HP NonStop TMF Application Programmer's Guide

Abstract

This manual describes how to design requester modules and server modules that run in the HP NonStop™ Transaction Management Facility (TMF) programming environment on HP NonStop servers, and how to use the TMF audit-reading procedures. It discusses features and operations available with the TMF 3.7 product.

Product Version

TMF H01

Supported Releases

This manual supports J06.03 and all subsequent J-series RVUs and H06.06 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
540139-009	February 2014

Document History

Part Number	Product Version	Published
540139-002	TMF H01	November 2005
540139-003	TMF H01	April 2006
540139-004	TMF H01	February 2009
540139-006	TMF H01	August 2010
540139-007	TMF H01	February 2012
540139-008	TMF H01	February 2013
540139-009	TMF H01	February 2014

Legal Notices

© Copyright 2005, 2014 Hewlett-Packard Development Company, L.P.

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Export of the information contained in this publication may require authorization from the U.S. Department of Commerce.

Microsoft, Windows, and Windows NT are U.S. registered trademarks of Microsoft Corporation.

Intel, Itanium, Pentium, and Celeron are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Motif, OSF/1, UNIX, X/Open, and the "X" device are registered trademarks and IT DialTone and The Open Group are trademarks of The Open Group in the U.S. and other countries.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE OSF MATERIAL PROVIDED HEREIN, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material.

© 1990, 1991, 1992, 1993 Open Software Foundation, Inc. This documentation and the software to which it relates are derived in part from materials supplied by the following:

© 1987, 1988, 1989 Carnegie-Mellon University. © 1989, 1990, 1991 Digital Equipment Corporation. © 1985, 1988, 1989, 1990 Encore Computer Corporation. © 1988 Free Software Foundation, Inc. © 1987, 1988, 1989, 1990, 1991 Hewlett-Packard Company. © 1985, 1987, 1988, 1989, 1990, 1991, 1992 International Business Machines Corporation. © 1988, 1989 Massachusetts Institute of Technology. © 1988, 1989, 1990 Mentat Inc. © 1988 Microsoft Corporation. © 1987, 1988, 1989, 1990, 1991, 1992 SecureWare, Inc. © 1990, 1991 Siemens Nixdorf Informationssysteme AG. © 1986, 1989, 1996, 1997 Sun Microsystems, Inc. © 1989, 1990, 1991 Transarc Corporation.

This software and documentation are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California. OSF acknowledges the following individuals and institutions for their role in its development: Kenneth C.R.C. Arnold, Gregory S. Couch, Conrad C. Huang, Ed James, Symmetric Computer Systems, Robert Elz. © 1980, 1981, 1982, 1983, 1985, 1986, 1987, 1988, 1989 Regents of the University of California.

Printed in the US

HP NonStop TMF Application Programmer's Guide

Index Examples Figures Tables

Legal Notices

What's New in This Manual vii
- Manual Information vii
- New and Changed Information viii

About This Manual xiii
- Who Should Read This Manual xiii
- How this Manual is Organized xiii
- About the TMF Library xiii
- Related Manuals xiii
- Notation Conventions xiv
- HP Encourages Your Comments xvii

1. TMF Programming Environment

The TMF Transaction 1-2
- Defining a Transaction 1-2
- Initiating a Transaction 1-3
- Committing a Transaction 1-4
- Aborting a Transaction 1-4

Heterogeneous Transaction Processing 1-5

The Requester/Server Model 1-6

The Current Transaction 1-7
- The Nil State 1-8
- The Current Transaction Identifier 1-8
- Excluding a Server from a TMF Transaction 1-8
- Setting the Current Transaction to Nil 1-10
- Marking an OPEN to Not Share a Transaction Identifier 1-10

Consistency and Concurrency 1-11
- Achieving Maximum Consistency 1-11
- Levels of Consistency 1-12
- Enscribe Capabilities 1-14

2. Designing Single-Threaded Processes

Single-Threaded Requesters 2-1
Applicable System and TMF Procedures 2-1
Delegating Work to Servers 2-2
Terminating Transactions 2-3
Checkpointing Strategy 2-6
Single-Threaded Servers 2-11
Applicable System Procedures 2-11
Opening $RECEIVE 2-12
Matching Each READUPDATE With a REPLY 2-12
WRITEREAD to Another Server 2-13
The Use and Implications of ABORTTRANSACTION 2-13
The Implications of REPLY 2-14
NonStop Servers 2-14
Guarantees to Servers 2-15
Context-Sensitive Servers 2-15

