_s_ifspec,
                       (uuid_t *)NULL,
                       jill_d_v1_0_mgr_epv,
                       &status);
```

Refer to the *Encina Monitor Programming Guide* for information on registering an interface for an Encina Monitor application server.

TIDL input and output files

The names of the input and output files used by TIDL depend on the file system. Some file systems restrict the length and format of filenames.

The tables in this section use an example TIDL file called **interface** to illustrate how the same files are represented on different file systems. On FAT (DOS) file systems, note that the base filename **interface** is truncated due to the length restriction on filenames. In addition, extensions are truncated to three characters where necessary.

Input files (Table 3) are passed as arguments to the **tidl** command. The TIDL interface definition file is required. The transactional attribute configuration file (TACF) is optional.

Table 3. Input filenames for TIDL on different file systems

UNIX (default)	FAT	Description
interface.tidl	**interf.tid**	Transactional Interface Definition File
interface.tacf	**interf.tac**	Transactional Attribute Configuration File

Output files (Table 4) are generated by TIDL. The files that are generated are determined by the switches used with the **tidl** command.

Table 4. Output filenames generated by TIDL on different file systems

UNIX (default)	FAT	Description
interface_manager.c	**interf_m.c**	Shadow Manager Stubs
interface_client.c	**interf_c.c**	Shadow Client Stubs
interface_cswtch.c	**interf_w.c**	Client Switch File
interface.h	**interf.h**	Interface Header File

Table 4. Output filenames generated by TIDL on different file systems (continued)

UNIX (default)	FAT	Description
_interface.idl	**_interf.idl**	Interface Definition File
_interface.acf	**_interf.acf**	Attribute Configuration File
interface_dceOnlyRpc.idl	**inter_d.idl**	DCE-only RPC Interface Definition File
interface_dceOnlyRpc_manager.c	**inter_dm.c**	DCE-only RPC Shadow Manager Stubs
interface_dceOnlyRpc_mepv.c	**inter_dv.c**	DCE-only RPC Manager EPV File
interface_dceOnlyRpc.acf	**inter_d.acf**	DCE-only RPC Attribute Configuration File

IDL generates output files from the interface definition file produced by TIDL. The files that are generated are determined by the IDL compiler, the platform, and the switches used with the **idl** command. See the documentation for your IDL compiler for more information.

Part 2. Transactional-C

Chapter 3. Transactional-C concepts

Transactional-C (Tran-C) provides extensions to the C programming language that provide mechanisms for easily invoking and using the functionality provided by the Encina Toolkit. The Tran-C runtime system invokes the necessary Toolkit functions to support the Tran-C functions used in a program and automatically monitors the scope and state of transactions and their associated low-level data structures and constructs.

This chapter compares using Tran-C to develop transactional applications with developing those same applications by using lower-level functions such as those provided in the Encina Toolkit interfaces. It also explains the model of computation and program development used in Tran-C, and it discusses aspects of program development that are unique to Tran-C.

Introduction to Transactional-C

Encina's Transactional-C programming language is an interface to the Encina Toolkit that simplifies the development of transactional applications by greatly reducing the number of necessary calls to the underlying Encina Toolkit. Tran-C consists of the most commonly used macros and library functions from the Encina toolkit, eliminating the need for direct access to the Toolkit module interfaces in most cases. Tran-C does not incorporate seldom-used routines from other Toolkit modules (for example, Transaction Service routines that return information about the various relationships of transactions), and Tran-C does not incorporate any Recovery Service routines. Monitor application servers can be made recoverable with a single function call.

Transactional applications are programs that can create, commit, and abort transactions. They can also issue transactional remote procedure calls (TRPCs) to other applications. The basic properties of transactional applications are discussed in "Distributed transactions" on page 4 and "Outline of an Encina transactional application" on page 10. Transactional applications generally belong to one of three categories: servers, clients of servers, or standalone applications. A server is an application that accepts RPCs from other applications (the clients of that server). When a server application receives a request from a client, the server inherits the transaction attribute from the calling thread in the client program. A client is an application that typically requests services or information from another application. A standalone application is an application that does not depend on information or services provided by any other application.

To maximize throughput in transactional applications that typically handle large volumes of client/server communications, Tran-C provides threads. *Threads* are multiple, simultaneous paths of execution running within a single C application. Threads enable separate procedures and functions within an application to be executed concurrently, sharing access to the same data. For example, threaded server applications typically have multiple threads so that multiple incoming TRPCs can be processed concurrently. Similarly, threaded client applications can simultaneously query or update multiple resources.

To integrate threads into the transactional model of application development, threads carry the identifier of the transaction under which they are running as a runtime attribute. If a thread is not currently running under a transaction, it is referred to as being outside the scope of a transaction. The scope of the transaction identifier attribute is dynamic; it persists across function calls. Control flow follows the status of the transaction attribute, meaning that if a transaction aborts, program execution resumes after the scope of the transaction.

Tran-C also provides higher-level interfaces to other Toolkit modules, such as the Toolkit's Lock Service (LOCK). To guarantee the consistency of any recoverable data managed by an application, the application must explicitly secure an appropriate lock on that data before accessing it. LOCK provides a logical locking facility that enables the programmer to associate lock names with recoverable data, and then to obtain locks on those lock names. The Tran-C transactional locks support multiple access modes, expediting concurrency between threads that require nonconflicting access to available data. To guarantee that transactions are serialized, locks are generally held until the end of the transaction.

The Tran-C model of computation

Throughout this document, specific terms are used to refer to the different types of expressions provided by Tran-C. *Constructs* are groups of associated phrases that together make up an entity that affects the flow of control in a program. Examples of constructs are the Tran-C **transaction** and **cofor** constructs. Constructs consist of multiple alternate sections to which control can pass—these sections of a construct are referred to as *clauses*. Single expressions, functions, or expressions executed inline with program code are referred to as *statements*.

Tran-C is divided into two groups of services and support, the *Executive* and the *Server Extensions*. The Tran-C Executive provides Tran-C support for all of the Encina Toolkit services used when writing client applications. The Tran-C Server Extensions provide Tran-C support for some of the Toolkit modules and services required by server applications. In this context, the primary difference between the client and server configurations of Tran-C is the

support provided in the Tran-C Server Extensions Encina Toolkit Lock Service. Locking guarantees that transactions executed at a server are serialized correctly, fulfilling the requirements of a transaction processing system.

The Tran-C Executive supports *ephemeral* applications, which are applications that do not directly support recoverable data. *Recoverable* applications use logging and recovery services to support recoverable data that survives across system problems or failures. Support for some services of these types is required to guarantee the *permanence* of a transaction. Applications written using only the Executive can achieve permanence by communicating with other applications that are themselves recoverable (for example, by contacting a recoverable server), or by integrating external (non-Encina) logging and recovery services into the Tran-C application development environment. If your application uses the Encina Monitor, it handles recovery services for your application.

The most important of the constructs provided by Tran-C is the **transaction** construct. This construct defines a scope within a program, bounding the normal execution of any number of computations. This scoping is dynamic, which means the construct affects all computation from the time execution enters the construct until execution exits the construct. These constructs obey a stack-like discipline following their nested execution.

All functions called within the scope of a single transaction share the same transaction attribute. When the end of the scope bounded by the transaction construct is reached, Transactional-C commits the transaction automatically. If the transaction is aborted during the execution of the computations bounded by the **transaction** construct, Tran-C automatically transfers control to the end of that scope. Tran-C provides special **onAbort** and **onCommit** clauses that enable the developer to associate program statements with the exit status of the transaction (in other words, whether it committed or aborted). These clauses are associated with a specific transaction, and are executed when the relevant event (commit or abort) occurs.

Tran-C also supports *nested transactions*, which are created by declaring a new transaction within the scope of another transaction. Nested transactions provide successively smaller units that make it easier to isolate potential failures in the code. For example, a transaction that requires a remote resource but fails to access it. This failure destroys the entire transaction and all computation carried out on its behalf. If the resource is replicated, then accessing it from within a nested transaction causes only the subtransaction to abort, protecting the parent transaction from the failure and allowing it to try a replica in place of the original resource.

Rather than rely on heavyweight processes, Tran-C provides C extensions that simplify creating and managing lightweight threads of control. Using multiple

lightweight threads within a process is different from using multiple processes within an application. Multiple threads within a process share the address space of that process. Threads can be created to run in the following ways:

- Within the scope of the transaction that is current at the time that the thread is created
- Within the scope of a newly created nested transaction
- Independently, outside the scope of the thread that created them

The Tran-C constructs for supporting nested transactions and multiple threads of control are described in detail in "Nested and top-level transactions" on page 53 and "Creating concurrent transactions or synchronous threads" on page 54.

Considerations for developers

This section introduces some important considerations for C programmers who are new to developing threaded transactional applications.

Dynamic scoping of transactions

Each time Tran-C encounters a Tran-C **transaction** construct, it creates a transaction that executes the statements bounded by the transaction clause of that statement. Each transaction is uniquely identified by a transaction identifier, which is considered an attribute of that transaction. The scope of this attribute is dynamic; it follows the flow of control within a program as control is passed to other functions and procedures and then returned. This attribute persists across all functions and procedures called within the code delimited by the transaction construct, rather than simply being defined by the inline statements between the beginning and end of the current transaction construct in the program's source code.

Transfer of control in transactions

The Tran-C **transaction** construct supports associated **onCommit** and **onAbort** clauses for each instance of the transaction construct. When a transaction bounded by the **transaction** construct aborts, control in the program transfers automatically to the associated **onAbort** clause. No subsequent statements within the scope of the transaction construct are executed. Control passes to the beginning of the associated **onAbort** statement immediately. This transfer of control has a number of consequences. The following suggestions can help prevent problems related to aborted transactions:

- Use the specification `volatile` for a local variable that is modified within the body of a transaction and then used outside it within the same lexical scope (including within the **onAbort** or **onCommit** clause). This term is an ANSI-C type specifier that guarantees the consistency of automatic variables in the case of modifications not under control of the C compiler.

Figure 6 shows an example. Without the `volatile` qualification of the `int` type for the variable *x*, an application in ANSI C does not guarantee that the value of *x* is 42.

- Consider that transfer of control can mean that cleanup routines occurring in the code bounded by the transaction construct are not executed, resulting in the creeping consumption of system resources (such as memory) by long-running applications. If your application requires allocated memory, use the transactional memory allocation functions provided by Tran-C and explained in "Transactional memory allocation" on page 84. A way to guarantee that other allocated resources are returned correctly to the system when a transaction aborts is to keep a list of all of the system resources used within a transaction. If the transaction aborts, the **onAbort** clause associated with that transaction can use that list to deallocate and return those resources.
- Use transactional locks and mutexes whenever possible, instead of mutexes supplied by the underlying thread package. Transactional locks are monitored by the Tran-C runtime system and dropped automatically when a transaction commits or aborts. Mutexes allocated by using the Tran-C mutex functions (explained in "Transactional mutexes" on page 85) are also dropped when a transaction commits or aborts. Transactional locks can also be advantageous because they enable other threads to access data simultaneously (unless conflicting modes exist), while mutexes are exclusive to a single thread. Transactional locks are discussed in more detail in "Locking data in Tran-C applications" on page 89.
- When dumping output to some device or interacting with a user and displaying information, do not modify the data display until a transaction commits. Updates to displayed data can be placed inside the **onCommit** clause associated with the transaction construct that is responsible for updating the data.

```
volatile int x = 0;
transaction {
    x = 42;
    abort("Meaningless");
}
onAbort {
    printf("%d", x);
}
```

Figure 6. Qualifying the type of an automatic variable as volatile

Resource limitations

The following considerations are related to resource limitations:

- The necessary overhead of maintaining nested transactions limits the number of nested transactions that can exist within a transaction family, but the limit is several thousand. Programmers should reconsider problem-solution techniques in applications that require this many nested transactions.
- The TRPC library imposes limitations on the amount of Transaction Service state that can be transmitted along with client RPCs as out-of-band data. These limitations exist to increase performance for normal uses of TRPC, but they affect the possible complexity of transactions; factors contributing to the complexity of transactions include the number of participants in a transaction family, the nesting depth of subtransactions, the number of aborted transactions within a family, and so on. When the amount of Transaction Service out-of-band data exceeds the TRPC limit, a TRPC fails, and the transaction aborts. The Transactional-C abort reason in this case is `RPC Failure: during mgrFunction: DCE-rpc-0125: fault invalid bound (dce / rpc)`. User errors also produce this abort reason. For example, if an application's TRPC takes an array defined to be bounded by another parameter, and the argument supplied as the size exceeds the array argument, the same error occurs.

Thread-safe functions under UNIX

The biggest advantage in using threads in an application is the ability to share and exchange data between cooperating threads representing separate paths of execution in the application. However, sharing data between threads means that threaded applications must avoid using global or static variables whenever possible. These must be avoided because the location of these variables is the same for all threads sharing an address space, making it impossible to determine which thread last wrote the value.

Implementations of certain UNIX library functions use static locations to store intermediate or final values. Examples of these functions are the time functions, random number generation functions, math functions, and any functions that use these library functions (such as the **crypt** and **ctime** functions). Other UNIX functions, such as the **read** function and other input and output functions, block the parent process until they can complete. Blocking the parent process means that all of the threads running within that process are also blocked.

To eliminate potential problems caused by external functions that are not aware of threads, or that share the use of fixed memory locations, operating system vendors that support threading provide equivalent or replacement functions for these external functions. Functions that can be safely used in a threaded multiprocessing environment are known as *thread-safe* functions. Before using functions that use static memory locations or affect process execution, check the documentation for your system to determine if thread-safe versions of these functions are provided.

All UNIX functions that use static memory locations or structures can be used in the main part of a Tran-C application, as long as they are called only by a single thread, which guarantees that multiple threads do not overwrite the shared structure or data. If thread-safe versions of UNIX functions that affect or block execution of a process are not available on your system, contact your customer support representative for information on your alternatives.

If you experience problems when developing applications using Tran-C, and these problems appear to be related to internal data structure corruption, check the UNIX manual to determine whether the function in question references any global or static variables. If the function references static variables, consider contacting your operating system vendor to verify that the version of the function that you are using is actually thread safe.

For more information about compatibility issues between Tran-C, ANSI-C, and standard UNIX functions, see "Chapter 5. Advanced Tran-C programming" on page 81.

Chapter 4. Managing transactions using Tran-C

This chapter describes how to use Tran-C constructs to manage transactions in your application program.

Hello, World: An introductory standalone application

Figure 7 on page 44 shows a Tran-C implementation of the familiar C language "Hello, World" program. This example shows some of the fundamental constructs and declarations used in developing a Tran-C program. It provides the minimum set of statements necessary to write a Tran-C program that uses the **transaction** construct.

This sample program introduces the fundamental construct provided by Tran-C, the **transaction** construct. The **transaction** construct is a complex C language statement that consists of three clauses—the primary **transaction** clause, an optional **onCommit** clause, and a mandatory **onAbort** clause. When this construct is encountered the statements inside the **transaction** clause are executed by a transaction created by Tran-C . The **onCommit** clause enables developers to associate C statements with the successful commit of the associated statements from a **transaction** clause. This clause is optional because it is often unnecessary to take any special action in an application when a transaction completes successfully. The **onAbort** clause enables developers to specify actions that are taken if the associated transaction cannot be committed for some reason and therefore the transaction aborts. This clause is mandatory because aborted transactions usually require that the user be notified or require that some alternate action be taken. For more complete information about using the transaction construct, see "Beginning and ending transactions" on page 50.

```
#include <tc/tc.h>
inModule("helloworld");
int main()
{
  int i;
  inFunction("main");
  initTC();
  for(i=0;i<10;i++) {
    transaction {
        printf("Hello World - transaction %d\n", getTid());
        if (i % 2)
            abort("Odd Numbered Transactions are aborted...");
    }
    onCommit
        printf("\t(Transaction committed)\n");
    onAbort
        printf("Aborted in module : %s\n\t%s\n", abortModuleName(),
                            abortReason());
  }
  return 0;
}
```

Figure 7. The "Hello, World" program in Tran-C

The sample program shown in Figure 7 creates 10 transactions. When the value of the loop-control variable is an even number, the program prints the phrase ″Hello World″ and then indicates whether or not the transaction was successfully committed. When the value of the loop-control variable is an odd number, the program aborts the current transaction. This causes the flow of control to immediately transfer to the transaction construct's **onAbort** clause, demonstrating the transfer of control in a **transaction** construct when a transaction is aborted. The core of this program is the **printf** function, enclosed within the Tran-C **transaction** construct. Because the transaction construct is executed10 times, each iteration of the **printf** function is actually executed by a different transaction. The **printf** function is written to display the transaction identifier for the transaction that printed each iteration.

In a transaction processing environment, each transaction has a unique name known as a *transaction identifier* (TID). The Tran-C runtime system automatically manages passing, generating, and manipulating the TIDs in Tran-C programs. Ordinarily, these identifiers are transparent to the programmer because it is seldom necessary to know the identifier for any given transaction. However, in some circumstances—such as this first introduction to Tran-C—it is useful to actually see the identifiers allocated to different transactions to insure that they are indeed different. The Tran-C **getTid** function returns the transaction identifier associated with the calling transaction (see "Getting information about a transaction" on page 52 for details).

```
Hello World - transaction 65536
        (Transaction committed)
Hello World - transaction 1
Aborted in module : helloworld
        Odd Numbered Transactions are aborted...
Hello World - transaction 2
        (Transaction committed)
Hello World - transaction 3
Aborted in module : helloworld
        Odd Numbered Transactions are aborted...
Hello World - transaction 4
        (Transaction committed)
Hello World - transaction 5
Aborted in module : helloworld
        Odd Numbered Transactions are aborted...
Hello World - transaction 6
        (Transaction committed)
Hello World - transaction 7
Aborted in module : helloworld
        Odd Numbered Transactions are aborted...
Hello World - transaction 8
        (Transaction committed)
Hello World - transaction 9
Aborted in module : helloworld
        Odd Numbered Transactions are aborted...
```

Figure 8. Output from running the "Hello, World" program successfully

After this program is compiled as explained in "Chapter 6. Compiling Tran-C applications" on page 111, running the program produces the output shown in Figure 8.

This sample program illustrates the use of the Tran-C **transaction** construct in a very simple situation, and also illustrates some of the transactional heuristics provided by the modules of the Encina Toolkit used by Tran-C. The sample application did not register a recovery service, meaning that it does not use recoverable storage, and also made no RPCs, which means that no external services are involved. In general, before a transaction commits, all of the participants in the transaction must prepare, agreeing that they are able to commit the transaction. A record of any changes to recoverable data made as a result of the transaction must be generated by the recovery service and be logged by a log service before the transaction actually commits. In the example "Hello, World" program, since there are no other participants (for example, no recovery service, and therefore no log service), the application commits all transactions without waiting on any other participants.

Registering module and function names

A *module* is a collection of functions designed to perform a specific service. Modules are generally produced by compiling and linking multiple source files that contain the local functions and procedures used by that module.

All Tran-C applications must use the Tran-C **inModule** macro to register a module name in each source file that contains a **transaction** (or similar) statement, an **onAbort** clause, or an **abort** statement. Figure 7 on page 44 shows the **inModule** declaration. When a transaction is aborted, the name of the module that instigated the abort can be retrieved and displayed. This information can be useful when debugging transactional applications written using Tran-C.

To provide more detailed information about the source of an abort, Tran-C also provides the **inFunction** function. If you use this routine, it must be called from a function or procedure in a Tran-C source file before executing any other code in that procedure or function. Figure 11 on page 52 shows the call. Unlike the **inModule** macro, the **inFunction** declaration is optional. If **inFunction** is not called, the function name defaults to the string `TC_UNKNOWN_FUNCTION`. Figure 7 on page 44 shows the **inFunction** declaration in a program.

The syntaxes of the **inModule** and **inFunction** calls are identical:

```
inModule(char *moduleName);
inFunction(char *functionName);
```

The argument to each of these functions is a pointer to a null-terminated string of characters that uniquely identifies the module or function. The module and function names identify the source of a transaction's abort in any messages in an abort clause.

The information registered by using the **inModule** and **inFunction** declarations is retrieved by using the **currentModuleName** and **currentFunctionName** functions. Retrieving this information is discussed in "Determining where a transaction aborted" on page 75.

The module and function names registered by using the **inModule** macro and the **inFunction** function are stored in the Tran-C runtime environment. These strings must not be deallocated.

Initializing a Tran-C application

If your application is a Monitor client or a Monitor application server, the Monitor initializes Tran-C automatically. This section describes the functions used to initialize specific Toolkit services that can be used in a Tran-C application that does not use the Monitor.

Before any Tran-C statements can be executed, the Tran-C runtime system and the mechanism used to communicate between Tran-C client and server applications must be initialized. To provide a flexible initialization mechanism that makes it easy to integrate external modules with Tran-C applications, Tran-C provides several different ways of initializing the modules and services required for Tran-C applications. Using these initialization alternatives is discussed in the next few sections. The following is a quick summary of these alternatives:

- If you are using no external services (for example, the Recovery Service), call the **initTC** function to initialize only Tran-C.
- If your application requires external services that must be initialized between the time when the Encina Transaction Service is initialized, use the **preInitTC** and **postInitTC** functions, which are explained later in this section. For more information on initializing the Transaction Service, see the *Encina Toolkit Programming Guide*.
- If you are making RPCs, you must initialize the RPC mechanism before the final stage of Tran-C initialization. This does not mean you must initialize Tran-C in stages, as mentioned in the previous item, because you possibly do not require any Encina Transaction Service initialization to initialize the RPC mechanism. You can explicitly initialize only the callbacks required by TRPC by calling the **registerTRPCCallbacks** function. You can also initialize TRPC by calling the **tc_InitTRPC** function, followed by the **initTC** function. Other RPC mechanisms possibly require the flexibility of multiple stage Tran-C initialization. If you are using well-known endpoints, you must call the **trpc_InitWithTrdce** function.
- If you are using only Tran-C and TRPC, you can initialize these with one call, **initTCWithTRPC**.

These alternative initialization paths simplify integrating other software and services that use callbacks with Tran-C. Callbacks are executed in a reverse sequence of the order in which they are registered. Callbacks for specific applications must usually be registered in a specific order, to insure that Tran-C and the application each receive the appropriate callbacks. Registering and using callbacks is explained in "Registering and using callbacks" on page 81.

Using initTC to initialize Tran-C

The **initTC** function initializes only the Tran-C runtime system. Any initialization functions required by the modules used to communicate between

Tran-C client and server applications, such an RPC mechanism, must be separately initialized. The use of **initTC** is shown in Figure 7 on page 44. It has the following syntax:

```
void initTC();
```

The **initTC** function requires no arguments and returns no value. This function initializes the other Toolkit modules used by Tran-C and the Tran-C runtime system.

Two-Stage initialization of Tran-C

Trans-C provides application initialization and termination callbacks for initializing and terminating any external components used by a Tran-C application. To provide a flexible initialization mechanism that simplifies integrating these external components within a Tran-C application, Tran-C provides a two-stage initialization mechanism. The first step is to call the **preInitTC** function to initialize the low-level services used internally by Tran-C, and to initialize the Transaction Service. After this function has been called, the routines required to initialize external components, or register callbacks for them, can be called. For example, you can call the Tran-C function **tc_InitTRPC** to initialize TRPC or directly call the TRPC function **trpc_Init**. When all external initialization is complete, your application must call **postInitTC**, which executes any application callbacks that were registered and notifies the Transaction Service that the application's initialization is complete.

The **preInitTC** and **postInitTC** functions have the following syntax:

```
int preInitTC(void);
void postInitTC(void);
```

The **preInitTC** function returns `TRUE` (1) the first time it is called and `FALSE` (0) thereafter. Using this function enables you to isolate a block of code that you want to be performed only when an application is first initialized, for example:

```
if (preInitTC()) {
    statements
}
postInitTC();
```

If this initialization mechanism is used, the application must explicitly terminate any external services before calling the Tran-C **exitTC** function. Since these services are manually initialized within the application, they must be manually terminated before Tran-C applications are exited. Tran-C has no way of telling what services were initialized or what their termination and cleanup requirements are.

Initializing an RPC mechanism

If you are using an RPC mechanism—which is how clients and servers communicate—you must initialize it before the final stage of initializing Trans-C. Whether you need to initialize Tran-C in stages depends on the requirements of the RPC mechanism. If you do not need to initialize Tran-C in stages, you can simply initialize Tran-C (using the **initTC** function) after initializing the RPC mechanism. The **tc_InitTRPC** function can be used to initialize the Transactional Remote Procedure Call (TRPC) service. No Tran-C initialization is required before calling this function. This function has the following syntax:

```
void tc_InitTRPC(void);
```

The example shown in Figure 9 on page 50 calls **trpc_InitWithTrdce** before calling **tc_InitTRPC**. The **trpc_InitWithTrdce** function informs TRPC that TRDCE functions have been used for server registration and requests that TRPC use any protocol sequences and well-known endpoints described by those TRDCE calls.

If you use only Tran-C and TRPC and you do not use one of the Tran-C routines to initialize TRPC (**tc_InitTRPC** or **initTCWithTRPC**), you can initialize just the TRPC callbacks for your application. You do this by calling the **registerTRPCCallbacks** function. This function has the following syntax:

```
void registerTRPCCallbacks(void)
```

If you use this function to directly register the callbacks required by TRPC, you must explicitly terminate TRPC by calling the TRPC **trpc_Terminate** function.

Single-step initialization of Tran-C and TRPC

If you are using only Tran-C and TRPC, you can use the **initTCWithTRPC** function, which initializes both the Tran-C runtime system and the TRPC communication mechanism. This function has the following syntax:

```
void initTCWithTRPC(void);
```

This function takes no arguments and returns no value. It is not necessary to call either **initTC** or **tc_InitTRPC** after using this function.

Sample client initialization

The example shown in Figure 9 on page 50 uses the two-stage method (see "Two-Stage initialization of Tran-C" on page 48). After calling **preInitTC**, the program sets the environment for TRPC, initializes TRPC, and then calls **postInitTC**. It then registers a function to convert local abort codes to strings (see "Defining abort reasons by using abort codes" on page 65).

```
/* Begin Tran-C initialization. */
preInitTC();
/* Initialize TRPC. */
status = trpc_InitWithTrdce();
CHECK_STATUS(status);
tc_InitTRPC();
/* Finish Tran-C initialization. */
postInitTC();
}
```

Figure 9. Sample client initialization routine

Beginning and ending transactions

The most important C language extension provided by Tran-C is the **transaction** construct and its associated **onAbort** and **onCommit** clauses. The syntax of the Tran-C transaction construct is shown in Figure 10.

```
transaction
    statement
onCommit
    statement
onAbort
    statement
```

Figure 10. Syntax of the transaction construct

The **transaction** construct consists of a **transaction** clause, an **onCommit** clause, and an **onAbort** clause. Each clause starts with the relevant keyword and is followed by a statement, which can be a compound C statement. Multiple statements in the **onAbort** or **onCommit** clauses, or in the primary transaction clause, must be enclosed within braces, just as any other complex C language construct. Each transaction construct must contain an **onAbort** clause. The **onCommit** clause is optional and if specified, must precede the **onAbort** clause. The **transaction** statement and its associated clauses comprise a compound C statement, and must therefore be enclosed within brackets whenever it is used in place of a simple C statement.

When a **transaction** statement is executed, a transaction is created, and all of the statements in the **transaction** clause are executed within the scope of that transaction. This scoping is dynamic—any functions called from within the transaction statement are executed within the scope of that transaction. If the transaction is not aborted before the end of the **transaction** clause is reached, the Transaction Service attempts to commit the transaction. Any statements

specified in the **onCommit** clause are executed only after the transaction is successfully committed; they are executed outside the scope of any transaction.

Within the scope of a **transaction** construct, you must be careful if you use C-language statements that transfer control unconditionally. Examples of these statements are **goto**, **return**, **break**, and **continue**. Because Tran-C creates a transaction scope in which code is executed, transferring control out of this scope (including any transaction constructs discussed in this document) prevents Tran-C from controlling the scope of the transaction.

If a transaction is aborted before the end of the transaction clause, or if the transaction cannot be committed, the statements in the **onAbort** clause are executed. These statements are executed outside the scope of any transaction. When a transaction is aborted, the flow of control in the thread controlling that transaction automatically transfers to the abort clause.

This transfer of control actually occurs when the next Tran-C statement is executed, because the Tran-C runtime system verifies the status of all active threads and transactions each time a Tran-C statement is executed. To avoid potentially long delays in the transfer of control in applications that infrequently use Tran-C constructs, Tran-C provides the **abortCheck** function. This function probes the status of the current transaction to force the transfer of control if the transaction has aborted. See "Aborting transactions" on page 64 and "Monitoring transaction status" on page 72 for more information.

Statements in the **onCommit** and **onAbort** clauses are executed only after the outcome of the associated transaction is known. If the transaction is aborted, the Tran-C functions **abortModuleName** and **abortReason** can be used to retrieve information about the reason for an abort from within an **onAbort** clause. Similarly, the **getCompletedTid** function can be used from within an **onCommit** or **onAbort** clause to get the transaction identifier of the last completed transaction. These functions are discussed in the next section.

Figure 11 on page 52 shows the **transaction** construct. Each call to **TakeOrder** generates a new transaction.

```
static void TakeOrder()
{
    inFunction("TakeOrder");
    transaction {

       /* Perform steps that are part of the transaction. */
   }
    onCommit
        printf("Order processed.\n");
    onAbort
        printf("Order aborted:  %s (%s)\n", abortReason(),
                abortModuleName());
}
```

Figure 11. The transaction construct

Getting information about a transaction

Tran-C provides several functions for obtaining information about current, committed, and aborted transactions. As mentioned earlier, transactions are named by using TIDs that are managed and maintained internally by the Tran-C runtime system. TIDs are instances of the **tran_tid_t** type, which is a type exported by the Encina Transaction Service (for more information on the **tran_tid_t** data type, see the *Encina Toolkit Programming Guide*). The Tran-C **getTid** function returns the TID of the current transaction. This function must be called from within a transaction. It returns the value `TRAN_TID_NULL` when the thread that called **getTid** is not being executed within the scope of a transaction.

You can determine whether the calling thread is being executed within the scope of a transaction by calling the **inTransaction** function. This function returns a nonzero (TRUE) value if the calling thread has a valid transaction identifier associated with it.

The Tran-C **currentModuleName** and **currentFunctionName** functions retrieve the names of the module (as registered with the **inModule** macro) and function (as registered with **inFunction**) that are currently being executed. Tran-C provides two similar functions, **abortModuleName** and **abortFunctionName** to retrieve information about the module and function in which a transaction aborted. These functions are described in "Determining where a transaction aborted" on page 75.

Tran-C also provides functions that retrieve information about a transaction that has just committed or aborted. Because the expected outcome of most transactions is successful completion, it is not ordinarily necessary to take any special action when this is the case. The **getCompletedTid** function determines the identifier of the transaction that just completed. Tran-C

provides the **getContainingTid** function to obtain information about the parent of a nested transaction. While both of these functions can be called from within either the **onAbort** or **onCommit** clauses of a transaction, they are most frequently used in the context of an **onAbort** clause. For complete information about these functions, see "Determining transaction identifiers" on page 74.

Nested and top-level transactions

Subtransactions are transactions begun within the scope of another transaction by nesting a transaction construct within a transaction construct. Subtransactions can be nested to any depth. A series of nested transactions is viewed as a hierarchy of transactions. A transaction which spawns another transaction by using a nested transaction construct, creating a nested subtransaction, is referred to as the *parent* of that subtransaction. The transaction that is the parent of the entire tree (family) of transactions is referred to as the *top-level transaction*, as shown in Figure 12:

```
transaction {
   ...
     transaction {
        ...
     }
     onCommit{
        ...
     }onAbort{
         ...
     }
}onCommit{
        ...
}onAbort{
        ...
   }
```

Figure 12. A nested transaction

Note: If your application accesses a relational database management system (RDBMS) supported by Encina, for example DB2 or Oracle, do not use nesting. The XA interface does not support subtransactions.

By default, subtransactions are executed sequentially within the scope of their parent transaction. The Tran-C **concurrent** and **cofor** statements can be used to create new threads of control and subtransactions that run concurrently on behalf of their parent transaction. For more information, see "Creating concurrent transactions or synchronous threads" on page 54.

Nested subtransactions commit with respect to their parent transaction. The statement enclosed in the **onCommit** clause or the **onAbort** clause of a nested subtransaction is executed when that subtransaction completes. Even though

statements in these clauses are executed when the subtransaction completes, the permanence of their effects depends on the parent transaction committing. For this reason, any statement displayed by the **onCommit** clauses of nested subtransactions is seen as a *suggestion* of the eventual outcome of that transaction family, and not as proof that the family will commit.

Using the topLevel construct

Under some circumstances, a transaction can need to create another transaction but to have that transaction run independently, with no hierarchical relationship between the two. This is done by using the Tran-C **topLevel** construct. The syntax of this construct, shown in Figure 13, is identical to that of the **transaction** construct, except that the **topLevel** keyword is used instead of the **transaction** construct's **transaction** keyword. Each clause's statement can be a compound C statement.

```
topLevel
    statement
onCommit
    statement
onAbort
    statement
```

Figure 13. Syntax of the topLevel construct

When a top-level transaction is created within the scope of another transaction, a new transaction is created. This transaction is referred to as a *nested top-level transaction* because the execution of the parent transaction is suspended until the newly created top-level transaction completes (either commits or aborts). The top-level transaction executes outside the scope of the transaction in which it was created.

A nested transaction created by using the **topLevel** construct commits or aborts independently of the transaction that created it. This independence enables such a transaction to do work that is not undone if the transaction that created it aborts. This behavior differs from that of nested transactions created by using the standard **transaction** construct, which commit or abort only with respect to their parent transaction. Using a **topLevel** construct outside the dynamic scope of a **transaction** construct is equivalent to using the **transaction** construct.

Creating concurrent transactions or synchronous threads

The **transaction** and **topLevel** constructs execute only a single nested transaction at a time. To take advantage of the multiprocessing capabilities provided by the use of threads, Tran-C provides two constructs that enable applications to create multiple concurrent threads, the **concurrent** and **cofor** constructs. Both of these constructs create multiple concurrent threads that can run either as subtransactions or as concurrent threads within the scope of the

current transaction. During the lifetime of these concurrent threads the thread executing the **concurrent** or **cofor** construct waits for all threads created by the construct to terminate.

The **concurrent** construct enables an application to execute a predetermined number of threads concurrently , each performing a specified function. The **cofor** construct enables an application to execute a variable number of threads concurrently, all performing the same function. These threads can be executed in the context of subtransactions or in the context of the transaction that is current when the **concurrent** or **cofor** construct is executed. The context in which these threads start depends on the constructs used inside the **concurrent** or **cofor** statements (see "The subTran and subThread constructs").

The **cofor** construct takes an argument that determines the number of concurrent subtransactions to be created. The **concurrent** construct contains a number of statements for which new threads are created. When either of these constructs is executed outside the scope of a transaction, the appropriate number of concurrent top-level transactions are created.

For more information, see "Using the concurrent construct" on page 57 and "Using the cofor construct" on page 58.

Note: Do not use Tran-C constructs in threads that can be canceled. If a thread is canceled during the execution of a Tran-C construct or function, then the state of the transaction on whose behalf the call was made is undefined.

The subTran and subThread constructs

The function to be performed by each thread created by either the **concurrent** or **cofor** construct is specified by either a **subTran** or **subThread** construct. Both of these create concurrent threads to execute specified functions. Threads created by the **subTran** construct are executed as subtransactions, while threads created by the **subThread** construct are executed within the scope of the current transaction. Whenever a **concurrent** or **cofor** construct is executed, the thread executing it waits until all the created threads terminate.

Since threads created by the **subTran** construct are created as subtransactions, they are assigned their own TIDs, can have an optional **onCommit** clause, and must have an associated **onAbort** clause. Threads created by using the **subThread** construct do not have **onAbort** or **onCommit** clauses because they are executed within the context of the current transaction, not as subtransactions.

Figure 14 on page 56 shows the format of a **subTran** construct. A **subTran** clause consists of the **subTran** keyword, followed by a pointer to a function and a pointer to the arguments required by that function. The specified

function cannot return a value. When a **subTran** clause is executed, both a new thread and new transaction are created. Like any other transaction construct, it must have an associated **onAbort** clause and can have an optional **onCommit** clause.

```
subTran
    (void (*funcPtr)(void *), void *)
onCommit
    statement
onAbort
    statement
```

Figure 14. Syntax of the subTran construct

A statement in the **onAbort** or **onCommit** clause is executed only after all concurrent subtransactions (created by a single **concurrent** or **cofor** construct) have completed. Each **onAbort** or **onCommit** clause is executed in the order in which it appears in the **concurrent** or **cofor** statement, and which clause is executed depends on whether the relevant subtransaction committed or aborted. Like the **onCommit** and **onAbort** clauses of other Tran-C constructs, the statements in these clauses of the **subTran** construct are executed outside the scope of any transaction; see "Beginning and ending transactions" on page 50.

The **concurrent** statement and the **cofor** statement can contain **subTran** statements or **subThread** statements. Threads created using the **subThread** clause are executed concurrently, directly within the scope of the parent transaction. The syntax of the **subThread** clause is shown in Figure 15.

```
subThread(void (*funcPtr)(void *), void *);
```

Figure 15. Syntax of the subThread construct

Threads created by using the **subThread** clause differ from those created by using the **subTran** clause because they do not create new transactions and therefore do not have associated **onCommit** or **onAbort** clauses. All threads created by using **subThread** clauses are executed within the scope of the transaction that executed the **concurrent** statement.

The **subThread** construct is not well suited to data manipulation because of potential problems in its interaction with the Tran-C locking mechanism. Locks prevent invalid concurrent access to data; they are obtained and held on data based on the TID of the transaction that obtained them. The Lock Service determines if a thread can access locked data by checking the ID of the transaction under which it is running. This check determines if that transaction either obtained the lock or inherited it from a parent transaction.

However, multiple threads created by **subThread** clauses all run under the same TID, and therefore have equal ability to concurrently access the locked data.

A good example is a transaction that creates two threads by using the **subThread** statement. The first thread intends to update some data, so it obtains a write lock on that data. The second thread needs to read the same data, so it obtains a read lock on that same data. Because these two threads have the same transaction identifier, and a single TID can hold both a read and write lock simultaneously, both locks are granted. However, the first thread can be changing the locked data while the second is attempting to read it, which can introduce inconsistencies.

Using the concurrent construct

The **concurrent** construct uses the threading capabilities of Tran-C to create a predetermined number of transactions that run concurrently. The number of threads or subtransactions created is determined by the number of **subTran** or **subThread** statements enclosed by the **concurrent** clause. Using the **concurrent** construct within the scope of a transaction creates a predetermined number of concurrent threads, running either within the scope of the current transaction (if the **subThread** statement was used), or as subtransactions (if the **subTran** keyword was used). If the **concurrent** construct is used outside the scope of a transaction, the threads created by that statement are executed as concurrent top-level transactions.

A **concurrent** construct's **onCommit** and **onAbort** clauses provide a way to determine the outcome of the entire **concurrent** construct. The **onCommit** clause is executed only if all subtransactions created by the **concurrent** construct commit. The **onAbort** clause is executed if any of those subtransactions abort, and the abort code is set to `CONC_STMT_ABORT_CODE`. Calling the Tran-C **getCompletedTid** function from either the **onAbort** or **onCommit** clauses of a concurrent statement always returns the transaction identifier `TRAN_TID_NULL`.

The syntax of the **concurrent** statement is shown in Figures Figure 16 on page 58 and Figure 17 on page 58. The **concurrent** construct begins with the **concurrent** keyword, followed by multiple subtransaction or subthread clauses, which can be mixed freely. The construct can end with the **coEnd** keyword (shown in Figure 16 on page 58). Otherwise, the construct ends with an optional **onCommit** clause followed by an **onAbort** clause (as shown in Figure 17 on page 58). Use the **coEnd** keyword when it is unnecessary to determine whether the **concurrent** construct, as a whole, committed or aborted. Also, use this when all of the statements inside the **concurrent** clause are **subThread** statements because the **concurrent** construct has no transactional outcome status in this situation.

```
concurrent {
    subTran or subThread clause 1
    .
    .
    subTran or subThread clause N
} coEnd;
```

Figure 16. Syntax of the concurrent construct terminated by the coEnd statement

```
concurrent {
    subTran or subThread clause 1
    .
    .
    subTran or subThread clause N
}
onCommit
    statement
onAbort
    statement
```

Figure 17. Syntax of the concurrent construct with onCommit and onAbort clauses

Using the cofor construct

The **cofor** construct creates a variable number of concurrent subtransactions or threads. A **cofor** construct consists of the **cofor** keyword and two associated arguments, enclosed within parentheses, and a single **subTran** or **subThread** clause. The **cofor** construct is a threaded analog of a standard C **for** loop, where each iteration spawns a separate concurrent subtransaction or thread. The arguments to the **cofor** keyword determine the number of times the **subTran** or **subThread** clause is executed. The syntax of these clauses is explained in "The subTran and subThread constructs" on page 55. Like the **concurrent** construct, the **cofor** construct can be terminated either by a **coEnd** statement (as shown in Figure 19 on page 59), or by standard **onCommit** and **onAbort** clauses (as shown in Figure 18 on page 59). The **coEnd** keyword is used when it is not necessary to know whether the **cofor** construct committed or aborted. It is also used when the statement inside the **cofor** clause is a **subThread** statement and therefore has no transactional status outcome to report.