3. Designing Multithreaded Processes

Multithreaded Requesters 3-1
Opening the TFILE 3-1
Manipulating the Current Transaction 3-2
Nowait ENDTRANSACTION Calls 3-3
Checkpointing Strategy 3-3
Multithreaded Servers 3-12
Opening $RECEIVE 3-12
Manipulating the Current Transaction 3-13
Replying to Requesters 3-13
NonStop Servers 3-13
Multithreaded Requester/Server Processes 3-14

4. File System Procedures

ABORTTRANSACTION 4-3
ACTIVATERECEIVETRANSID 4-5
BEGINTRANSACTION 4-7
BEGINTRANSACTION_EXT_ 4-10
COMPUTETRANSID 4-12
ENDTRANSACTION 4-15
GETTMPNAME 4-17
GETTRANSACTIONDETAILS 4-19

GETTRANSID 4-24
GETTRANSINFO 4-26
INTERPRETTRANSID 4-28
RESUMETRANSACTION 4-31
STATUSTRANSACTION 4-34
TEXTTOTRANSID 4-37
TMF_BEGINTAG_FROM_TXHANDLE_ 4-40
TMF_GETTXHANDLE_ 4-43
TMF_GET_EXTTRANSID_ 4-44
TMF_GET_TX_ID_ 4-46
TMF_JOIN_ 4-48
TMF_JOIN_EXT_ 4-50
Usage Considerations 4-51
TMF_RESUME_ 4-53
TMF_SETTXHANDLE_ 4-55
TMF_SUSPEND_ 4-57
TMF_SUSPEND_EXT_ 4-59
TMF_TXBEGIN_ 4-61
TMF_TXHANDLE_FROM_BEGINTAG_ 4-63
TRANSIDTOTEXT 4-65
TMF_VERSION_EXT_ 4-67

5. TMF ARLIB2 Audit-Reading Procedures

ARLIB2 Compared to ARLIB 5-2
Cursors 5-3
Cursor Declaration 5-3
Cursor Positioning 5-5
Restoring Audit-Trail Files From Audit Dumps 5-5
Retrieving Information From Audit Records 5-6
Error Reporting 5-7
Return Codes 5-7
Messages Printed to the Operator Terminal 5-7
Procedural Retrieval of Message Text 5-8
Audit Compression 5-8
Enscribe 5-8
NonStop SQL/MP 5-8
Reading Active Audit Files 5-9
Reading a Range of Audit-Trail Files 5-10
Reading a Merged Audit Trail With a MERGE Cursor 5-11

Reading a Merged Audit Trail Without a MERGE Cursor 5-12
Reading Audit Records for SQL/MX Objects 5-13
Distributed Transactions 5-13
Auxiliary Audit Trails 5-16
  Subset Audit Records 5-16
NonStop SQL/MP Internal Field Formats 5-17
  Field Alignment 5-17
  Variable-Length Character (VARCHAR) Fields 5-18
  DATETIME and INTERVAL Fields 5-19
  Null Fields 5-19
Audit Records 5-20
  Record Types 5-20
  Record Formats 5-21
  Field Descriptions 5-29
Procedure Calls 5-33
  ARCLOSE 5-36
  ARCOMPLETEIO 5-37
  ARFETCHAFTERIMAGE 5-38
  ARFETCHAUXPOINTER 5-40
  ARFETCHBEFOREIMAGE 5-41
  ARFETCHCHILDNODELIST 5-42
  ARFETCHFIELDVALUE 5-43
  ARFETCHFRAGMENT 5-46
  ARFETCHMXAFTERDATA 5-48
  ARFETCHMXAFTERDATA2 5-51
  ARFETCHMXBEFOREDATA 5-54
  ARFETCHMXBEFOREDATA2 5-57
  ARFETCHRECORDKEY 5-60
  ARGETANSINAME 5-62
  ARGETAUDRECHEADERINFO 5-66
  ARGETFIELDINFO 5-67
  ARGETMESSAGELINE 5-71
  ARGETMXCOLUMNINFO 5-73
  ARGETNETWORKRECS 5-78
  ARGETNONDATACHNGRECS 5-79
  ARGETRECADDR 5-80
  ARGETRECADDR64 5-83
  AROPEN 5-86
  ARPOSITION 5-89

ARPOSITION2 5-92
ARPRINTMESSAGE 5-94
ARREAD 5-95
ARSETOPTIONS 5-97
ARSTART 5-99
ARSTOP 5-101
ARSTOPNETWORKRECS 5-102
ARSTOPNONDATACHNGRECS 5-103
Error Codes 5-104
How to Include Audit Reading in an Application 5-111
Use of AWAITIOX 5-112