```
cofor(int, integer expression) {
    subTran or subThread clause
}
onCommit
    statement
onAbort
    statement
```

Figure 18. Syntax of the cofor construct with onCommit and onAbort clauses

The first argument is a previously declared integer variable that is used as a loop control variable for the **cofor** construct. The second argument is an expression that must yield an integer result. The result of the expression determines the number of times that the **subTran** or **subThread** clause in the **cofor** statement is executed. The value of the loop variable ranges from zero to the result of the expression, minus one. The expression is evaluated only when the **cofor** construct is first encountered, before the specified **subTran** or **subThread** statement is executed for the first time. When execution continues past the **cofor** statement, the loop variable has the value that is the expression's result.

Each execution of the loop creates a new subtransaction or synchronous thread. Typically, the arguments to the function specified in the **subTran** or **subThread** clause use the loop variable as an index for referencing some other data structure. This enables each iteration of the **subTran** or **subThread** statement to use different data and to generate and store unique results, for example, in different array locations.

```
cofor(int, integer expression) {
    subTran or subThread clause
} coEnd;
```

Figure 19. Syntax of the cofor construct terminated by the coEnd statement

Figure 20 on page 60 shows an example of the use of a **cofor** construct to update three replicas of a transactional server in parallel. Because the server is transactional, all three updates must occur successfully for the update to commit. For this reason, the **subTran** statement's **onAbort** clause is empty, because no additional information is gained by explicitly reacting to any single abort. If any of the updates fails, the entire **cofor** construct must be aborted to undo any successful updates.

```
void *arguments[3];        /* contains information to access
                              replica and its arguments */
extern void update(void*); /* update function */
int loopVar;               /* loop variable for cofor statement */
    cofor(loopVar, 3) {
        subTran(update, arguments[loopVar]);
        onAbort
            ;  /* interested only in over all outcome,
                     not that of specific subTrans */
    }
    onCommit
        printf("Update successful\n");
    onAbort
        abort("One or more updates failed\n");
```

Figure 20. Example of using the cofor construct

Exceeding thread limits

Many systems have a limit on the number of threads they can manage. The **concurrent** and **cofor** constructs create all threads, create all transactions when appropriate, and execute all functions, or they do nothing. If the system reaches the thread limit before all required threads have been created, these constructs terminate all newly created threads and report the problem in one of the following ways:

- If the **concurrent** or **cofor** construct contains an **onAbort** clause, the **onAbort** clause is executed, with the abort code CONC_STMT_INSUFF_THREADS_CODE.
- If the **concurrent** or **cofor** construct ends with a **coEnd** statement, the system displays a warning message.

Suspending transactions

Applications that need to access resources transactionally can also interact with nontransactional applications. When an application accepts a nontransactional RPC, it creates a transaction, performs the requested work, and suspends the transaction pending further requests from the nontransactional client. Subsequent requests can then resume the suspended transaction, perform additional work, and finally either suspend or commit it. Good examples of this sort of interaction are transactional servers that use information from a time service to keep their internal clocks synchronized, or servers that must simply acknowledge requests from a network process monitor to show that they are still active.

One way of optimizing the transactional costs of accessing the necessary resources when the requests come from a nontransactional application is to create one transaction for the application and *suspend* it after handling each request. Since the transaction can easily resume, this saves completing the

transaction and creating a new one each time a request arrives. Another way to optimize this situation is discussed in "Lazy transactions" on page 97, but this method is valid only in servers using recoverable data.

The suspend clause

The **suspend** clause is an optional transaction clause that can replace the **onCommit** clause in any nonnested **transaction** construct. The **suspend** clause causes that transaction to suspend execution rather than commit. Like the **onCommit** clause, the **suspend** clause must appear before the **onAbort** clause. The **suspend** clause can be used only in transaction statements that result in top-level transactions, such as nonnested uses of the **transaction** construct, or with **topLevel** constructs. Suspending the execution of any nested transaction is not allowed.

```
transaction
    statement
suspend(tran_tid_t *)
    statement
onAbort
    statement
```

Figure 21. Syntax of the suspend clause within a transaction construct

Figure 21 shows the syntax of the **suspend** clause, used within a **transaction** construct. A **suspend** clause consists of the **suspend** keyword, an argument to that keyword, and a single statement that is executed if the transaction completes or is suspended. The argument to the **suspend** keyword is a pointer to a TID, which is a variable of type **tran_tid_t**. Suspending the transaction sets this pointer to the TID that is returned by a call to **getTid** within the body of the **transaction** construct. Any statement can be associated with a successful suspension of the transaction. Because the transaction can be aborted before execution reaches the **suspend** clause, there are no guarantees that the statement in the **suspend** clause is executed. It is not allowed to call the **getCompletedTid** function in a **suspend** clause because the transaction has not completed (that is, neither committed nor aborted).

If a transaction is aborted while it is suspended, the **onAbort** clause is executed when the transaction resumes.

The resumeTran construct

The Tran-C **resumeTran** construct resumes a suspended transaction. Because only nonnested or top-level transactions can be suspended, suspended transactions can be resumed only at the same (top) level. If a suspended transaction is resumed within the scope of another transaction, the resumed transaction behaves as a nested top-level transaction.

```
resumeTran(tran_tid_t)
    statement
onCommit
    statement
onAbort
    statement
```

Figure 22. Syntax of the resumeTran construct

Figure 22 shows the syntax of the **resumeTran** construct. The **resumeTran** keyword requires one argument, which is the transaction identifier of a suspended transaction. Like any other top-level transaction construct, the **resumeTran** construct can have either an **onCommit** or **suspend** clause. If the **resumeTran** construct does not have a **suspend** clause and completes successfully, the transaction is committed. When a **resumeTran** construct has a **suspend** clause, the transaction is suspended again when that clause is reached.

Creating server-side transactions

Tran-C supports a mechanism for creating server-side transactions. A *server-side transaction* is a transaction that is initiated by the client but created and ended at the server. Server-side transactions run entirely at the server as top-level transactions. Transactions running at the client do not depend on the outcome of server-side transactions, so the client and server do not need to share transactional information. Eliminating the transactional information normally sent with transactional RPCs can improve the performance of the client application, but because the client and server do not share this information, only client applications that do not rely on transactional guarantees can use server-side transactions.

The wrapEachTrpc construct

A client application can use the **wrapEachTrpc** construct to define an execution scope; each transactional RPC made from within this scope is wrapped in a separate top-level server-side transaction. Nontransactional RPCs called within the wrapping scope are *not* executed as server-side transactions. The syntax of the **wrapEachTrpc** construct is shown in Figure 23.

```
wrapEachTrpc
     statement
onCommit
     statement
onAbort
     statement
```

Figure 23. Syntax of the wrapEachTrpc construct

The **wrapEachTrpc** construct consists of a **wrapEachTrpc** clause, an **onCommit** clause, and an **onAbort** clause. As in similar constructs, the **onAbort** clause is

required, and the **onCommit** clause is optional; if specified, the **onCommit** clause must precede the **onAbort** clause. A **wrapEachTrpc** construct can be nested within a transaction construct, and a transaction construct can be nested within a **wrapEachTrpc** construct.

Transactional RPCs made within the **wrapEachTrpc** clause result in server-side transactions that last for the duration of the RPC. If the server-side transaction aborts, the **onAbort** clause is executed. If a communication failure causes the abort, `UNKNOWN_ABORT_REASON` is returned as the abort code. If the server-side transaction completes successfully, the **onCommit** clause is executed (if it exists).

Multiple transactional RPCs can be called within the same **wrapEachTrpc** construct; they are executed sequentially, and each RPC creates a separate server-side transaction. If one of the server-side transactions is aborted by the server, control jumps to the **onAbort** clause of the **wrapEachTrpc** construct, and any remaining transactional RPCs are not executed.

Care must be taken when concurrent threads are nested within a **wrapEachTrpc** construct. All transactional RPCs made by the functions executed by **subThread** constructs create server-side transactions. Therefore, the flow of control can be different from what is expected.

Getting information about server-side transactions

Because server-side transactions are executed entirely at the server, the Tran-C functions used to determine transaction identity at the client do not return the identifier for the server-side transaction. In the **wrapEachTrpc** clause, calling the **getTid** function returns `TRAN_TID_NULL`, and calling the **getContainingTid** function returns the transaction identifier for the transaction that encloses the **wrapEachTrpc** construct. Similarly, in the **onAbort** and **onCommit** clauses, calling the **getCompletedTid** function returns `TRAN_TID_NULL`, and calling the **getContainingTid** function returns the transaction identifier for the transaction that encloses the **wrapEachTrpc** construct.

A client application can determine whether the current thread is being executed within a scope used for creating server-side transactions. The **inWrapEachTrpc** function determines whether the calling thread is executing within a **wrapEachTrpc** construct. If the function returns a nonzero (`TRUE`) value, the thread is executing within a *wrapping* scope, and each transactional RPC made in this scope begins and ends a transaction at the server. The **trpcPermitted** function can be used to determine whether the current execution context is one in which a transactional RPC can be made. If the calling thread is being executed either within a transaction or within a **wrapEachTrpc** construct, the function returns a nonzero (`TRUE`) value.

A server manager function can call the **inWrappedTran** function to determine whether the current transaction was created on the server in response to a client request for a server-side transaction. The **inWrappedTran** function returns a nonzero value (`TRUE`) if the current transaction at the server is a server-side transaction; otherwise, the function returns `0` (zero).

Aborting transactions

The most common reason why a transaction is aborted by the Tran-C runtime system is a communications or data-access failure. These types of aborts are automatically detected by the runtime system when the next Tran-C statement is executed. Transactions can also be explicitly aborted from within applications by the use of Tran-C functions.

When the runtime system detects an abort, the flow of control immediately jumps to the **onAbort** clause associated with the **transaction** construct (or similar construct) within which the code is running. The Tran-C runtime system detects an abort in a Tran-C construct only when the next Tran-C construct or statement is executed, but all **transaction** constructs implicitly check for aborts at the end of their bodies. When an abort is detected by a remote application that is performing work on behalf of a transaction, that application immediately stops performing work on behalf of the aborted transaction. The RPC then returns to the calling application, and that application transfers control to the associated **onAbort** clause since all TRPCs check the status of the transaction upon return.

All participants are eventually notified of a transaction's termination. When a transaction is explicitly aborted (for example, when the **abort** function is called), the call can return before the Transaction Service notifies any participants, including the application issuing the call. TRAN does not guarantee notification within any given time, except for the application that explicitly aborts a transaction. An application (or any of its communications or recovery services) is responsible for issuing timeouts for other transactions as necessary. An application can be affected in the following way: a client can begin a transaction, make a call to a server involving a significantly large amount of data, and abort the transaction when the call returns. Then the client can begin another transaction and makes a call to the same server to access some of the same data with conflicting locks. There is a possibility that the second transaction can wait or time out because the first transaction retains locks during its recovery procedures.

The next few sections describe how to define abort reasons, explain the Tran-C functions provided to explicitly abort transactions, describe how to obtain information about the aborted transaction, and describe the messages returned by the runtime system when a transaction is aborted by the system.

Defining abort reasons

Tran-C uses the Encina Abort Facility (described in the *Encina Toolkit Programming Guide*) to assign and retrieve information about an aborted transaction. This information, referred to as an *abort reason*, includes an abort code or string or both, a module and function name, and a format identifier for the abort reason. The application specifies the reason for aborting a transaction (typically as an abort code); any participant in the transaction can then determine the abort reason for the aborted transaction. The information contained in an abort reason is covered in greater detail in "Using abort data" on page 68.

In Tran-C, abort reasons can be defined by using either abort codes or abort strings. Abort codes enable abort reasons to be compared easily and can be encoded in such a way that they can be converted to an NLS-compliant string for printing. Abort strings are variable-length strings can possibly be generated in a different NLS locale; therefore, abort strings cannot be compared as easily as codes.

The following sections cover defining abort codes, abort strings, and other abort data, and briefly describe the mechanisms underlying the management of abort reasons in Tran-C.

Defining abort reasons by using abort codes

An *abort code* is a signed integer constant that describes the reason why a transaction aborted. In a Tran-C application, define an abort code for each different condition under which the application aborts transactions. The example shown in Figure 24 on page 66 defines one abort code called `CANCELED_BY_USER`.

Using abort codes requires that a means for translating codes to a format appropriate for the application, such as an NLS-compliant string, must be provided. Tran-C supplies a mechanism and defines conventions for doing this. In addition to defining abort codes, you must also take the following steps:

1. Define an abort format identifier and specify its scope.
2. Define a formatting function with a specific format identifier that formats the abort code or data or both.
3. Register the format identifier and its associated formatting function with the application.

An abort format identifier must be defined so that Tran-C can associate a formatting function with the abort reason generated by an aborted transaction. The format identifier is a DCE universal unique identifier (UUID) that

uniquely identifies the format for abort reasons. This UUID can be created with the **uuidgen** utility provided by the DCE; the format identifier is referred to as the *format UUID*.

The **useAbortFormat** function can be used to specify the scope of the format UUID. If this function is called at the beginning of a source file (before any functions), the scope of the format UUID is the file. If the function is called at the beginning of a function within a source file, the scope is limited to that function. If different scopes are specified, different abort formatting functions can be used based on the scope in which a transaction aborts.

The syntax of the **useAbortFormat** function follows:

```
void useAbortFormat(char *formatUuidString);
```

This function takes one argument, which is an abort format UUID in string form. The definition of this format UUID is shown in Figure 24.

```
/* Abort codes and format */
typedef enum {
   CANCELED_BY_USER = 1
} abortCode_t;
static char ABORT_FORMAT[] = "0014ad20-e154-1d68-85b0-9e62092caa77";
inModule("myApp");
useAbortFormat(ABORT_FORMAT);
```

Figure 24. Defining the abort code

A formatting function must be defined for abort reasons. The purpose of the formatting function is to take the information in an abort reason and use it to generate output appropriate to the application. When invoked, the formatting function is automatically passed two arguments: a pointer to an abort reason for the aborted transaction and a pointer to a buffer. By default, the buffer has a maximum size of `ENCINA_MAX_STATUS_STRING_SIZE` bytes.

The **AbortFormatter** function is shown in Figure 25 on page 67. The example function checks the abort code set for the abort reason, and based on the value of the abort code, returns a string describing the reason for the abort in the *bufferP* parameter. Note that this example generates a printable string that is not NLS-compliant.

```
/* AbortFormatter - converts integer abort code contained in
 * abortReasonP to a string. */
static void AbortFormatter(
encina_abortReason_t *abortReasonP, char *bufferP)
  {
    char *abortString;
    switch(abortReasonP->code) {
      case CANCELED_BY_USER:
        abortString = "User canceled the order.";
        break;
      default:
        abortString = "Unknown abort code.";
    }
    strcpy(bufferP, abortString);
}
```

Figure 25. Example function for formatting an abort reason

After the formatting function is defined, it must be associated with a format UUID and registered with an application. The **encina_RegisterAbortFormatter** function is used. This function takes two arguments: a pointer to a format UUID and the name of a function. The format UUID must be of type **uuid_t** (the **uuid_from_string** DCE function can be used to convert a string to this type, if necessary).

Once the function is registered, calling the Tran-C function to retrieve an abort reason string for an aborted transaction automatically invokes the formatting function to return the abort reason string.

Defining abort reasons using strings

The Tran-C **abort** or **abortNamedTran** functions can be used to define an abort reason. The string passed as the argument to either of these functions is returned as the abort reason for the transaction in which the call was made (see "Using abort reason strings" on page 70).

When the **abort** or **abortNamedTran** function is used to define an abort reason for a transaction, the `ENCINA_STRING_FORMAT_UUID` variable is set as the format UUID for the abort reason, the abort code is set to zero by default, and Tran-C automatically registers a formatting function for that abort reason. Calling the **abortReason** function invokes this formatting function, which simply returns the abort string as a null-terminated character string. Refer to the *Encina Toolkit Programming Guide* for more information on the `ENCINA_STRING_FORMAT_UUID` variable.

Though it is easier to define abort reasons with strings than it is with codes, the usefulness of abort strings is limited and less flexible. For example, the use of strings makes the direct comparison of abort reasons difficult.

Using abort data

An Encina abort reason is a structure of type **encina_abortReason_t** that contains information about an aborted transaction when defined by a Tran-C-generated abort; some of the information in an abort reason is only supplied by Tran-C (for example, the name of the module in which the transaction aborted). An abort reason structure consists of the following:

- A format identifier—a DCE UUID (universal unique identifier) uniquely identifying the abort reason.
- An abort code—a signed, 32-bit integer defining the reason for aborting a transaction.
- Abort data—a structure consisting of the module name, the function name, and possibly a string describing the reason for an aborted transaction (all null-terminated strings). Abort data can also contain additional information that further qualifies an abort reason.

Refer to the *Encina Toolkit Programming Guide* for more information on the **encina_abortReason_t** data type.

The **setAbortData** function can be used to add additional data to an abort reason. This function must be called in the thread currently running on behalf of the transaction, and it must be called immediately before a function is called to abort the transaction, for example, the **abortWithCode** or **abort** functions. The Tran-C **setAbortData** function has the following syntax:

```
void setAbortData(void *dataP, unsigned long length);
```

The **setAbortData** function takes two arguments: a pointer to the abort-specific data and the length of the abort-specific data. The abort-specific data can be of any form, though the caller must ensure that data that is platform specific is correctly marshaled, since the data is copied exactly.

Note that Tran-C limits the maximum size of all the abort data for an abort reason to `ENCINA_MAX_STATUS_STRING_SIZE` bytes. If the size of the entire abort data exceeds this limit, the abort-specific data (set with the **setAbortData** function) is truncated.

Using abort reasons with other Encina components

For Tran-C applications that use other Encina Toolkit products, Tran-C provides a way for those applications to share abort reasons. When a transaction is explicitly aborted in a Tran-C application (for example, by the Tran-C **abort** or **abortWithCode** functions), the reason for the abort is automatically associated with the transaction through the Encina Abort Facility.

The Encina Abort Facility is a library of interface functions designed to provide generalized support for abort reasons. This facility is used by all the

Encina components that use abort reasons so that abort reasons are always handled in a consistent way. The Encina Abort Facility is documented in the *Encina Toolkit Programming Guide*.

Aborting transactions from within an application

Tran-C provides four functions that explicitly abort transactions. The **abort** and **abortWithCode** functions abort the current transaction. The **abortNamedTran** and **abortNamedTranWithCode** functions abort a specified transaction.

Using abort codes

Tran-C provides two functions to use an abort code when explicitly aborting transactions. The **abortWithCode** function enables an application to abort the current transaction, and the **abortNamedTranWithCode** function enables a Tran-C application to abort a specified transaction. A transaction can be aborted only by a thread being executed within a named Tran-C module. See "Registering module and function names" on page 46 for information about declaring Tran-C modules.

The Tran-C **abortWithCode** function has the following syntax:

```
void abortWithCode(long abortCode);
```

This function enables a program to abort the transaction from which the function was called. Calling this function in a process operating outside of the scope of a transaction is not allowed. The **abortWithCode** function requires one argument, an integer abort code that describes the reason for the abort. The Tran-C **abortCode** function (explained in "Retrieving general abort information" on page 72) can be used in the associated **onAbort** clause to retrieve this code. Remember that the reason for an abort can be determined only in the **onAbort** clause of the transaction statement that created the aborted transaction.

In the example shown in Figure 26 on page 70 the **abortWithCode** function is called to abort the transaction with the `CANCELED_BY_USER` abort reason code as its argument. Note that in the associated **onAbort** clause, the **abortReason** function is used to return the abort reason string to the user.

```
/* TakeOrder -- interactively construct and process an order. */
static void TakeOrder()
{

    inFunction("TakeOrder");
    transaction {

        /* Ask user whether transaction is to be submitted. */
        /* If not, cancel this transaction. */
        abortWithCode(CANCELED_BY_USER);
     }
    onCommit
        printf("Order processed.\n");
    onAbort
        printf("Order aborted:  %s (%s)\n", abortReason(),
                abortModuleName());
}
```

Figure 26. Example of aborting a transaction with an abort code

The Tran-C **abortNamedTranWithCode** function has the following syntax:

```
void abortNamedTranWithCode(tran_tid_t tid, long abortCode);
```

The first argument to this function is the TID of the transaction to be aborted. The second argument is an integer abort code that describes the reason for aborting the transaction. This function provides a powerful mechanism for explicitly aborting transactions, because the thread calling this function does not have to be within the scope of the transaction being aborted.

Using abort reason strings

Tran-C also provides two functions that use abort reason strings when explicitly aborting transactions. The **abort** function enables an application to abort the current transaction, and the **abortNamedTran** function enables a Tran-C application to abort a specified transaction. A transaction can be aborted only by a thread running within a named Tran-C module. See "Registering module and function names" on page 46 for information about declaring Tran-C modules.

The Tran-C **abort** function has the following syntax:

```
void abort(char * abortString);
```

This function enables a program to abort the transaction from which the function was called. Attempting to initiate an abort when the initiating process is operating outside of the scope of a transaction is not allowed. The **abort** function requires one argument: a character pointer to a string describing the reason for the abort. The Tran-C **abortReason** function (explained in "Retrieving general abort information" on page 72) can be used in the associated **onAbort** clause to retrieve this string. Remember that the

reason for an abort can be determined only in the **onAbort** clause of the transaction statement that created the aborted transaction.

The syntax of the **abortNamedTran** function is the following:

```
void abortNamedTran(tran_tid_t tid, char * abortString);
```

The first argument to this function is the TID of the transaction to be aborted. The second argument is a pointer to a null-terminated string that describes the reason for aborting the transaction. This function provides a powerful mechanism for explicitly aborting transactions, because the thread calling this function does not have to be within the scope of the transaction being aborted.

Executing statements before transferring control on abort

The Tran-C **catchAbort** construct provides a way for an application to intercept an abort and do some mandatory processing before transferring control to the **onAbort** clause of the current transaction. At this point, the transaction has already aborted, but some processing can still be required before control is transferred to the **onAbort** clause. The **catchAbort** clause can be used inside any Tran-C application and can be used with routines called from the **transaction**, **topLevel**, and **resumeTran** constructs. The primary function of the **catchAbort** construct is to provide a way to ensure that locally allocated resources are deallocated. "Using exceptions in Tran-C applications" on page 77 provides further information on the dynamic flow of control and catching aborts. The syntax of the **catchAbort** function is shown in Figure 27.

```
catchAbort {
    statement
}
onAbort  {
    statement
}
```

Figure 27. Syntax of the catchAbort clause

The **catchAbort** construct can enclose any number of C-language statements comprised by a single compound statement. When a **catchAbort** construct is executed, the statements in the **catchAbort** clause execute as if they appeared in the text of the program without any **catchAbort** construct surrounding them. If the current transaction aborts while the statements are being executed, control passes immediately to the statements inside the **catchAbort** construct's **onAbort** clause. If the current transaction progresses without aborting until flow of control exits the dynamic scope of the **catchAbort** construct, the statements within the **onAbort** clause are ignored. Flow of control transfers to the statement immediately following the **onAbort** clause.

The statements in the **onAbort** clause are executes outside the scope of any transaction. When functions containing the **catchAbort** construct are called from outside a transaction, the **catchAbort** clause is essentially invisible, and the statements inside the **onAbort** clause are ignored.

Monitoring transaction status

If a transaction aborts, control transfers from the main clause of the transaction construct to its associated **onAbort** clause. Because the Tran-C runtime system checks the status of active threads and transactions only when a Tran-C statement is executed, the actual transfer of control does not occur until the next Tran-C statement is executed. This means that the transfer can be delayed if the transaction is long running and uses Tran-C constructs infrequently.

To avoid any delay in the transfer of control, the **abortCheck** function can be called periodically within the scope of a transaction. Executing the **abortCheck** function causes Tran-C to check the status of the transaction; control is transferred to the **onAbort** clause if the transaction has aborted. The syntax of the **abortCheck** function follows:

```
void abortCheck(void);
```

This function takes no arguments and returns no value. Its sole purpose is to provide a means for the Tran-C runtime system to signal a long-running application that a transaction has aborted.

Getting information about aborted transactions

Tran-C provides several different types of functions that retrieve information about an aborted transaction. These functions are described in the next few sections.

Retrieving general abort information

When a transaction aborts, there are four Tran-C functions that can be called to retrieve some general information about the abort. These functions can be called only from within the **onAbort** clause of a transaction or from an abort callback, and the value returned by each of these functions is valid only within the **onAbort** clause from which the function is called. See "Registering and using callbacks" on page 81 for more information.

Tran-C provides two functions that can be called to retrieve information about why the transaction aborted: the **abortReason** function and the **abortCode** function. These functions are commonly used with the **abortModuleName** and **abortFunctionName** functions (see "Determining where a transaction aborted" on page 75), which identify the location at which the associated transaction aborted.

The syntax of the **abortReason** function follows:

```
char* abortReason(void);
```

The **abortReason** function returns a string describing the reason that the current transaction aborted. Typically, this string is either the string passed to the **abort** function or a string returned by a formatting function if the **abortWithCode** function was used to abort the transaction. If the reason was defined as an abort code, but no formatting function was registered for the code's format, the **abortReason** function returns an error string. See "Defining abort reasons by using abort codes" on page 65 for more information.

If a transaction is aborted by a non-Transactional-C module (or a component other than the Encina components used to support Transactional-C), the **abortReason** function returns a string indicating that the reason is unknown.

The syntax of the **abortCode** function is the following:

```
long abortCode(void);
```

The **abortCode** function returns an integer code giving the reason that the current transaction aborted. This code is the value passed to the Tran-C **abortWithCode** function when it is used to abort a transaction. Call the **abortFormat** function before the **abortCode** function to ensure that the abort reason for the aborted transaction is a properly formatted abort code.

Tran-C provides the **abortFormat** function to retrieve information about the format of the abort reason for an aborted transaction. This function can be used to determine if the abort reason for the transaction is an abort reason code and is properly formatted. If the **abortFormat** function returns a null pointer, the abort reason is not formatted as an abort reason code and, therefore, calling the **abortCode** function to obtain an abort reason code is invalid.

The syntax of the **abortFormat** function is the following:

```
uuid_t* abortFormat(void);
```

The **abortFormat** function returns the abort format UUID associated with the aborted transaction. This UUID is the identifier of the formatting function passed to the Tran-C **useAbortFormat** function. The UUID returned by the **abortFormat** function remains valid for the duration of the **onAbort** clause and must not be deallocated. See "Defining abort reasons by using abort codes" on page 65 for more information.

Tran-C provides the **getAbortData** function to retrieve abort-specific data stored with an abort reason. The returned data is not guaranteed to be aligned on a word boundary.

The syntax of the **getAbortData** function is the following:

```
void getAbortData(void **dataPP, unsigned long *lengthP);
```

The **getAbortData** function returns a pointer to the length of the abort data and a pointer to a pointer to the abort-specific data in *dataPP*. The **setAbortData** function can be used to set the abort-specific data for an abort reason as described in "Using abort data" on page 68.

In rare situations, the abort reason used to abort a transaction can be different from the abort reason retrieved in the **onAbort** clause of that transaction. For example, when users abort committed subtransactions, the Transaction Service must abort the first uncommitted ancestor to recover the subtransaction's work. When this occurs, the abort reason used for the subtransaction is obscured by an alternative abort reason supplied by the Transaction Service to indicate it had to abort an ancestor of a committed subtransaction. Another example is when a server aborts a transaction and communications are delayed between the client and server; the client can retrieve a different abort reason from the one used at the server.

Do not design applications whose correctness requires accurate retrieval of abort reasons. The relaxed functionality of abort reasons is important for ensuring efficient performance and the ability to abort transactions autonomously.

Determining transaction identifiers

The statements in the **onAbort** and **onCommit** clauses of a transaction construct are executed outside the scope of the aborted or committed transaction. The **getTid** function cannot be used successfully in these clauses because the transaction that transferred control to those clauses is no longer active. Using the **getTid** function in these cases returns misleading information.

Tran-C provides the **getCompletedTid** function to determine the identifier of the transaction that just completed. This function's only valid use is in **onCommit** or **onAbort** clauses. The **getCompletedTid** function returns the identifier of the transaction that just completed and transferred control to the clauses in which **getCompletedTid** is called. If called from outside a transaction's **onAbort** or **onCommit** clauses, the **getCompletedTid** function generates a fatal Tran-C runtime error. The syntax of the **getCompletedTid** function follows:

```
tran_tid_t getCompletedTid(void);
```

Tran-C provides a similar function, **getContainingTid**, to retrieve the identifier of the parent of a nested transaction. The syntax of the **getContainingTid** function follows:

```
tran_tid_t getContainingTid(void);
```

Unlike the **getCompletedTid** function, the **getContainingTid** function can be called anywhere in an application. However, when called from within a completion clause of a transaction construct, this function returns the identifier of the transaction that is the parent of the completed transaction. If the completed transaction is a top-level one, the function returns TRAN_TID_NULL, as it does when called from any top-level transaction.

Determining where a transaction aborted

When an abort occurs, the module and function names set by using the Tran-C **inModule** and **inFunction** calls can be retrieved to accurately identify the source module and specific function where an abort was initiated. Using these functions to register the module and function names is described in "Registering module and function names" on page 46. The **abortModuleName** and **abortFunctionName** functions retrieve the current module and function name from within an **onAbort** clause.

The **abortModuleName** function takes no arguments and returns the string value set by using the **inModule** macro in the module in which the transaction aborted. The syntax of the **abortModuleName** function follows:

```
char* abortModuleName(void);
```

The **abortFunctionName** function takes no arguments and returns the string value set by using **inFunction** macro in the function that executing when the transaction aborted. The syntax of the **abortFunctionName** function follows:

```
char* abortFunctionName(void);
```

The strings returned by the **abortFunctionName** and **abortModuleName** functions remain valid for the duration of the **onAbort** clause and must not be deallocated.

Determining the cause of an RPC failure

Tran-C returns the RPC_FAILURE_CODE abort code to indicate than an RPC failure caused the transaction to abort. When this abort code is returned, you can retrieve the DCE status code associated with the RPC failure by calling the **commError** function.

The **commError** function takes no arguments and returns an integer status code. A value of 0 is returned if no DCE status code is found. The syntax of the **commError** function follows:

```
long commError(void);
```

The **commError** function must be called within an **onAbort** clause or from an abort callback. The DCE status code returned by the **commError** function is valid only within the **onAbort** clause or abort callback from which the

function was called. Figure 28 provides an example of how to check for an RPC failure and return the DCE status code as a string.

```
if (abortCode() == RPC_FAILURE_CODE)
{
   char dceErrorString[ENCINA_MAX_STATUS_STRING_SIZE];
   encina_StatusToString(commError(),
                         ENCINA_MAX_STATUS_STRING_SIZE,
                         dceErrorString);
   printf("RPC failure: %s\n", dceErrorString);
}
```

Figure 28. Retrieving an RPC status code

Tran-C abort reasons

Tran-C defines several abort reasons that it uses when internally aborting a transaction. They permit an application to recognize system aborts in addition to aborts initiated by the application itself.

When an internal system abort occurs, Tran-C returns the associated reason as an abort reason code. The application can call the **abortCode** function within an **onAbort** clause to retrieve the Tran-C abort code. The following Tran-C abort codes are defined for system aborts:

- `APPL_EXIT_CODE`—The **exitTC** function was called while the transaction was still active.
- `CAUGHT_ABORT_EXCEPTION_CODE`—An abort exception was raised inside a transaction construct. Typically, applications call a Tran-C function, such as **abortWithCode**, to abort transactions, and this function then raises the `ABORT_EXCEPTION` exception. If, however, an application raises the `ABORT_EXCEPTION` exception directly (see "Using exceptions in Tran-C applications" on page 77), then the abort reason code in the current transaction's **onAbort** clause is `CAUGHT_ABORT_EXCEPTION_CODE`.
- `CAUGHT_ADDRESS_EXCEPTION_CODE`—An exception was raised inside a transaction construct, causing the transaction to abort. This is the default type of DCE exception.
- `CAUGHT_STATUS_EXCEPTION_CODE`—An exception was raised inside a transaction construct, causing the transaction to abort, and a status value was associated with that exception using the DCE Exception-Returning Interface **exc_set_status** function.
- `CONC_STMT_ABORT_CODE`—A **concurrent** or **cofor** statement aborted because one or more of the subtransactions aborted. This always is the abort reason returned in the **onAbort** clause of a **concurrent** or **cofor** construct.

- CONC_STMT_INSUFF_THREADS_CODE—When a **concurrent** or **cofor** construct cannot create all the threads designated in the construct, it immediately terminates any threads it did create and executes the **onAbort** clause. The **onAbort** clause returns this code.
- DEADLOCK_DETECTED_CODE—Resources required by different transactions could not be allocated in the correct sequence to satisfy the requirements of all those transactions, so the transaction was aborted.
- MEMORY_EXHAUSTED_CODE—It was impossible to allocate additional memory required by a call to the Tran-C **tranMemAlloc** function, so the transaction was aborted.
- RPC_FAILURE_CODE—An RPC failed. The codes returned by this abort condition are specific to the various RPC mechanisms supported by Tran-C. For an example of such a failure, see "Transfer of control in transactions" on page 38. For an example of how to retrieve the DCE status code associated with an RPC failure, see Figure 28 on page 76 in "Determining the cause of an RPC failure" on page 75.
- SERVER_SHUTDOWN_CODE—The server was stopped from servicing requests from new transactions because the **quiesceTC** or **exitTC** function was called.
- UNKNOWN_ABORT_REASON_CODE—Tran-C was unable to determine why the transaction aborted.

Note: Tran-C also defines string constants that are equivalent to the code constants (without the _CODE appended). These abort reason string constants will not be supported in future releases of Encina; therefore, use the abort code constants.

Using exceptions in Tran-C applications

Tran-C provides integral support for raising, catching, and propagating exceptions as implemented by the DCE. *Exceptions* provide a way of returning error information about exceptional conditions back through multiple levels of procedure or function calls, propagating this information until a function or procedure is reached that can take appropriate action. When an exceptional condition occurs, the exception related to that condition is *raised*. That exception then is propagated back through the procedures or function calls that lead to the routine in which the exception was raised, and it stops when one of those procedures or functions *catches* the exception. Once an exception has been caught, it can be reraised and propagated to calling functions or procedures. This behavior enables multiple functions to respond differently to a single exception, performing intermediate cleanup or processing related to that particular exception condition.

For an application to handle such exceptions correctly and propagate that information back through multiple levels of function calls, exceptions must occur within previously defined exception scopes. An *exception scope* is a

delimited portion of application code within which a particular exception has special significance. DCE provides macros for defining an exception scope within which exceptions can be raised and caught. Exceptions are declared as variables of type **EXCEPTION**. A scope is declared by using the **TRY**, **CATCH**, and **ENDTRY** macros. An exception is raised by using the **RAISE** macro. If an exception is not caught within an exception scope and therefore is raised within the default scope of the current thread, that thread terminates. Threads created by Tran-C constructs, such as the **cofor** construct, also terminate under these circumstances. If a thread was running on behalf of a transaction, that transaction is aborted, as described later in this section. If a thread was running outside the scope of a transaction, Tran-C generates a warning message describing the type and value of the uncaught exception that caused the thread to terminate.

For more detailed information about exceptions or about the macros provided in the DCE exceptions package, see the *OSF DCE Application Development Guide*.

The Tran-C **transaction** construct creates a dynamic scope that is equivalent to a DCE exception scope. Within the scope implicitly defined by a **transaction** construct, you can declare additional exception scopes by using the DCE exception macros and raise exceptions. If an exception is raised, and there is no intervening exception scope before the implicit scope of a transaction statement, then the transaction statement catches the exception and aborts the transaction. It uses one of two abort reasons, either `CAUGHT_ADDRESS_EXCEPTION_CODE` for address exceptions or `CAUGHT_STATUS_EXCEPTION_CODE` for status exceptions. Address exceptions are the standard type of DCE exception. Status exceptions differ from address exceptions in that they also return a status value, which must have been set by the DCE **exc_set_status** function. This status value can be returned by the **exc_get_status** function.

Tran-C also enables you to use the exception mechanism to transfer control on abort. Tran-C uses the `ABORT_EXCEPTION` exception to abort transactions. By default, this exception immediately transfers control to the **onAbort** clause of the transaction statement within whose scope the `ABORT_EXCEPTION` exception was raised. Unlike other exceptions, after it is caught by an **onAbort** clause, this exception is not reraised, .

If you abort a transaction by raising the `ABORT_EXCEPTION` exception, the transaction appears to have aborted in the function that initiated the transaction construct, regardless of where the transaction actually aborted. The abort reason code is `CAUGHT_ABORT_EXCEPTION_CODE`. This description assumes there were no intervening exception scopes that possibly caught the exception and taken some action on it, such as calling **abortWithCode**. It is better to use one of the Tran-C functions that explicitly abort transactions instead of raising

the ABORT_EXCEPTION exception, because they provide accurate function and module names for the **abortFunctionName** and **abortModuleName** functions in **onAbort** clauses.

Even though raising the ABORT_EXCEPTIONexception transfers control to the **onAbort** clause of the transaction construct within whose scope the abort occurred, you can also declare additional explicit exception scopes that catch the abort exception and perform some processing before actually transferring control to the **onAbort** clause. However, if you explicitly catch this exception, the exception must be reraised at the end of the **CATCH** clause. For example, the code fragment shown in Figure 29 is the equivalent of the use of the **catchAbort** clause shown in Figure 27 on page 71. The statements inside the **CATCH** clause are executed after the exception has been caught, and the exception is reraised as the last statement of this clause. If the ABORT_EXCEPTION exception is not reraised, the application can erroneously perform work on behalf of a transaction until an **abortCheck** or other Tran-C construct is executed, at which point the Tran-C runtime system raises the abort exception again.