Index

Examples

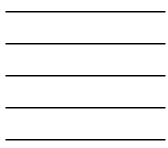
Example 5-1. IMAGEINFO and COLUMNINFO Definitions 5-76

Figures

Figure 2-1. Checkpointing Within Single-threaded Requesters 2-9
Figure 3-1. The Flow of an Individual Thread 3-6
Figure 3-2. Multithreaded Requester Flow Chart 3-7
Figure 3-3. Multithreaded Requester; Detailed Functionality 3-8
Figure 5-1. Basic Parent-Child Relationship 5-14
Figure 5-2. Layered Offspring Relationships 5-15

Tables

Table 1-1. Enscribe Locking Modes 1-16
Table 2-1. WRITEREAD Error Numbers 2-2
Table 2-2. Unilateral Abort Error Numbers 2-5
Table 5-1. RECTYPE Constants 5-20
Table 5-2. TMF Audit-Reading Procedures 5-33
Table 5-3. Error Codes by Class 5-104
Table 5-4. Subsystem Codes 5-108
Table 5-5. Errors returned in case return-code is -1000 5-108
Table 5-6. Files Supplied With TMFARLB2 (T2781) 5-112



What's New in This Manual

Manual Information

Abstract

This manual describes how to design requester modules and server modules that run in the HP NonStop™ Transaction Management Facility (TMF) programming environment on HP NonStop servers, and how to use the TMF audit-reading procedures. It discusses features and operations available with the TMF 3.7 product.

Product Version

TMF H01

Supported Releases

This manual supports J06.03 and all subsequent J-series RVUs and H06.06 and all subsequent H-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
540139-009	February 2014

Document History

Part Number	Product Version	Published
540139-002	TMF H01	November 2005
540139-003	TMF H01	April 2006
540139-004	TMF H01	February 2009
540139-006	TMF H01	August 2010
540139-007	TMF H01	February 2012
540139-008	TMF H01	February 2013
540139-009	TMF H01	February 2014

New and Changed Information

Changes to the H06.28/J06.17 Manual

- Added a Note for the following procedure:
    - ARLIB2 Compared to ARLIB on page 5-2.

Changes to the H06.26/J06.15 Manual

- Added the procedure GETTRANSACTIONDETAILS on page 4-19.
- Added the procedure TMF_VERSION_EXT_ on page 4-67.

Changes to the H06.24/J06.13 Manual

- Added the procedure TMF_GET_EXTTRANSID_ on page 4-44.
- Added the 64 bit syntax for the following procedures:
    - BEGINTRANSACTION on page 4-7
    - BEGINTRANSACTION_EXT_ on page 4-10
    - COMPUTETRANSID on page 4-12
    - GETTMPNAME on page 4-17
    - GETTRANSID on page 4-24
    - GETRANSINFO on page 4-26
    - INTERPRETTRANSID on page 4-28
    - STATUSTRANSACTION on page 4-34
    - TEXTTOTRANSID on page 4-37
    - TMF_BEGINTAG_FROM_TXHANDLE_ on page 4-40
    - TMF_GETTXHANDLE_ on page 4-43
    - TMF_GET_TX_ID_ on page 4-46
    - TMF_JOIN_EXT_ on page 4-50
    - TMF_SETTXHANDLE_ on page 4-55
    - TMF_SUSPEND_ on page 4-57
    - TMF_SUSPEND_EXT_ on page 4-59
    - TMF_TXBEGIN_ on page 4-61
    - TMF_TXHANDLE_FROM_BEGINTAG_ on page 4-63

  - TRANSIDTOTEXT on page 4-65
- Added a Note for the following procedures:
  - ABORTTRANSACTION on page 4-3
  - ACTIVATERECEIVETRANSID on page 4-5
  - ENDTRANSACTION on page 4-15
  - TMF_JOIN_ on page 4-48
  - TMF_RESUME_ on page 4-53
- Added new parameters for the following procedures:
  - ARCLOSE on page 5-35
  - ARFETCHFIELDVALUE on page 5-42
  - ARFETCHMXAFTERDATA on page 5-47
  - ARFETCHMXAFTERDATA2 on page 5-50
  - ARFETCHMXBEFOREDATA on page 5-53
  - ARFETCHMXBEFOREDATA2 on page 5-56
  - ARGETANSINAME on page 5-61
  - ARGETFIELDINFO on page 5-66
  - ARGETMXCOLUMNINFO on page 5-72
  - ARGETRECADDR on page 5-79
  - ARGETRECADDR64 on page 5-82
  - AROPEN on page 5-85
  - ARPOSITION on page 5-88
  - ARPOSITION2 on page 5-91
  - ARREAD on page 5-94
  - ARSTOP on page 5-100
- Updated the meaning for the error code -1000 in Table 5-3, Error Codes by Class, on page 5-103.
- Added the following tables:
  - Table 5-4, Subsystem Codes, on page 5-107
  - Table 5-5, Errors returned in case return-code is -1000, on page 5-107