```
TRY {
    statement
    .
    .
    statement
}
CATCH(ABORT_EXCEPTION) {
    ...      /* onAbort clause code */
    RERAISE; /* reraise the abort exception */
}
ENDTRY
```

Figure 29. Using exceptions to simulate the catchAbort statement

Exiting a Tran-C application

This section describes Tran-C functions that are used by applications that do not use the Encina Monitor. Encina Monitor applications do not need to call the functions described here because the Monitor handles exiting transactions.

Tran-C applications can exit by calling the **exitTC** function. The **exitTC** function quickly and cleanly terminates an application. It aborts all unprepared transactions; if there are prepared transactions (see "The two-phase commit protocol" on page 5) outstanding for the application, the **exitTC** function waits until they complete before terminating the application. If any prepared transaction cannot be resolved, the function times out and terminates the application. The syntax of the **exitTC** function follows:

```
exitTC(int status);
```

The **exitTC** function takes an integer value as its argument; the value specified is the status returned by the application program when it is terminated with the **exitTC** function. The **exitTC** function never returns.

Before an application terminates, it can be desirable to allow all outstanding transactions to complete. The **quiesceTC** function waits for all outstanding transactions to complete before returning. If any outstanding transaction does not commit or abort, the function times out and returns automatically, allowing the application to terminate. The syntax of the **quiesceTC** function follows:

```
quiesceTC(void);
```

The **quiesceTC** function takes no arguments. It must be called before the **exitTC** function. Calling the **quiesceTC** function before exiting is not required, but if the **quiesceTC** function is not called, the **exitTC** function can abort some transactions before terminating the application.

When shutting down a server, use the **quiesceTC** function with caution. If an application calls this function within the context of a transaction (for example, from within a TRPC manager function), the **quiesceTC** function waits until that transaction terminates; the function cannot return until either it or the transaction times out. To avoid this situation, the server can either call **quiesceTC** outside the context of any transaction, or call only the **exitTC** function.

For some applications, such as interactive client programs, it is useful to exit in response to program interrupts. Program interrupts are system-specific signals; for example, on UNIX platforms, the `SIGHUP`, `SIGINT`, and `SIGTERM` signals are considered interrupts. Interrupts, however, can possibly fail to terminate an application gracefully. The **exitTConInterrupt** function can be used to ensure that the application terminates gracefully when a program interrupt occurs. The syntax of the **exitTConInterrupt** function follows:

```
exitTConInterrupt(int status);
```

The **exitTConInterrupt** function takes an integer value as its argument; the value specified is the status to be returned by the application program when it is terminated by an interrupt. It is preferable to call the **exitTConInterrupt** function after Tran-C is initialized and before any transactional work is done, but you can call it at any point during program execution. After an application has called the **exitTConInterrupt** function, the application exits automatically (by using the **exitTC** function) whenever the program is interrupted.

Chapter 5. Advanced Tran-C programming

This chapter discusses more advanced topics of using Tran-C to develop transactional applications.

Registering and using callbacks

A callback is a function that an application registers with the system. The system invokes the callback function when certain events occur. These events can be relative to the life cycle of an application or to the life cycle of a transaction. Various services to which the client subscribes can also offer callback registration to enable a client to provide special actions for special events in those services. When one of these events occurs, and the callback is invoked, this is known as *delivering* the callback. This section discusses application and transaction callbacks for applications that do not use the Encina Monitor. See the *Encina Monitor Programming Guide* for information about registering callbacks in a Monitor application program.

Application callbacks

The Tran-C function **registerApplCallback** is used to register callbacks for the initialization or termination phases of a Tran-C application. For example, this function is used when registering a recovery service as part of the initialization of a Tran-C application. Functions registered for these phases of a Tran-C application must be registered before the phase occurs. For example, if an application uses one of the single-stage initialization methods (see "Initializing a Tran-C application" on page 47), it must register initialization callbacks to initialize a particular service at the correct time during Tran-C initialization. The application in this situation must register the callback before calling **initTC**.

The syntax of the **registerApplCallback** function follows:

```
void registerApplCallback(applCallback_t callbackType,
                                unsigned int (*callback) (void *),
                                void *callbackArg);
```

The **registerApplCallback** function takes several parameters. The *callbackType* parameter, of type **applCallback_t**, is either APPL_INIT_CALLBACK, indicating that this callback is being set for application initialization, or APPL_TERM_CALLBACK, indicating that the callback is being registered for application termination. The next two arguments are the callback itself and a pointer to any arguments required by the callback. This can be a pointer to a block of arguments if multiple arguments are required. Note that the callback functions must themselves return an integer value and take a **void*** argument.

To indicate an error, the callback can return the value `0` (`FALSE`). Returning any other value indicates that the callback was successfully executed. Errors in callback execution result in generating a runtime error.

The initialization callbacks are invoked during calls to initialize the Transaction Service, effectively between calls to the **tran_Init** and **tran_Ready** functions (see "Initializing a Tran-C application" on page 47 and the *Encina Toolkit Programming Guide*). The first function initializes the Transaction Service, and the second function finishes initialization after executing any registered initialization callbacks.

Termination callbacks (callbacks registered with the `APPL_TERM_CALLBACK` callback type) are provided to enable components to perform cleanup tasks before an application terminates (such as unregistering a server from the Directory Service). Before doing anything else, the **exitTC** function invokes termination callbacks.

Transaction callbacks

Both client and server applications can associate certain events with changes in the commit or abort status of a transaction. These programs can associate nontransactional activity with requests from transactional clients. For example, when a server must allocate resources to enable it to respond to a specific transactional request. Associating callbacks with transaction resolution provides an easy mechanism to ensure that those resources are deallocated when they are no longer needed.

The Tran-C **registerTranCallback** function enables client or server applications to explicitly register commit and abort callbacks for specific transactions. The syntax of the **registerTranCallback** function follows:

```
void registerTranCallback(tranCallback_t callbackType,
     void (*callback)(void*),
     void *callbackArg);
```

This function takes three arguments. The first argument specifies the type of callback:

- Callbacks that are issued when the transaction prepares
- Callbacks that are issued when the transaction commits
- Callbacks that are issued when the transaction aborts

The constants `TRAN_ABORT_CALLBACK`, `TRAN_COMMIT_CALLBACK`, or `TRAN_PREPARE_CALLBACK`are used to specify the callback type. The second parameter is the name of the function associated with the specified transaction state. The third parameter is a pointer to any arguments that will be passed to the callback function.

The **registerTranCallback** function can be called only from within the scope of a currently executing transaction, and can be associated only with a nonnested or top-level transaction. When this function is called, the Tran-C runtime system first checks to insure that a valid transaction identifier is present with which to associate the specified callback. Prepare callbacks cannot be registered for nested transactions because TRAN does not use a prepare-commit protocol for nested subtransactions. TRAN does this to expedite resolving subtransactions on whose outcome other transactions depend.

The prepare callback is the only one of these three types of callbacks that runs within the scope of a transaction. Reaching the prepare phase of the commit process indicates that a transaction's coordinator has requested that all participants indicate if they can commit the transaction. The prepare callback is executed while still within the scope of the transaction, and therefore is the only callback whose execution can actually affect the outcome its associated transaction. Statements in a function registered as a prepare callback can perform additional processing on behalf of that transaction and can even abort the transaction, if necessary.

In the same way the statements placed inside the Tran-C **onAbort** and **onCommit** clauses can be executed only after the outcome of a transaction is known, functions registered as commit and abort callbacks can be executed only at this time. The callbacks are also executed outside the scope of any transaction. Callback functions can use the Tran-C **getCompletedTid** function to determine the identifier of the transaction on whose behalf they are running.

As mentioned earlier, a nested transaction's **onCommit** clause is executed when that transaction commits, even though that transaction can later abort because its ancestor aborts. Callbacks associated with nested transactions are executed when the entire transaction family, of which that transaction is a part, completes. Table 5 illustrates the interaction between the **onCommit** clause, **onAbort** clause, commit callbacks, and abort callbacks for nested transactions.

Table 5. Transaction clause and callback interactions in nested transactions

Action	**Clauses and Callbacks Executed**			
	onCommit Clause	**onAbort Clause**	**Commit Callback**	**Abort Callback**
Nested transaction aborts		When that transaction aborts		Immediately

Table 5. Transaction clause and callback interactions in nested transactions (continued)

Action	Clauses and Callbacks Executed			
	onCommit Clause	**onAbort Clause**	**Commit Callback**	**Abort Callback**
Nested transaction commits, but some ancestor aborts	When transaction commits	Never		When that ancestor aborts
Nested transaction commits, and transaction family commits	When transaction commits		When top-level ancestor commits	

Transactional resource allocation

When a Tran-C transaction aborts, the thread under which it ran transfers control to the transaction's **onAbort** clause. If the transaction is aborted by its parent thread, then this transfer of control occurs immediately. If the transaction is aborted by another thread, then control is transferred to the **onAbort** clause the next time a Tran-C construct is executed by a thread under that the transaction was executing.

This uncertainty about exactly when the transfer of control to a transaction statement's **onAbort** clause occurs can cause problems if the transaction has allocated resources. These resources cannot be freed until control transfers to the **onAbort** clause. To automate the de-allocation of resources acquired during a transaction, Tran-C provides the **tranMemAlloc** and **tranMemFree** functions, and a number of different functions that initialize, lock, unlock, and discard mutexes. A primary reason to use these functions is that they automatically deallocate resources if the transaction in which they were allocated aborts. The memory allocation and mutex functions are explained in the next two sections.

Transactional memory allocation

The Tran-C **tranMemAlloc** and **tranMemFree** functions associate allocated memory with the current transaction. When the **tranMemAlloc** function is used to allocate memory inside a transaction, this memory is automatically freed if the transaction within which it was allocated aborts. Tran-C does not automatically deallocate memory if the transaction commits; the application must free the transaction memory explicitly by calling the **tranMemFree** function.

The syntax of these functions follows:

```
void *tranMemAlloc(unsigned long size);
void  tranMemFree(void *addr);
```

When a nested transaction allocates memory and then commits with respect to its parent, the parent inherits any memory allocated during the nested transaction. If the parent aborts, the memory is freed. If the amount of memory requested by **tranMemAlloc** is not available, the containing transaction aborts with the abort reason MEMORY_EXHAUSTED_CODE.

The **tranMemAlloc** function stores information about the memory allocated to different transactions in the Tran-C runtime environment. Such memory must not be freed by using system functions that free memory, such as the standard C **free** function. Similarly, memory allocated using the standard C **malloc** function must not be freed by using **tranMemFree**, because the runtime environment will not be able to locate the internal information required to manage such memory. The results of attempting to mix Tran-C and system allocation and de-allocation functions are unpredictable. Because **tranMemAlloc** affects memory associated with the current transaction, it is illegal to call it outside the scope of a transaction, but an application can call **tranMemFree** anytime.

Transactional mutexes

Tran-C provides *transactional mutexes* and a collection of functions for initializing, terminating, locking, and unlocking this type of mutex. A mutex is a data type that represents sections of code or allocated memory that only one thread can access, restricting all others. A mutex locked within a transaction by using one of the Tran-C mutex functions is unlocked automatically when the transaction ends (commits or aborts).

Mutexes are intended for short-term mutual exclusion of internal data structures (as opposed to recoverable data) within an application. Mutexes are thread dependent, but a mutex is associated with the transaction within whose context it is locked. This allows Tran-C to unlock the mutex when execution exits the transaction's scope. Mutexes are different from locks (see "Locking data in Tran-C applications" on page 89), which are intended for long-term (relative to mutexes) isolation of recoverable data to prevent multiple transactions from accessing inconsistent data. Locks have a rich set of locking modes, providing more than simple mutual exclusion. When using a lock, multiple threads operating within the context of one transaction can interfere with each other because locks are not thread dependent. When using a mutex within one thread, a nested transaction can interfere with an ancestor because mutexes are not transaction dependent.

The following functions create and manipulate mutexes:

```
void tranMutexInit(tranMutex_t *mutexP);
void tranMutexInitOnce(tranMutex_t *mutexP);
void tranMutexLock(tranMutex_t *mutexP);
```

```
int  tranMutexTryLock(tranMutex_t *mutexP);
void tranMutexUnlock(tranMutex_t *mutexP);
void tranMutexTerminate(tranMutex_t *mutexP);
```

Initialize a given mutex variable with **tranMutexInit** when the application is guaranteed to execute the initialization call only once. If the application can possibly try to initialize a mutex variable more than once due to the placement of the initialization call, then use the Tran-C mutex function **tranMutexInitOnce**. It initializes the mutex variable once regardless of how many times it is called. This action prevents an already locked mutex from being reinitialized and allowing two threads to interfere with each other by modifying the same data simultaneously or executing a critical section of code at the same time. When using the function **tranMutexInitOnce**, assign the mutex variable to the constant `TRAN_MUTEX_INITIALIZER` before calling this function. When an application no longer needs a mutex, it should terminate the mutex with **tranMutexTerminate** because mutexes consume system resources.

The Tran-C mutex functions associate mutexes with the current transaction. It is not permitted to call any transactional mutex functions (other than the transactional mutex initialization functions) outside the scope of a transaction. Like other Tran-C constructs associated with a specific transaction, transactional mutexes allocated within any kind of transaction construct cannot be used within the **onAbort** or **onCommit** clauses, or within upcalls, because these are executed outside the scope of any transaction.

The **tranMutexLock** function locks a mutex for a particular thread. This prevents other threads that try to lock the mutex from interfering with the locked data. If the mutex is already locked when this function is called, the function blocks until the mutex is available for locking. Since blocking is possibly inappropriate for the application, Tran-C provides the **tranMutexTryLock** function. If the mutex is not locked, then this function locks the mutex and returns immediately with a return value of `TRUE`. If it cannot lock the mutex, it does not block but returns immediately with a return value of `FALSE`.

The **tranMutexUnlock** function releases a locked mutex, allowing another thread to lock the mutex if necessary. Mutexes are automatically deallocated when the transaction that acquired them either commits or aborts.

Creating and maintaining transactional mutexes requires considerable time and resources; therefore, use them only when absolutely necessary. Use the system's underlying mutexes for an application unless the guarantees provided by transactional mutexes are necessary for the application's correct operation.

Using standard mutexes within Tran-C applications

Do not use standard mutexes, such as those provided by the Distributed Computing Environment (DCE) Threads package, when the mutexes are held across Tran-C function calls or control constructs. In Tran-C applications, the transfer of control on transaction abort can occur whenever a Tran-C function or construct is executed. If an abort occurs within a Tran-C application, any standard mutexes held by the application are not be released if this transfer of control occurs before they are explicitly released. Use the transactional mutex functions described in the previous section if mutexes must be held across Tran-C function calls or control statements, since they associate mutexes with the current transaction and automatically release those mutexes if the transaction aborts.

Standard mutexes can be used to isolate regions of memory within any standard C language code in a Tran-C application, as long as they are explicitly released before any Tran-C functions or constructs are encountered.

Creating asynchronous threads

The Tran-C **concThread** function creates an autonomous thread and is the threaded analogue to the **fork/exec** combination used at the process level in UNIX systems. Unlike the concurrency constructs, using **concThread** leaves the current thread of execution running instead of the current thread waiting for the spawned thread to terminate. The syntax of this function follows:

```
int concThread(void (*funcPtr)(void *), void *);
```

The **concThread** function takes two arguments. The first is the name of the function to be executed as a separate thread. Because it is executed as an asynchronous, autonomous thread outside the scope of any calling transaction, this function cannot return a value. The second is a pointer to any arguments required by that function. The value returned by the **concThread** function specifies whether the system successfully created the thread: `TRUE` if successful and `FALSE` if not. When returning `FALSE`, Tran-C guarantees the function referenced by the argument was not executed.

The new thread created by a **concThread** function is executed concurrently with the calling thread and vanishes when the function returns. Threads created using the **concThread** function are executed outside the scope of a transaction, but can themselves begin transactions.

Tran-C does not support using Tran-C constructs in threads that can be canceled. If a thread is canceled during the execution of a Tran-C construct or function, then the state of the transaction on whose behalf the call was made is undefined.

Maintenance and monitoring functions

Transactions that remain unprepared for long periods of time can consume resources that are otherwise normally allocated to active transactions. To minimize the number of such transactions, Tran-C provides two *watchdog* functions. These functions provide a mechanism for associating a time limit and an expiration callback with transactions. If the specified time limit is exceeded before the transaction commits or aborts, the callback is invoked.

Tran-C provides two different implementations of this mechanism: one associates a time limit with the current transaction, and one associates a time limit with any named transaction identified by a transaction ID.

The **watchTran** function registers a time limit and callback for the current transaction. The syntax follows:

```
unsigned int watchTran(void (*callback) (tran_tid_t, void *),
    void *callbackArg,
    unsigned int timeperiod,
    int disallowreset,
    int familyWatch)
```

The *timeperiod* parameter is the number of seconds the transaction can remain active before the callback is invoked. Supplying `0` (`zero`) cancels the watch. The first time this function is called to set a watch, it returns `0` (`zero`). Upon future invocations, this function returns an integer representing the number of seconds remaining on the current watch. If the *disallowreset* parameter is `TRUE` when a watch is initially set, subsequent attempts to reset the watch return `0` (`zero`), indicating the watch cannot be reset.

The **watchTran** function takes a number of arguments. The first is the callback to associate with the current transaction. The second argument is an argument to that callback, and if the callback requires multiple pieces of data, the second argument must be a pointer to a block of memory containing the necessary information. The time limit and callback can be canceled by calling **watchTran** with the same callback and argument, and a time period of zero. Resetting the timeout period resets the timeout counter to `0`; for example, if the current timeout period is 10 seconds with five seconds having passed, and you reset the timeout period to 20 seconds, then the callback is not delivered until 20 more seconds have elapsed.

The last parameter determines how the **watchTran** function behaves for a transaction that is part of a transaction family in which other transactions have the same callback and argument registered. If `TRUE`, the expiration of the timeout set for the current transaction does not cause the callback to be delivered if the same callback has been registered for other transactions in the same family.

The **watchTran** function associates a time limit and callback with the current transaction. Tran-C also provides the **watchNamedTran** function, which enables the application to register a time limit and expiration callback for any transaction, identified by TID. The syntax follows:

```
unsigned int watchNamedTran( tran_tid_t tid,
    void (*callback) (tran_tid_t, void *),
    void *callbackArg,
    unsigned int timePeriod,
    int disallowreset,
    int familyWatch)
```

Like the **watchTran** function, the **watchNamedTran** function returns an integer which is the number of seconds remaining from a previous setting when the watch on a transaction is reset. The first time it is called to set a particular watch, it returns `0` (`zero`). If the *disallowreset* parameter is set to `TRUE` when a watch is initially set, subsequent attempts to reset or cancel that watch will return `0` (`zero`), indicating the watch cannot be changed.

The arguments to this function are the same as those to the **watchTran** function, except that the first parameter to the **watchNamedTran** is the TID of the transaction for which the time limit and callback are being registered.

If the callbacks registered by using the **watchTran** or **watchNamedTran** function are delivered, they are executed outside the scope of any transaction. Once these callbacks have been delivered, they are automatically canceled—they are not delivered again unless explicitly reset.

Locking data in Tran-C applications

Locking provides a way to manage access by multiple transactions to shared data resources within a given application. Tran-C provides lock support for protecting resources that do not implement locking. The locking functions supported by Tran-C are based on the Lock Service (LOCK) of the Encina Toolkit. Other locking mechanisms can also be used in a Tran-C application if those external locking mechanisms can be interfaced to C programs. Tran-C also provides a mutex-locking mechanism; “Transactional mutexes” on page 85 discusses the differences between Tran-C locks and mutexes.

Tran-C provides a layer on top of the Lock Service for managing locks within the context of transactions. If a server is ephemeral (that is, does not use a recovery service), Tran-C automatically releases any locks held by a transaction when the transaction completes. If the application uses the Encina Toolkit Recovery Service, the appropriate Lock Service routines must be registered as Recovery Service upcalls so that locks are dropped at the right times.

This section provides background information about locking, the locking model provided by the Toolkit's LOCK module, and the Tran-C locking routines. This section also explains deadlocks and how to resolve them in Tran-C. An example from the merchandise server is included at the end of this section.

General information about locking

The Tran-C locking mechanisms use the features provided by the Lock Service. This service helps guarantee the serialization of transactions by enabling transactions to lock resources before accessing or modifying them. Locking is a form of *concurrency control* that insures that transactions cannot access the same resources at the same time unless concurrent access by multiple transactions is explicitly permitted. Resource locks are obtained on behalf of transactions in different *lock modes*, which determine the degree of concurrent access permitted to the locked data. A lock request made by a transaction is denied if another transaction holds the lock in a mode that does not permit concurrent access. For example, a transaction cannot obtain a read lock on data for which another transaction holds a write lock, because write locks do not permit concurrent access.

A primary goal of a locking mechanism is to guarantee the maximum amount of concurrency possible between nonconflicting transactions requiring access to shared data. This means that only the minimum amount of data required by a transaction is locked by the application.

Another important goal of a locking mechanism is to provide a way to logically isolate locks from the specific type and format of the data being locked. The Toolkit's LOCK module provides a logical locking mechanism that isolates locks from the format of the resources that are being locked. This logical locking mechanism associates *lock names* with various resources—a lock obtained by an application is associated with the lock name rather than with the resource itself. Because logical locks are independent of the format of the underlying data, the application developer determines the granularity of the data held by logical locks.

Tran-C is implemented with the assumption that locking is used correctly. If locking is used incorrectly, Tran-C generates a fatal error. For example, if an application drops a lock it does not own, Tran-C terminates the application after issuing an error message.

Note: The Tran-C locking interface does not include a call to initialize the Lock Service. This action must be performed during initialization of the Recovery Service. (Refer to the *Encina Toolkit Programming Guide* for information about initialization and the interface between the Recovery Service and the Lock Service.)

Lock modes

The Tran-C lock interface supports five different lock modes: read, write, intention read, intention write, and upgrade. Predefined values of type **lock_mode_t** are provided for these lock modes. The following are descriptions of the lock modes:

Read
Other transactions can read the locked data, but none can modify the data while a read lock is held. Holding this type of lock indicates that a transaction depends on the consistent state of the locked data. A read lock is obtained by requesting a lock of type `LOCK_MODE_READ`, using the functions explained in "Tran-C locking functions" on page 93.

Write
No other transaction can simultaneously access the locked data while a write lock is held. A write lock is obtained by requesting a lock of type `LOCK_MODE_WRITE`, using the functions explained in "Tran-C locking functions" on page 93.

Intention
Intention locks are used when hierarchical resources are locked and are typically obtained on the root or ancestors of a desired resource. They provide a way to minimize potential conflicts on lower-level resources without needlessly using more- coarsely-grained locks. Tran-C supports both *intention read* and *intention write* locks. An intention lock is obtained by requesting a lock of type `LOCK_MODE_INTENT_READ` or `LOCK_MODE_INTENT_WRITE`, using the locking functions explained in "Tran-C locking functions" on page 93. Obtaining intention locks on higher-level resources provides an easy way to indicate that specific transactions are using a subset of the locked data without requiring that an entire database be exclusively locked. This type of lock is typically used when attempting to update a specified field in a specified record of a large database that is updated constantly. An intention write lock is first obtained on the database itself, and another is obtained on the specified database record. A conventional write lock can then be obtained on the specific field. Another transaction wishing to update another record in the database, or even another field in the specified record, can infer that the database is in use but can still proceed to lock the data it requires.

Upgrade
An upgrade lock is a type of read lock that conflicts with other upgrade locks held on behalf of other transactions. If a transaction needs to read data that it can subsequently need to write, the transaction should obtain an upgrade lock rather than a simple read lock on that data. If the lock is obtained successfully, it indicates that no other upgrade lock is held on that data and prevents any new upgrade locks from being obtained on that data. This type of lock can be used to avoid potential deadlocks. An

upgrade lock is obtained by requesting a lock of type `LOCK_MODE_UPGRADE`, using the functions explained in "Tran-C locking functions" on page 93.

The lock modes are used when lock modes are set explicitly. Tran-C provides an additional predefined value, `LOCK_MODE_NONE`, that is useful for indicating there is no lock held when writing application-specific lock functions. This value cannot be passed as an argument to any Tran-C or Lock Service functions that take **lock_mode_t** values as parameters.

Table 6 shows the compatibility between the lock modes supported by the Tran-C lock interface. An *X* indicates that locks do not conflict, which means two different entities can obtain the same lock in two different, nonconflicting modes without affecting each other adversely.

Table 6. Table of nonconflicting lock modes

	IR	**R**	**U**	**IW**	**W**
Intention Read (IR)	X	X	X	X	
Read (R)	X	X	X		
Upgrade (U)	X	X			
Intention Write (IW)	X			X	
Write (W)					

Lock names and name spaces

A lock name is used to associate a lock with a particular resource. Lock names have no predefined data types—any C variable can be used as a lock name. Character strings or integers (representing an index into data) are the most common items used for lock names. Logical lock names are variable length and are specified as pointer-length pairs. The lock name must be greater than `0` (`zero`) and less than or equal to 512 bytes in size.

All lock names exist within a specified lock namespace. The locking functions require both a lock name and namespace as arguments, specifying the namespace in which the lock name exists. A lock conflicts with another lock of the same name only if both locks share the same lock namespace. See "Tran-C locking functions" on page 93.

Lock names are application dependent. Most applications can use a single lock namespace for lock names. For these applications, Tran-C provides a default lock namespace with the name `lockSpacePrimary`. The `lockSpacePrimary` constant (a pointer to type **lock_space_t**) is defined as follows:

```
extern lock_space_t *lockSpacePrimary;
```

To specify that a lock name exists with the default namespace, the value `lockSpacePrimary` can be passed to the locking functions.

For large applications involving multiple independent modules, the use of different lock namespaces can be necessary to make choosing names within particular modules easier for developers. New lock namespaces can be created with the **createNameSpace** function. The **createNameSpace** function has the following syntax:

```
void createNameSpace(
    lock_space_t *spaceP);
```

The function takes a pointer to an object of type **lock_space_t** as its only argument. After the function returns, the object identifies a newly created namespace.

Tran-C locking functions

The Tran-C locking functions are used to obtain locks, change the mode of a lock, and release locks. The functions used to obtain locks are either *blocking* or *nonblocking* functions. A blocking lock function is one that does not return control to the calling procedure until the lock is obtained. A nonblocking lock function attempts to obtain a lock and, if unable to do so, immediately returns an error code indicating that the attempt to lock the data failed. Nonblocking lock functions return the value 1 (`TRUE`) if the lock is obtained successfully. Tran-C provides both blocking and nonblocking versions of all of its locking functions. Nonblocking lock functions have names prefixed with **try** (for example, **tryLock**).

The type of lock function used in an application depends on the requirements of that application. Applications typically use blocking locks when a particular thread of control requires access to data that it must have in order to proceed.

Note: The functions to obtain and release locks do so on behalf of the transaction within whose scope they are called. Do not call these functions outside the scope of a transaction.

The function used to obtain a lock determines the duration of the lock. The **lock** and **tryLock** functions can be used to obtain a *transaction duration* lock. A transaction duration lock is one that is held until the transaction holding the lock aborts, the transaction family commits, or the lock is released explicitly. Tran-C provides blocking and nonblocking functions for obtaining this type of lock:

```
void lock(
    lock_mode_t mode,
    void *lockNameP,
    unsigned int nameLength,
    lock_space_t *spaceP);
unsigned int tryLock(
```

```
    lock_mode_t mode,
    void *lockNameP,
    unsigned int nameLength,
    lock_space_t *spaceP);
```

Both functions take four arguments: the mode of lock the application is trying to obtain; the name to be associated with the lock; the length of the lock name; and the lock name space in which that lock must be created.

The **instantLock** and **tryInstantLock** functions can be used to obtain an *instant duration* lock. An instant duration lock is one that is granted but not actually held, which is equivalent to obtaining and immediately dropping a standard lock. If the lock is granted successfully, it indicates that no unrelated transactions have the data locked in a conflicting mode. This type of lock helps maximize concurrency in Tran-C applications by providing a way to test lock availability without obtaining and dropping each required lock explicitly. The syntax of these functions is the same as the equivalent functions for obtaining transaction duration locks:

```
void instantLock(
    lock_mode_t mode,
    void *lockNameP,
    unsigned int nameLength,
    lock_space_t *spaceP);
unsigned int tryInstantLock(
    lock_mode_t mode,
    void *lockNameP,
    unsigned int nameLength,
    lock_space_t *spaceP);
```

Applications that must lock a large number of resources can test the availability of locks first using an instant duration lock function rather than using a nonblocking lock function and checking its return code. Instant duration locks allow the application to test locks without blocking other transactions from obtaining conflicting locks on the same data until all required resources can be locked simultaneously. They are most often used for implementing schemes such as key-range locking.

Once an application has obtained a specified lock, it may be necessary to change the mode of that lock. To change the mode of a lock that is already held, Tran-C provides the **changeMode** function. The syntax of this function is the following:

```
void changeMode(
    lock_mode_t oldMode,
    lock_mode_t newMode,
    void *lockNameP,
    unsigned int nameLength,
    lock_space_t *spaceP);
```

The arguments to this function are very similar to those used to obtain the original lock except that this function requires passing both the old (current) mode of the lock and the new lock mode to which the application wishes to change. Downgrading the mode of a lock that is currently held should be done with extreme caution because it affects the ability of other threads to access the locked data. For example, downgrading a write lock to a read lock can enable other threads to access and act upon modified data before those changes are permanent (in other words, before the transaction that modified the data commits). If the transaction subsequently aborts but other transactions and threads have already done work that is based on the modified data, inconsistencies can be introduced into an application's data.

The **changeMode** function blocks if the application tries to change the lock to a mode that is incompatible with another lock held on the same data. Though Tran-C does not offer a nonblocking interface for changing a lock's mode, the Lock Service does support such a routine.

When a transaction obtains a transaction duration lock, that lock is ordinarily held until the parent transaction either commits or aborts. The **unlock** function can be used to drop a transaction duration lock explicitly. This function is useful in a long-running transaction that requires obtaining multiple locks but does not need to retain all of those locks for the duration of the transaction. Like downgrading the modes of currently held locks, dropping locks should be done with extreme caution because it can enable other threads and transactions to access modified data that has not been made permanent.

The syntax of the **unlock** function is the following:

```
void unlock(
    lock_mode_t mode,
    void *lockNameP,
    unsigned int nameLength,
    lock_space_t *spaceP);
```

Like the other locking functions, this call takes four arguments: the mode of the currently held lock to be released; the name of the lock to be released; the length of the lock name; and the lock name space in which the lock was created.

To simplify sharing locks and tracing relationships between nested transactions, all nested transactions having a common ancestor are considered to belong to the same *transaction family*. A nested transaction can obtain a lock held by any of its parent transactions. Because all members of a transaction family commit or abort together, they also drop their locks simultaneously.

Detecting deadlocks

Access conflicts frequently arise when multiple transactions require exclusive access to the same resources before proceeding. Each transaction begins to lock the resources it requires, but at some point encounters a resource that is locked by one of the other transactions. For example, transaction A requires access to locked data and holds locks on objects that are required by transaction B; however transaction B holds locks on the objects required by transaction A. At this point, none of the transactions can proceed until it obtains the remaining locks it requires. Without some external means of detecting and resolving these types of situations, none of the transactions will ever be able to proceed.

This type of situation is called a *deadlock*. To automatically resolve deadlocks in Tran-C applications, Tran-C provides the **deadlockDetect** function, which has the following syntax:

```
void deadlockDetect(
    unsigned int seconds);
```

This function takes a single argument, which is the maximum time (in seconds) that a transaction waiting on a lock request can block. After a transaction waits for a lock the amount of time specified, the Tran-C runtime system checks whether the transaction is involved in a deadlock. Applications typically call this function before making any lock requests because it only affects transactions which subsequently wait for locks. When the system detects a deadlock, it aborts a transaction that will resolve the deadlock and whose family holds the fewest locks. The abort code `DEADLOCK_DETECTED_CODE` is used as the reason for aborting one of the transactions involved in the deadlock.

Locking example

The example shown in Figure 30 on page 97 illustrates that applications usually do not drop locks explicitly because the locks held by a transaction are automatically dropped when the transaction commits or aborts. The sample program uses the indexes into the array as lock names, which exist in the default lock name space, `lockSpacePrimary`, provided by Tran-C.

```
/* Merchandise_QueryItem - returns the quantity of an item in
 * stock, given the stock number.  Aborts the calling transaction
 * if stockNum is out of bounds. */
void merchandise_QueryItem(trpcHandle, stockNum, quantityP)
   trpc_handle_t trpcHandle;
   ndr_long_int stockNum;
   ndr_long_int *quantityP;
{
   inFunction("merchandise_QueryItem");
   /* If stockNum is out of bounds abort the transaction */
   if (stockNum <= 0 || stockNum >= REC_ARRAY_SIZE)
       abortWithCode(MERCHANDISE_BAD_STOCK_NUM);
   /* Lock the merchandise with number stockNum */
   lock(LOCK_MODE_READ, &stockNum, sizeof(int), lockSpacePrimary);
   /* Get the quantity */
   *quantityP = recArray_Read(stockNum);
   /* Set timer to abort transaction if client transaction does not
    * finish within MERCHANDISE_TRANSACTION_TIME_OUT seconds. */
   watchTran(TimeOutTran, NULL, MERCHANDISE_TRANSACTION_TIME_OUT,
             FALSE, TRUE);
}
```

Figure 30. Example of function using data locking

Lazy transactions

A *lazy transaction* is a top-level transaction that does not guarantee at the time it commits that updates to recoverable work are permanent; such a transaction is said to commit *lazily*. This means that if a lazy transaction performs updates to recoverable resources, those updates can be undone (for example, due to a system interruption) even after the transaction commits. If any nonlazy transaction commits after a lazy transaction commits, the lazy transaction's updates become permanent. The use of lazy transactions can improve performance, because the commit does not need to wait for the record of the transaction to be written to the log.

Lazy transactions can be used in server applications to access transactionally consistent resources on behalf of nontransactional client applications. Lazy transactions guarantee that nontransactional access is serialized with any concurrent transactional access.

The **lazyTran** construct, shown in Figure 31 on page 98, can be used to establish a lazy transaction. The **lazyTran** construct has the same form as a **transaction** construct. The **onCommit** clause is optional, and the **onAbort** clause is mandatory. The **onCommit** clause, however, has the same semantics as it does with subtransactions; that is, when the lazy transaction commits, the **onCommit** statements are executed, and even if the lazy transaction is later

aborted, the **onAbort** statements are not executed. When the lazy transaction aborts within the scope of the **lazyTran** construct, the **onAbort** statements are executed.

```
lazyTran {
    statements
}
onCommit  {
    statements
}
onAbort{
    statements
}
```

Figure 31. Syntax of the lazyTran construct

The **lazyTran** construct can also have a **suspend** clause, as shown in Figure 32.

```
tran_tid_t *id;
lazyTran {
    statements
}
suspend(id) {
    statements
}
onAbort{
    statements
}
```

Figure 32. Syntax of the lazyTran construct with suspend clause

Tran-C also provides the **markTranLazy** function, which can be used to mark the current transaction as lazy. This function must be called within the scope of a top-level transaction (that is, a transaction begun with the **topLevel** or **transaction** construct); a fatal error occurs if it is called within the scope of a subtransaction. The syntax of this function follows:

```
void markTranLazy();
```

The **markTranLazy** function eliminates the need to know that a transaction must be lazy at the time that transaction is begun; the decision to commit lazily can be made when the transaction is going to commit.

If a lazy transaction occurs within the dynamic scope of another transaction, then the system executes a nested top-level transaction (see "Nested and top-level transactions" on page 53). The lazy attribute applies to a transaction family, so all nested transactions created by members of the lazy transaction family are themselves lazy. The lazy attribute applies to a local-only

transaction family, and if any member of a lazy transaction family makes a transactional remote procedure call (thus becoming distributed), then the transaction family ceases to be lazy.

Tran-C implements lazy transactions by using support from the Lock Service and the Recovery Service. Therefore, the **lazyTran** construct and the **markTranLazy** function are available only in the server component of Tran-C. If an application attempts to create a lazy transaction, and the application has not initialized the Recovery Service, then the system generates a fatal runtime error indicating that the use of a lazy transaction is not allowed.

Saving and restoring Tran-C context

It is frequently useful to execute a callback or upcall outside the scope of the current transaction. Applications can save the current Tran-C context, generate a short-lived Tran-C environment, execute the desired functions, and then restore the saved context and continue execution. The short-lived Tran-C environment is not a new transaction or subtransaction, and it is outside the scope of any transaction. Execution in the new environment is completely unaffected by transaction aborts. The new environment is a newly initialized Tran-C environment that supports the execution of Tran-C constructs.

To save an existing context, Tran-C provides the **tc_SaveTranContext** function. The syntax of this function is the following:

```
void tc_SaveTranContext(OUT void **contextPP);
```

The **tc_SaveTranContext** function saves the current context of the thread that executes it, generates a new Tran-C context for that thread, and returns the saved context. After this call returns, the calling thread can use any Tran-C construct, which are executed outside the scope of any transactions within which the calling thread was operating.

To subsequently restore the context saved by a call to the **tc_SaveTranContext** function, Tran-C provides the **tc_RestoreTranContext** function. The syntax of this function is the following:

```
void tc_RestoreTranContext(IN void *contextP);
```

The **tc_RestoreTranContext** function must be passed the context returned by a matching **tc_SaveTranContext** function in order to end the current transaction-independent context and restore the previous transaction scope. After the saved context is restored, the thread again is executed within the transactional scope supplied.

There are a few rules the application must adhere to when using these routines:

- Calling **tc_SaveTranContext** repeatedly works, but the application must call **tc_RestoreTranContext** with the saved contexts in the opposite order from which they were obtained. That is, contexts must be saved and restored in a stack discipline. If the application does not follow this discipline, the results are undefined.
- The application must never restore a context more than once.
- When a context is restored, there must be no pending transactions in the executing thread (that is, the context ending, not the context being restored).

Overview of external function compatibility

Transactional applications written by using Tran-C operate in the threaded environment used with the Encina Toolkit. Some operating systems provide no native support for threads; therefore, many standard C and UNIX can create side effects when called from within Tran-C applications. Two primary concerns are involved:

- Whether the functions use static data structures that can be modified simultaneously by multiple threads
- Whether the functions operate at the process level, providing control of or access to internals that affect the execution of entire processes

Functions that use internal static data structures pose serialization problems if called from multiple threads. Since functions of this type either read or write a static area, it is impossible to determine whether the data in that location is still accurate with respect to a single process since you cannot know when various processes are swapped in and out. For example, the UNIX **errno** variable holds the system error code for the last error encountered in a static area. In applications, a common response to receiving an error condition is to read, interpret, and display this error number. However, if another error occurs in another thread immediately after the first error, it is possible that the application running in the first thread can display information about the second thread's error condition.

Functions that operate at the process level introduce several potential problems:

- Many C and UNIX I/O functions block further activity of the issuing process until that I/O is completed. This is incorrect for the threaded model of execution provided by Tran-C, because other threads should be able to continue execution while the thread that issued the read or write call should be blocked until its requested I/O can complete.
- Functions that cause the calling process to exit if the function does not succeed do not work correctly in a threaded environment. As with

process-oriented I/O functions, only a specified thread should be forced to exit under most conditions—not the entire process (and therefore all threads running within that process).

- Functions that affect the priority of processes should be used with caution. Using these functions to change the priority of a process can indirectly affect the priorities of all threads running within that process. Some systems provide special functions to change the scheduling priority of a single thread within a process.

The DCE Threads package provides thread-safe versions of many UNIX and ANSI-C functions that can then be used in a threaded environment.

Calling Encina Toolkit functions from Tran-C

In general, most of the functions provided by any module of the Encina Toolkit can be called from within a Tran-C application. However, directly calling Toolkit functions involving resources that are ordinarily managed by the Tran-C runtime system can be counter productive. Examples of these resources are the transaction management functions provided by the Transaction Service (TRAN) and the locking functions provided by the Lock Service (LOCK). When using these functions within a Tran-C application, consider the following issues:

- Tran-C internally handles the creation and scheduling of the threads necessary for most program activity. The Tran-C **concurrent**, **cofor**, **subThread**, and **concThread** constructs prevent the user from having to issue the low-level thread creation and management calls associated with concurrent activities. Tran-C also tracks the threads associated with a given transaction automatically, terminating them as necessary if the encompassing transaction aborts. Using external thread calls, such as DCE thread calls, from within a Tran-C application requires that the application devote some of its time to managing and tracking the activity of concurrent threads associated with each transaction. When Tran-C creates a thread, it automatically allocates the data structures used internally to hold information about that thread. These underlying structures are not created or maintained by the external (non-Tran-C) threading functions.
- Control constructs such as the Tran-C **transaction** construct and its associated **onAbort** and **onCommit** clauses reduce the complexity of transactional application development by removing the need to associate callbacks with the general outcome of specific transactions. Instead, the statements inside those clauses provide the function of the callbacks for the completion of associated transactions. Similarly, the Tran-C runtime environment manages much of the low-level data necessary for communications with remote processes by automatically providing mandatory remote procedure call (RPC) parameters such as transaction

identifiers. Other Tran-C constructs, such as the **subTran** construct, handle the creation of nested transactions and their associated data structures automatically.

- Tran-C provides a layer on top of the Lock Service for managing locks within the context of transactions. If the application is ephemeral (that is, does not use a recovery service), then Tran-C automatically releases any locks held by a transaction when the transaction completes. If the application uses the Recovery Service, developers are responsible for releasing locks at appropriate moments. They typically can do this by registering Lock Service routines as Recovery Service upcalls when they initialize the Recovery Service.
- Internally, Tran-C uses the Thread-to-Tid Mapping Service (ThreadTid) component of the Encina Toolkit. Because of this, do not call the **threadTid_End** function when it affects the current thread that is created by Tran-C. If the application explicitly begins its own threads using the ThreadTid component, then it can use **threadTid_End** to end those threads, but the application must not affect a Tran-C maintained thread. It is, however, allowed at any time to query the transaction associated with the current thread by using the **threadTid_Lookup** function.
- Encina Toolkit Transaction Service functions that request information about the outcome of a transaction are not meaningful when called within a Tran-C transaction construct. For example, the **tran_ForceHeuristicOutcome** function is not meaningful within a Tran-C transaction construct because the outcome of the transaction is not yet known.

The Tran-C runtime environment automatically manages a large amount of lower-level information that must be explicitly maintained by each application if equivalent Tran-C functions are not used. This rule does not apply to lower-level functions that provide thread-safe versions of standard ANSI C and UNIX functions, as discussed in the next two sections.

Another way of determining the Tran-C and Encina Toolkit functions that can safely interoperate is to consider what Tran-C does to establish transaction contexts (see the example **transaction** construct in Box 1 in Figure 33 on page 103). At the beginning of a **transaction** (or similar) construct, Tran-C begins exception, transaction, and ThreadTid scopes. These are all terminated at the end of the construct's main body. In Figure 33 on page 103, Box 2 shows the pseudocode expansion of the example **transaction** construct shown in Box 1 of the same figure. If the application calls functions such as **tran_End** or **threadTid_End** within the scope of a transaction managed by Tran-C, the application undermines the ability of Tran-C to keep track of these entities. However, the application can explicitly establish and terminate nested exception, transaction, and ThreadTid scopes within a Tran-C scope.

Box 1: Example transaction construct

```
transaction {

        <transaction code>

}onAbort  {

        <onAbort code>

}
```

Box 2: Transaction construct pseudo-expansion

```
TRY {
   tran_Begin(...);
   threadTid_Begin (...);

      <transaction code>

   threadTid_End ();
   tran_End ();
}Catch(TC_ABORT_EXEPTION) {

      <onAbort code>

}
ENDTRY
```

Figure 33. Example transaction construct and its pseudocode expansion

Figure 34 shows an example of explicitly established and terminated transaction and ThreadTid scopes nested within a Tran-C scope.

```
transaction {
    ...
    tran_Begin(...);
    threadTid_Begin(...);
    ...
    threadTid_End();
    tran_End();
    ...
} onAbort  {
    <onAbort code>
}
```

Figure 34. Explicit transaction and ThreadTid scopes

Tran-C and TX interaction

The X/Open TX interface is a proposed standard that enables application programs to call a transaction manager in order to open and close resource managers and resolve transactions. The Encina TX Interface implements the preliminary specification from X/Open as specified in *Distributed Transaction Processing*. The Encina TX interface is documented in "Chapter 7. X/Open TX interface for Encina" on page 117.

Because of possible interaction problems, TX functions and Tran-C constructs should not be used in the same application process. It is safe, however, to use both Tran-C and TX in a single transaction when more than one application

process is involved. That is, a client written in Tran-C can make a transactional remote procedure call (TRPC) to a server written in TX, or a client written in TX can make a TRPC to a server written in Tran-C. These situations are likely to occur when modules from different sources are used in an application.

For example, Figure 35 shows a COBOL client using TX interacting with a Tran-C server. The COBOL client initiates the transaction and calls the **merchandise_QueryItem** remote procedure. The Tran-C server can participate in the transaction by aborting the transaction.

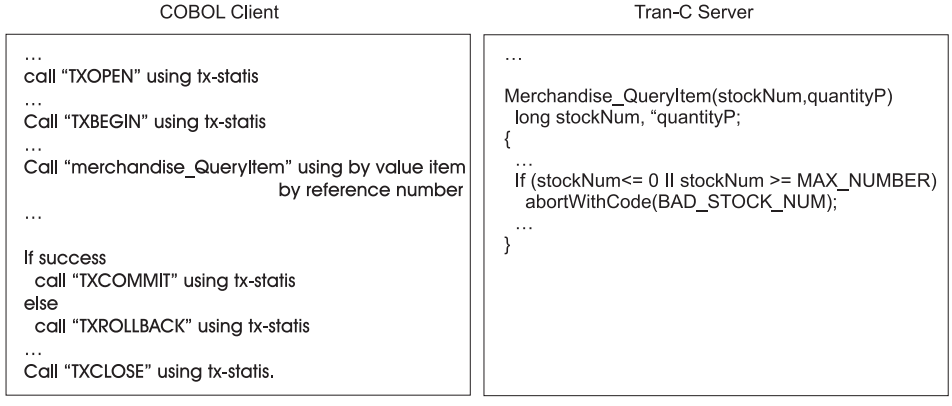

Figure 35. Pseudocode using TX and Tran-C

Interaction of Tran-C and PPC Services

The Peer-To-Peer Communications Services (PPC) enables Encina products to interoperate with other transaction processing systems such as IBM mainframes that support the LU6.2 protocol. Because IBM mainframes currently store a significant portion of the world’s databases, the ability to access and manipulate data stored on these machines is important. The standard communications method for accessing IBM hosts is the IBM System Network Architecture (SNA) LU6.2 protocol.

In addition to providing an interface that enables applications using Encina software to communicate with any IBM compatible mainframe over a SNA network, applications can use LU6.2 primitives over non-SNA networks, such as TCP/IP. The PPC Services hide the details of underlying protocols and systems. For more information, see the *Encina PPC Services Programming Guide*. The PPC Services products are separate from Tran-C; contact your sales representative if you are interested in their functionality.

When PPC Services are used with Tran-C, there are restrictions on manipulating synclevel syncpoint conversations. Figure 36 on page 105 and

Figure 37 illustrate this with pseudocode. The first is an example of a client application that uses the **transaction** construct to initiate a transaction.

```
transaction {
    cminit
    cmssl(CM_SYNC_POINT)
    cmallc  /* conversation allocated */
    . . .
    cm*
    . . .
    /* All synclevel syncpoint conversations in state:
     * deferDeallocate or syncpointDeallocate.
     */
}
```

Figure 36. Pseudocode using PPC Services via Tran-C

Before exiting the scope of the **transaction** construct, all synclevel syncpoint conversations must be in the CM_DEFER_DEALLOCATE_STATE or CM_SYNC_POINT_DEALLOCATE_STATE state. The application must not issue CPI-RR commands within the scope of a transaction construct, including any threads executing on behalf of the transaction at a server. If CPI-RR commands are issued in this way, the results are undefined.

The second pseudocode example, Figure 37, shows a fragment from a TRPC manager function in a server application that interacts with the PPC Services. The same restrictions apply to this code fragment that applied to the previous example.

```
cminit
cmssl(CM_SYNC_POINT)
cmallc  /* conversation allocated */
. . .
cm*
. . .
/* All synclevel syncpoint conversations in state:
 * deferDeallocate or syncpointDeallocate.
 */
return
```

Figure 37. Pseudocode using PPC Services via TRPC manager function

If you are using the PPC Distributed Program Link (DPL), all function calls must be made from within the context of a transaction, as shown in Figure 38 on page 106.

```
transaction{
     ...
     /*Make DPL call */
     Dynamic_Program_Link(conversationId,
                TRUE, "Query", "MY_PROGM", commArea,
                 COMM_AREA_SIZE, &datalength, &returnCode);
     /* Check return code to detect server errors */
     EXTRACT_DPL_ERROR(conversationId, abendCode,
                *condCodeP, *returnCode);
     ...
}OnCommit{
     /*Do commit processing here */
}OnAbort{
     /*Do abort processing here */
}
```

Figure 38. Using DPL within a transaction

The following general interoperability issues apply to using the PPC Services via Tran-C:

- Any number of conversations can be active, concurrently or otherwise, within the scope of the transaction.
- The application can issue TRPCs at any time while conversations are active.
- Synclevel none and synclevel confirm conversations can exist independent of transactions. They can begin before or during a given transaction, and they can end during the transaction or after execution exits its scope.
- If any synclevel syncpoint conversations are in an unacceptable state when the application exits the scope of a transaction, the transaction aborts with an abort reason of `PPC: premature transaction end`.
- If the application explicitly aborts the transaction, all synclevel syncpoint conversations are deallocated with the DEALLOCATE_ABEND_SVC option. All of the application's peers will receive CM_DEALLOCATED_ABEND_SVC.

Using ANSI C and UNIX with Tran-C

Tran-C supports the ANSI C standards for standard I/O, string and character I/O, date, and time libraries. However, functions that use static data structures (such as many of those in the date and time libraries) must be used with caution in threaded applications, where multiple threads can write simultaneously to static, and therefore common, structures. In general, it is safe to use these functions within a Tran-C application as long as these functions are called only from within the same (single) thread.

When writing threaded applications, it is also important to remember that the primary advantage of using threads is the ability to share and exchange data between cooperating processes. However, sharing data between threads means

that whenever possible threaded applications must avoid using global or static variables unintended for sharing . Because the location of these variables is the same for all threads sharing an address space, it is impossible to determine which thread writes the value last.

Some implementations of certain UNIX library functions use static locations, for example, the time functions, random-number generation functions, many math functions, and any other functions, such as the **crypt** and **ctime** functions, that themselves use these library functions. Other UNIX functions, such as the **read** function and other input and output functions, block the parent process until they can complete. Blocking the parent process means that all of the threads running within that process are also blocked.

Concerns about threading issues and conflicting access to static data structures are not the only C language concerns in Tran-C applications. Unstructured C language constructs that directly manipulate the flow of control in an application should not be used in most Tran-C applications. For example, C language constructs such as **break**, **continue**, **goto**, and **return**, must never transfer control out of a transaction statement or similar Tran-C construct. Tran-C manages the flow of control automatically in a Tran-C application.

Debugging Tran-C applications

Tran-C supports the standard tracing mechanism used by all Encina Toolkit components, as documented in the *Encina Toolkit Programming Guide*, and it also provides the **tc_DumpState** function. This section also lists all the error messages generated by Tran-C. See "Transfer of control in transactions" on page 38 for a list of general warnings and some failure conditions for Encina components.

Dumping application state

The **tc_DumpState** function dumps the state of Tran-C in an application. Calling this function generates the following output for each pending transaction that was created by using Tran-C:

- Transaction identifier
- Whether it is nested or top-level
- Transaction state (running, aborted, or committed)

Tracing applications

The tracing facility enables you to follow the execution path of the application as it makes calls into Tran-C. Applications can enable and disable tracing, set the tracing level, and direct the tracing output for applications. For details see the *Encina Toolkit Programming Guide*.

The exported global variable *tc_traceMask* enables applications to turn tracing on and off and set the desired tracing level. There are no special events traced for Tran-C.

Tran-C error messages

Tran-C generates a fatal runtime error whenever an unrecoverable event occurs. The error is presented as an output message through the Encina Toolkit trace facility (see "Tracing applications" on page 107), and the system terminates the application.

This section lists the fatal error messages Tran-C generates. In all cases, correct these errors by fixing the erroneous code that caused them.

Client error messages

```
Transactional-C was not initialized.

 Transactional-C was already initialized.

Illegal use of the suspend clause.

The tran_tid_t pointer passed as an argument to the suspend clause
was NULL.

The resumeTran statement was used in an illegal context (e.g. in a
concurrent statement).

The tran_tid_t argument passed to the resumeTran statement does not
belong to a suspended transaction.

The subTran clause was used outside a concurrent or cofor statement.

The subThread clause was used outside a concurrent or cofor
statement.

The transaction statement was executed within a concurrent or cofor
statement.

The topLevel statement was executed within a concurrent or cofor
statement.

The catchAbort statement was executed within a concurrent or cofor
statement.

The abort call was made outside the scope of a transaction.

The concurrent or cofor statement was executed within a concurrent
or cofor statement.

The concThread statement was executed within a concurrent or cofor
statement.

An attempt was made to register a transaction service callback
outside the scope of a transaction.

The getCompletedTid function was called outside commit or abort
clause.
```

The getContainingTid function called outside the scope of a transaction.

The tranMutexTerminate function was called on an uninitialized tranMutex_t.

The tranMutexLock function was called on an uninitialized tranMutex_t.

The tranMutexUnlock function was called on an uninitialized, or unlocked, tranMutex_t.

Attempt to set a watch on TRAN_TID_NULL.

Call to tranMemAlloc made with either a size of zero or on behalf of a nonexistent transaction.

Call to tranMemFree with a NULL pointer or memory that was not allocated to the calling transaction.

Transactional RPC made on behalf of nonexistent transaction.

The abortReason function was called outside the scope of an onAbort clause.

The abortModuleName function was called outside the scope of an onAbort clause.

The abortFunctionName function was called outside the scope of an onAbort clause.

The lazyTran statement was executed before the recovery service was initialized.

An application initialization callback returned FALSE.

An application termination callback returned FALSE.

Internal Transactional-C error -- vital memory allocation failed.

Lock extension error messages

A call to obtain a lock was made outside the scope of a transaction.

A call to release a lock was made outside the scope of a transaction.

A call to change a lock mode was made outside the scope of a transaction.

Miscellaneous error messages

Internal Transactional-C error.

Warning messages

Tran-C produces warning messages when an uncaught exception is handled by the Tran-C runtime system. Normally an uncaught exception aborts the containing transaction, but if there is no transaction to abort, then the warning messages are displayed. Two events generate a warning message: an uncaught exception in a thread created using **subThread**, and an uncaught exception

during a transaction's prepare callback. The system generates two types of warning message: an uncaught address exception, and an uncaught value exception. This results in four cases, and in each the system includes the value of the exception in the warning message. The warning messages are as follows:

```
Unexpected address exception in prepare callback: value
Unexpected status exception in prepare callback: value
Unexpected address exception from subThread: value
Unexpected status exception from subThread: value
```

Administering servers

You can use the **tkadmin** utility for administrating servers you develop. This utility enables you to control the server with actions such as enabling tracing, allocating and organizing physical storage, and setting or modifying authentication requirements. To implement this control with your server, you must include the **admin.h** file and call the **ADMIN_INIT** function. See the *Encina Administration Guide Volume 1: Basic Administration* and *Encina Toolkit Programming Guide* for more details.

Chapter 6. Compiling Tran-C applications

This chapter describes the Encina header files and libraries used with Tran-C applications. For more details on compiling Encina applications, see *Writing Encina Applications* and the *Encina Monitor Programming Guide*.

Specifying Tran-C and Toolkit header files and libraries

All Tran-C applications must include a Tran-C header file. In addition, applications must include the appropriate Toolkit header files for any Toolkit modules used in the application. Many of the Toolkit modules are provided as libraries. When compiling a Toolkit application, you must load the libraries for the modules you are using in that application.

Header files for Tran-C applications

Header files are text files that specify structure, variable, and macro definitions used by C applications. The header files for the Encina Toolkit are organized hierarchically under the **include** subdirectory of the Encina installation directory. To find and use the header files required by the various Toolkit modules, you must specify that this directory be searched for header files.

All Tran-C applications must include a header file to define the data types and internal structures used in Tran-C applications and by the Tran-C runtime environment. Use the following list to determine which Tran-C header files are appropriate for your application:

- If your application uses the Tran-C server extensions, it must include the file **<tc/tc_server.h>**.
- If your application does *not* use the Tran-C server extensions, it must include the file **<tc/tc.h>**.
- If your application uses the Tran-C transactional remote procedure call (TRPC) communication functions, it must include the file **<tc/rpc/tc_trpc.h>**.

Library files for Tran-C applications

Library files are binary files that contain precompiled routines and functions used by C applications. The libraries for the Encina Toolkit and Tran-C are located in the **lib** subdirectory of the Encina installation directory. To find and include the libraries required by the various Toolkit modules, you must specify that this directory be searched for libraries.

The functions provided or required by various Toolkit modules are contained in appropriately named libraries. The format of the names of these libraries is

lib*name***.a**, where *name* is considered the name of the library. Use the following list to determine which libraries are appropriate for compiling your application:

- All applications must compile with the **libEncina.a** library, which contains the functions for the Encina Toolkit Executive. For example, the Telshop client and server applications both link with this library.
- Server applications must compile with the **libEncServer.a** library, which contains the functions for the Encina Toolkit Server Core for servers. For example, the Telshop server application links with this library.

Compiling your Tran-C application probably requires additional non-Encina libraries, such as the Distributed Computing Environment (DCE) library **libdce.a**. These additional libraries are operating-system dependent. See the *Writing Encina Applications* for more information.

Resolving compilation problems

This section provides information on resolving problems that can occur when you compile Tran-C applications. Compilation problems specific to Tran-C are name collisions and incorrect macro expansion.

Resolving name collisions during compilation

Both Tran-C and the Toolkit are designed to support the development of other systems and applications using the technology that those systems and applications provide. This type of design requires a means for resolving *name collisions* that occur during compilation. Name collisions occur when a function defined in the application has the same name as a function provided in the libraries of an enabling technology. Functions that have common names, such as **lock**, which is used in the Tran-C server extensions, are the most likely to cause name collisions.

To enable users to resolve name collisions, the Tran-C function names used in this document, such as **lock** and **tryLock**, are actually suggested names that are mapped in Tran-C to functions with names of the form **tc**_*function-name*. Internally, all of the actual Tran-C functions begin with the **tc** prefix to indicate that the function is a part of Tran-C—the remainder of the name preserves the purpose of the function. Thus, the Tran-C **lock** and **tryLock** functions are internally named **tc_lock** and **tc_tryLock**.

The mapping between suggested and internal names (that is, between names like **lock** and **tc_lock**) is done by C preprocessor `#define` statements. The `#define` statements that declare the suggested names for all of the functions used in Tran-C are defined in the header file **tc/tc.h**.

Name collisions are detected during compilation. Collisions between function names can be resolved by undefining the suggested name of the Tran-C

function that caused the collision. The defined mapping between the suggested name and the internal name can be removed by using the C preprocessor's `#undef` statement. The `#undef` statement appears immediately after the `#include <tc/tc.h>` statement in the application program. For example, a name collision between the Tran-C **lock** function and a function of the same name in an application can be eliminated with the following line:

```
#undef lock
```

To use the Tran-C **lock** function after it has been undefined, refer to it as **tc_lock**. The identifier **lock** is now available for other uses.

Resolving macro expansion problems

Many Tran-C constructs are implemented as macros. As a result, diagnostic messages can fail to pinpoint the cause of problems precisely. When debugging a Tran-C application that does not compile, check for the following items:

- Non-terminated or incorrectly terminated C constructs within an application can cause a correctly defined Tran-C construct to be expanded incorrectly. Check the syntax of your C code to make sure that it is correct when the compilation of a Tran-C application fails.
- Incorrectly terminated Tran-C constructs can cause the Tran-C construct to be expanded incorrectly. An example of this is a **transaction** construct with no **onAbort** clause.

Other syntax errors in Tran-C constructs can cause a construct to be expanded differently from what is expected, resulting in execution or logic errors rather than compilation failure. For example, a **transaction** construct in which the **onAbort** clause contains multiple statements that are not enclosed within braces, or not terminated by a trailing semicolon, compiles, but only the first statement is interpreted as being associated with the **onAbort** clause; the others are interpreted as standard C statements outside the **transaction** construct.

Part 3. Encina TX Interface

Chapter 7. X/Open TX interface for Encina

This chapter introduces the Encina TX interface and describes several important issues concerning the implementation of the TX interface. The chapter also describes Encina's extensions to the standard interface and the diagnostic support provided for TX.

Introduction

The X/Open TX interface is a standard, well-known API that an application program uses to open and close resource managers, start and end global transactions, direct the completion of transactions, and obtain information about the status of transactions. The Encina TX interface implements the preliminary specification from X/Open as specified in *Distributed Transaction Processing*, published by X/Open Company Ltd., October 1992. Read this specification before reading the Encina TX documentation.

The Encina TX interface can be used in both client and server programs. It is intended as an alternative to Transactional-C for C programmers and is the recommended interface for COBOL programmers. See the *Encina COBOL Programming Guide* for information on the COBOL interface to TX. (COBOL programs are not required to use the Encina Monitor in order to use the TX interface.)

The Encina TX interface interoperates with the Transaction Manager-XA Service (TM-XA), but does not require that the application use TM-XA. Similarly, the TM-XA Service interoperates with the Encina TX interface, but does not require that the application use the TX interface.

As part of the TX implementation in Encina, the Transactional Interface Definition Language (TIDL) used by Encina supports the TxRPC specification and its associated attributes (the `transaction_mandatory` and `transaction_optional` attributes). For more information on these attributes, see the X/Open TX and TxRPC documentation. In order to process files using these attributes, special switches must be used on the **tidl** command line. See the reference page for the **tidl** command for more information.

Implementation issues

The following sections present several important issues involving the implementation of the Encina TX interface.

Threads

The X/Open specification refers to a thread of control as an operating system process. This definition conflicts with Encina's use of multiple threads within an application process. Encina maps the TX specification onto multiple threads consistent with the X/Open specification and the Encina environment. The **tx_open** and **tx_close** functions affect all threads within the process. All other functions in the X/Open specification affect a single thread within the process.

Each Encina thread has its own TX *commit_return*, *transaction_control*, and *transaction_timeout* characteristics. The **tx_set_*** calls change the characteristics of the invoking thread only. When a thread is created, the characteristics are initialized to the values required by the X/Open specification: *commit_return* is set to `TX_COMMIT_COMPLETED`, *transaction_control* is set to `TX_UNCHAINED`, and *transaction_timeout* is set to 0 (zero). Note that a new thread does *not* inherit the transaction characteristics of its parent thread.

Nested transactions

The X/Open specification for TX does not support the use of nested transactions. By default, the Encina TX interface does not allow transactions to be nested, but an application can call the **tx_allow_nesting** function to enable the use of nested transactions. Because the X/Open XA specification also does not support nested transactions, it is recommended that you do not use nested transactions when working with XA-compliant resource managers; such use can result in deadlock.

The TX Interface and Transactional-C

The Encina TX interface is intended as an alternative to Transactional-C (Tran-C). Within a single application process, use either Tran-C or TX , not both. A transaction begun with a TX function must end with a TX function and must not contain any Tran-C constructs (such as **transaction** or **abort**). Likewise, a transaction begun with Tran-C should end with Tran-C, and TX functions (such as the **tx_info** or **tx_rollback** functions) must not be used within that transaction.

Tran-C and TX can, however, be used in a single transaction when more than one process is involved. That is, a Tran-C client can make a transactional remote procedure call (TRPC) to a TX server, or a TX client can make a TRPC to a Tran-C server. Situations in which Tran-C and TX are used in a single transaction can occur when modules from different sources are used in an application, fir example, an application that requires the use of an existing server written in COBOL and new PC-based clients. If the server is ported to Encina, it uses the TX interface, while the PC-based clients can be written using Tran-C.

Chaining transactions

TX enables transactions to be chained together by using the **tx_set_transaction_control** function. If chaining is enabled, a new transaction is begun automatically in the same process when the current transaction is completed.

In the Encina TX interface, the chaining functionality does not invoke the CPI-C syncpoint operation. The **tx_begin** function can be used to initiate the first CPI-C transaction, and the **tx_commit** function can be used to complete the last CPI-C transaction, but the **SRRCMIT** or **SRRBACK** function must be used to chain the intervening transactions. See the *Encina PPC Services Programming Guide* for more information.

X/Open standard interface

TX interface header files and libraries

The file **tx/tx.h** contains the X/Open TX interface declarations for the C language. This file must be included in any C file that uses the TX interface functions.

The Encina TX functions are contained in the **Encina** library. See *Writing Encina Applications* for further information on the libraries used during compilation.

TX interface functions

The Encina TX interface provides C functions that open and close an application's resource managers, begin and end transactions, and set transaction characteristics. The following C functions are exported:

- **tx_begin**
- **tx_close**
- **tx_commit**
- **tx_info**
- **tx_open**
- **tx_rollback**
- **tx_set_commit_return**
- **tx_set_transaction_timeout**

The Encina TX interface also provides COBOL calls that provide the same functionality as the C functions. The COBOL library is written by using the C interface to TX (and, therefore, the behavior is the same); see the *Encina COBOL Programming Guide* for the documentation for the COBOL calls.

Note: The Encina TX documentation is intended to describe only the differences between the X/Open specification and the Encina implementation of the C function descriptions. For additional information, refer to the X/Open TX specification.

Encina extensions to the X/Open interface

The functions defined in this section are not part of the X/Open specification. The following functions extend the standard TX interface:

- **tx_allow_nesting**—enables the nesting of transactions in TX
- **tx_get_rollback_code**—obtains the abort code from the most recent TX transaction that aborted
- **tx_get_rollback_string**—obtains the abort string from the most recent TX transaction that aborted
- **tx_set_rollback_code**—sets the abort code for an aborted transaction
- **tx_set_rollback_string**—sets the abort string for an aborted transaction

The extensions to the standard interface include the **tx_RegisterXaUpcalls** function, which is intended for use only by TM-XA. This C function is documented for completeness only and should not be used in application programs; no COBOL equivalent is defined. See the *Encina Toolkit Programming Guide* for more information on TM-XA.

Header files

The file **tx/tx_extensions.h** contains structure and data type declarations used by the C functions for the Encina extensions to the TX interface. This file must be included in any C file that uses the extended TX interface functions.

Abort reasons

TX generates an abort reason when the **tx_rollback** function is called to abort a transaction. This information is stored in a data structure containing an integer code, a character string, or both, as well as other data used in formatting abort reasons. (See the reference page for the **encina_abortReason_t** data type for details of the abort reason data structure.)

The TX interface defines some default abort reasons that are implemented as abort codes with associated abort strings. If no abort reason has been set explicitly for a transaction before it aborts, one of the default abort reasons is generated. Table 7 lists the abort reasons that are defined. For default abort reasons, the abort code can be retrieved by using the **tx_get_rollback_code** function, and the abort string can be retrieved by using the **tx_get_rollback_string** function.

Table 7. TX interface abort codes

Abort Code	Description

Table 7. TX interface abort codes (continued)

TX_TIMEOUT_ABORT_CODE	Transaction timeout expired. The transaction aborted because the timeout set by the **tx_set_transaction_timeout** function expired.
TX_ROLLBACK_ABORT_CODE	tx_rollback was called. The transaction aborted because the application called the **tx_rollback** function. If no other abort reason is set, this abort reason is returned by default.
TX_THREAD_EXIT_ABORT_CODE	Thread exited without calling tx_commit. The transaction aborted because the associated thread ended before the transaction was committed.

The Encina TX interface provides functions for setting and retrieving abort reasons as abort codes. The **tx_set_rollback_code** function can be used to set an abort code, and the **tx_get_rollback_code** function can be used to retrieve it. These functions rely on other functions and data types defined as part of the Encina Abort Facility. See the documentation for the Abort Facility in the *Encina Toolkit Programming Guide* for more information.

The Encina TX interface also provides functions for setting and retrieving abort reasons as simple text strings. The **tx_set_rollback_string** function can be used to set an abort string to be used as the abort reason for a transaction. Set the abort string immediately before calling the **tx_rollback** function to ensure that the desired abort reason is generated. The abort reason stored by the **tx_rollback** function depends on the following circumstances:

- If no abort reason has been explicitly set before the **tx_rollback** function is called, the TX_ROLLBACK_ABORT_CODE abort code is generated as the abort reason.
- If an abort reason has been set by using either the **tx_set_rollback_string** function or the **tx_set_rollback_code** function, that abort reason is used.
- If the transaction has been aborted by another component, any abort reason set by that component is used.

After the abort reason has been stored, either the **tx_get_rollback_string** or **tx_get_rollback_code** function can be used to retrieve it. The **tx_get_rollback_code** function returns the abort reason in an integer format, and **tx_get_rollback_string** function returns the abort reason in a printable string format. Abort reasons generated by Encina components (such as TX) generally are available in both formats. Abort reasons generated by user programs are available only in the format in which they were registered, unless additional functions (defined by the Encina Abort Facility) are used.

Diagnostics

This section documents the diagnostic support provided by the Encina TX interface. This support takes the form of informative messages that can be generated during use of TX. It also documents the snapshot that the state dump can possibly produce.

In most cases, TX does not return `TX_ERROR` or `TX_FAIL` codes. Encina automatically generates a warning message for a transient error and terminates the application if a fatal runtime error is encountered. The Transaction Manager-XA Service (TM-XA) issues a warning, but does not terminate execution, when a resource manager returns the `XAER_RMFAIL` code.

Directing output

All trace, dump, warning, and fatal error output can be directed to user-specified files or a ring buffer by use of the relevant Encina Toolkit Trace Facility functions. By default, this output is sent to the ring buffer. Use the **trace_DumpRingBuffer** function to dump the ring buffer.

Trace provides an upcall, **trace_FileUpcall**, that can be registered to direct the output to either a user-created file or to the standard error or standard output streams. See the *Encina Toolkit Programming Guide* for more details on the Encina Trace Facility.

Fatal error messages

Encina support for TX provides the following fatal error message:

`tx_RegisterXaUpcalls: status`. This messages indicates that the Transaction Service has not been initialized before calling the **txRegisterXaUpcalls** function.

Warning messages

Encina support for TX provides the following warning messages:

- `tran_Ready failed: status` The call to the **tx_open** function did not succeed because initialization of the application interface did not complete.
- `tx_close called with number uncompleted transactions` A protocol error occurred in the call to the **tx_close** function because the TX Service was closed while uncompleted transactions still existed. Complete the transactions with the **tx_rollback** or **tx_commit** function before calling the **tx_close** function.
- `bde_SetAlarm failed: status` In the call to the **tx_begin** function, the initialization of the transaction timeout characteristic failed.
- `tx_RegisterXaUpcalls: TX already initialized` The call to the **tx_RegisterXaUpcalls** function did not complete, because the TX Service has already been initialized. Call the **tx_RegisterXaUpcalls** function before

calling the **tx_open** function. If you are using TM-XA, call the **tmxa_Init** function before calling the **tx_open** function.

Audit messages

The Encina TX interface does not generate any audit messages.

Trace and state dump information

The trace output generated during the execution of an application can be used to follow the execution path of the application as it makes TX calls. All TX functions are capable of tracing their entry, parameters, and exit.

The level of tracing is controlled by the value of a single variable that is exported by the TX interface:

```
unsigned long tx_traceMask;
```

The Encina Trace Facility defines a number of bit constants that, when set in the above variable, cause a specific form of tracing to occur. For example, the `TRACE_ENTRY` constant enables entry and exit tracing of TX functions.

The following section describes the levels of tracing supported by the Encina TX interface.

Global trace levels

The TX interface supports the following global trace levels defined by the Encina Trace Facility:

- `TRACE_ENTRY`—enables tracing of the entry and exit of functions that make up the TX interface.
- `TRACE_PARAM`—enables tracing of the parameters passed to the interface functions.
- `TRACE_EVENT`—enables tracing of all events in TX.
- `TRACE_GLOBAL`—enables all trace levels.
- `TRACE_NONE`—disables all trace levels.

State dump

The TX interface provides the **tx_DumpState** function to dump the state of TX in an application.

Part 4. Encina Transaction Service Interface

Chapter 8. Using TRAN for transactional programming

The Encina Toolkit is the foundation of Encina's distributed transaction processing system. The Encina Transaction Service (TRAN)—a primary component of the Toolkit—defines the underlying functionality for creating distributed transactions in Encina applications.

This chapter provides an overview of TRAN and describes situations in which you can use TRAN functions instead of the higher-level interfaces such as Tran-C. The chapter includes a small example illustrating the use of common TRAN functions and a summary of other functions provided by low-level components of the Encina Toolkit. These functions can be useful for managing transactions. See the *Encina Toolkit Programming Guide* for details of the TRAN interface and other Toolkit components.

An overview of TRAN

TRAN is a module of the Encina Toolkit that consists of a set of low-level interfaces designed for the definition and management of distributed transactions. TRAN interoperates with Encina's communications and recovery services to provide distributed, recoverable applications.

The functions exported by the TRAN interface provide a wide range of functionality for managing distributed transactions. TRAN functions can be used to begin and end transactions, find out when significant events occur during a transaction, and optimize processing.

Encina's higher-level transactional interfaces (such as Tran-C) are built on top of the TRAN interfaces. Some of the high-level functions and constructs map directly to TRAN functions; many high-level functions, however, provide simplified interfaces to TRAN functionality for the tasks that are regularly used in most applications. For example, Tran-C's **transaction** construct is a simplified interface to TRAN's functions to begin and end transactions—one Tran-C construct replaces several TRAN functions.

Some aspects of TRAN functionality do not have analogous functions in the higher-level interfaces. Although you rarely need to use the majority of TRAN functions directly, you probably want to use a few of them in certain circumstances. "Reasons to use TRAN" on page 128 describes a few of the low-level functions that have proven to be the most useful.

TRAN works in conjunction with other components of the Encina Toolkit such as the Thread-to-Tid Mapping Service (ThreadTid) and the Transactional

Remote Procedure Call Service (TRPC). See "Related Toolkit functionality" on page 132 for information on these Toolkit components.

Reasons to use TRAN

TRAN functions are most commonly used for the following reasons:

- Registering callbacks that are executed when certain events occur during the lifetime of a transaction
- Getting information about the relationship between transactions
- Avoiding conflicts between Tran-C constructs and C++
- Controlling the lexical scope of transactions
- Chaining transactions in Peer-to-Peer Communications (PPC) applications

"A TRAN example" on page 130 provides an example program that uses several of the TRAN functions mentioned in this section. "Related Toolkit functionality" on page 132 lists other low-level features of the Encina Toolkit that are available in managing transactions.

Registering callbacks

The TRAN interface provides a number of functions that you can use to register callbacks that are executed whenever specific transactions enter certain states. Using these functions enables your application to respond to specific changes in the state of a transaction. For example, you want your application to perform specific logging activity before a particular transaction aborts, or you want to initiate other actions based on whether the transaction is prepared to commit (but before it attempts to do so). In these cases, you can use TRAN functions to register the callbacks that perform the appropriate activities.

TRAN defines a different registration function for each change in the state of a transaction. For example, you can register a callback to be executed before a transaction prepares by calling the **tran_CallBeforePrepare** function. The registration functions include the following:

- The **tran_CallAfterCWRT** function registers a callback that is executed after one transaction commits with respect to another.
- The **tran_CallAfterFinished** function registers a callback that is executed after all communications and recovery is completed for a transaction.
- The **tran_CallAfterResolution** function registers a callback that is executed after the outcome of a transaction is determined.
- The **tran_CallAfterRestart** function registers a callback that is executed after a restart.
- The **tran_CallBeforeAbort** function registers a callback that is executed before a transaction aborts.

- The **tran_CallBeforePrepare** function registers a callback that is executed before a transaction prepares.
- The **tran_CallDuringRestart** function registers a callback that is executed during a restart.
- The **tran_CallTransactionallyBeforePrepare** function registers a callback that is executed as part of a transaction before the transaction prepares.

Each registration function requires you to specify a transaction with which to associate the callback.

Note: You must make sure to register your callback for a transaction before the specific state change occurs in that transaction. If the state change occurs before the callback is registered, the callback is never executed.

Getting transactional information

The TRAN interface provides several functions that enable you to obtain information about the relationships between various transactions. For example, if your application uses nested transactions, you can call functions to retrieve information about transactions in the same transaction family. A *transaction family* consists of all nested transactions that have a common ancestor. You can call the **tran_TidIsRelated** function to determine whether two specified transactions belong to the same family. You can call the **tran_TidParent** function to find the parent of a specified transaction and the **tran_TidTopAncestor** function to find the top ancestor of the transaction family to which a specified transaction belongs.

Avoiding C++ language conflicts

In applications in which conflicts between different language environments are possible, consider using the TRAN interface. Although most Encina components can be used directly in C++ applications, C++ can potentially interfere with the proper execution of some Tran-C constructs, for example, the **transaction** construct. Tran-C uses the Distributed Computing Environment (DCE) exception-handling mechanism for handling transaction aborts. This mechanism conflicts with the internal way in which exceptions are handled in many C++ implementations. Tran-C can be used in C++ applications only if exception contexts are managed carefully.

Note: One way to avoid conflicts with C++ is to use Encina++, which is a C++ interface to Encina. Encina++ is documented in the *Encina Object-Oriented Programming Guide*.

As an alternative, consider using the TRAN interface so that the DCE exception mechanism is not invoked automatically in your application when an exception occurs, as it is in Tran-C applications. Instead of using Tran-C constructs, you can use TRAN functions (for example, **tran_Begin**, **tran_End**, and **tran_Abort**) to begin and end transactions.

Controlling the lexical scope of transactions

The TRAN interface allows you to begin a transaction in one function and end it in another; the Tran-C **transaction** construct does not. Each **transaction** construct—and the **onCommit** and **onAbort** clauses associated with it—must appear within the same lexical scope of your application. Similar constructs, such as the **topLevel** or **resumeTran** construct, have the same restriction. These constructs and their associated clauses are macro based; therefore, when constructs are expanded during compilation of a Tran-C program, the code into which they are expanded must begin and end within the same function or routine.

If your application requires that a transaction begin and end in different functions, you can use TRAN functions to delimit the transaction. For example, you can call **tran_Begin** in one function and **tran_End** in another.

Chaining transactions in PPC applications

PPC Executive applications that both use synclevel syncpoint conversations and chain transactions must use either TRAN or the Encina TX interface to start the first transaction and end the last transaction. The application cannot use Tran-C, because Tran-C does not support chaining. Refer to the *Encina PPC Services Programming Guide* for details.

A TRAN example

This section provides an example that illustrates the use of TRAN functions that begin, commit, and abort transactions. The example (shown in Figure 39 on page 131) also shows how to initialize an application to use TRAN and how to register a TRAN callback.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <tran/tran.h>
#include <trpc/trpc.h>
void before_abort(tran_tid_t tid, void *arg);

int main()
{
  int i;
  tran_tid_t tid;

  tran_StandardEnvironment();
  tran_Init(0);
  tran_Ready();

  for (i=0;i<5;i++) {
    tran_Begin(TRAN_TID_NULL, &tid);
    tran_CallBeforeAbort(tid, before_abort, &i);

    printf("Hello World - transaction %d\n", tid);

    if (i % 2) {
      printf("\tOdd-numbered transactions are aborted...\n");
      tran_Abort(tid);
      printf("\t(Transaction aborted)\n");
    }
    else {
      tran_End(tid);
      printf("\t(Transaction committed)\n");
    }
  }
  tran_Terminate();
}


void before_abort(tran_tid_t tid, void *arg)
{
  printf("\tExecuting the before-abort callback for "
         "transaction %d...\n", tid);
}
```

Figure 39. The "Hello, World" program in TRAN

The example is a simple "Hello, World" program that prints a message for each transaction it creates. The **tran_Begin** function creates a transaction and returns an identifier for that transaction. Transactions that have even-numbered transaction identifiers commit when the **tran_End** function is called. Transactions that have odd-numbered transaction identifiers abort when the **tran_Abort** function is called.

In addition, a before-abort callback is registered for each transaction. The call to the **tran_CallBeforeAbort** function registers the **before_abort** function defined in the example code. If the transaction aborts, the callback is executed before any recovery work is performed for the transaction.

The calls to the **tran_StandardEnvironment**, **tran_Init**, and **tran_Ready** functions initialize the environment, initialize TRAN, and notify TRAN that the application is ready to begin creating transactions. The call to the **tran_Terminate** function terminates TRAN.

Figure 40 shows some sample output for the example program in Figure 39 on page 131.

```
Hello World - transaction 65536
        (Transaction committed)
Hello World - transaction 1
        Odd-numbered transactions are aborted...
        Executing the before-abort callback for transaction 1...
        (Transaction aborted)
Hello World - transaction 2
        (Transaction committed)
Hello World - transaction 3
        Odd-numbered transactions are aborted...
        Executing the before-abort callback for transaction 3...
        (Transaction aborted)
Hello World - transaction 4
        (Transaction committed)
```

Figure 40. Output from the TRAN "Hello, World" program

Related Toolkit functionality

Other components of the Encina Toolkit provide functionality that can be used with TRAN. This section describes the use of ThreadTid with TRAN transactions, and it lists a few additional features of Toolkit components that are seldom required but are sometimes useful in transactional applications.

Associating transactions with threads

Depending on the requirements of your application, consider using ThreadTid functions as part of the process of beginning and ending transactions. ThreadTid can be used to associate a transaction with the thread on which it was created. This association makes the identifier of the transaction available in any procedure called within the scope of the transaction; otherwise, you must pass the transaction identifier explicitly to procedures that need it. In addition, if your application is using an Encina resource (such as the Encina Structured File Server or Recoverable Queueing Service), you must use ThreadTid so that the resource can determine the transaction on whose behalf it is working.

After the transaction is begun, you can call the **threadTid_Begin** function to make the association. Once this association is made, you can call the **threadTid_Lookup** function to get the transaction identifier in any transactional remote procedure call (TRPC) called within the scope of the transaction. To end the association between the transaction and the thread, you can call the **threadTid_End** function before ending the transaction with either the **tran_End** or **tran_Abort** function. Refer to the *Encina Toolkit Programming Guide* for more information on ThreadTid.

Using other features of the Toolkit

In special circumstances, other features of TRAN and other low-level Toolkit components can be useful for the management of transactions. The following is a list of components and some less commonly used features of those components:

- ThreadTid
  - Registering callbacks that are executed when a transaction identifier is set or unset for a thread
  - Certifying and decertifying threads to control when transaction aborts occur
- TRAN
  - Choosing a coordinator for a transaction
  - Setting and retrieving transaction properties
- TRPC
  - Registering callbacks that are executed when certain events occur during a transactional RPC
  - Setting automatic timeouts for inactive transactions

These less commonly used features are not described in this manual. Refer to the *Encina Toolkit Programming Guide* for more information about these features of the Toolkit components.

Part 5. Appendixes

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS DOCUMENT "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will

be incorporated in new editions of the document. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
ATTN: Software Licensing
11 Stanwix Street
Pittsburgh, PA 15222
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks and service marks

The following terms are trademarks or registered trademarks of the IBM Corporation in the United States, other countries, or both:

Advanced Peer-to-Peer Networking
AFS
AIX
APPN
AS/400
CICS
CICS OS/2
CICS/400
CICS/6000
CICS/ESA
CICS/MVS
CICS/VSE
CICSPlex
DB2
DCE Encina Lightweight Client
DFS
Encina
IBM
IBM System Application Architecture
IMS
IMS/ESA
Informix
Language Environment
MQSeries
MVS
MVS/ESA
NetView
Open Class
OS/2
OS/390
OS/400
Parallel Sysplex
PowerPC
RACF
RAMAO
RMF
RISC System/6000
RS/6000
S/390
SAA
SecureWay
TeamConnection
Tivoli
TXSeries
VSE/ESA
VTAM
VisualAge
WebSphere

Domino, Lotus, and LotusScript are trademarks or registered trademarks of Lotus Development Corporation in the United States, other countries, or both.

ActiveX, Microsoft, Visual Basic, Visual C++, Visual J++, Visual Studio, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Pentium is a trademark of Intel Corporation in the United States, other countries, or both.

This software contains RSA encryption code.

Other company, product, and service names may be trademarks or service marks of others.

Index

S

T

U

V

W