Changes to the H06.21/J06.10 Manual

- Supported release statements have been updated to include J-series RVUs.
- Added the following procedures:
  - BEGINTRANSACTION_EXT_ on page 4-10
  - GETTRANSINFO on page 4-26
  - TMF_JOIN_EXT_ on page 4-50
  - TMF_SUSPEND_EXT_ on page 4-59
- Updated the description of UPDATE (10) on page 5-27.
- Updated the parameters of the following procedures:
  - ARFETCHAFTERIMAGE on page 5-37
  - ARFETCHBEFOREIMAGE on page 5-40
  - ARFETCHCHILDNODELIST on page 5-41
  - ARFETCHFIELDVALUE on page 5-43
  - ARFETCHMXAFTERDATA on page 5-47
  - ARFETCHMXAFTERDATA2 on page 5-50
  - ARFETCHMXBEFOREDATA on page 5-53
  - ARFETCHMXBEFOREDATA2 on page 5-56
  - ARFETCHRECORDKEY on page 5-59
  - ARGETANSINAME on page 5-62 and on page 5-63
  - ARGETFIELDINFO on page 5-66, page 5-67, and page 5-68
  - ARGETMESSAGELINE on page 5-70 and on page 5-71
  - ARGETMXCOLUMNINFO on page 5-72
  - ARGETRECADDR on page 5-79
  - ARGETRECADDR64 on page 5-82
  - AROPEN on page 5-85
  - ARREAD on page 5-94
- Added a new parameter, *next-field*, under ARFETCHFIELDVALUE on page 5-43.
- Added a new parameter, *guardian-name-length*, under ARGETANSINAME on page 5-62.
- Added the following parameters under ARGETFIELDINFO:

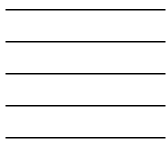
- `flags` on page 5-68
- `collation-def` on page 5-68

- Changed ARGETNONDATACHANGERECS to ARGETNONDATACHNGRECS on page 5-78.
- Added a new parameter, *`return-TypeFlags`*, under ARSETOPTIONS on page 5-97.
- Added the following error codes in Table 5-3, Error Codes by Class on page 5-94:
  - -903
  - -904
  - -905
- Updated the example in the How to Include Audit Reading in an Application on page 5-110.
- Added information about using the ZCLIDLL and ZCLIPDLL DLLs with the TMFARLB2 product on page 5-110.

Changes in the earlier version of the Manual

This is the third edition of the *HP NonStop TMF Application Programmer's Guide*. It has been updated to support the H06.04 release version update (RVU) of the TMF product, and to correct and clarify elements in the previous edition. The changes are indicated with change bars under these topics:

- Unilateral Aborts on page 2-4
- The Transaction Pseudofile (TFILE) on page 2-6
- Placement of Checkpoints on page 2-7
- Opening the TFILE on page 3-1



About This Manual

This manual also shows how to design requesters and servers to run effectively in that environment and describes a set of procedures that you can use to examine the contents of a TMF audit trail.

Who Should Read This Manual

This manual is intended for persons who design requester/server modules that run in the TMF 3.6 programming environment.

How this Manual is Organized

Section 1, TMF Programming Environment, provides an overview of the TMF programming environment.

Section 2, Designing Single-Threaded Processes, describes how to design single-threaded requesters and servers.

Section 3, Designing Multithreaded Processes, describes how to design multithreaded requesters and servers.

Section 4, File System Procedures, presents the syntax descriptions of those file-system procedures that apply specifically to TMF.

Section 5, TMF ARLIB2 Audit-Reading Procedures, describes the audit-reading procedures.

About the TMF Library

This manual is part of the **TMF** library of manuals, which also includes the following manuals:

- *TMF Introduction*
- *TMF Reference Manual*
- *TMF Operations and Recovery Guide*
- *TMF Planning and Configuration Guide*
- *TMF Management Programming Manual*
- *TMF Glossary*

Related Manuals

Other manuals that provide information about how TMF interfaces to HP software products are:

- *Introduction to Data Management* provides an overview of HP data-management products, including TMF, and discusses the use of those products in OLTP applications.
- *Enscribe Programmer's Guide* discusses writing applications for using TMF when accessing an Enscribe database.
- *Guardian Procedure Calls Reference Manual* describes the syntax of many system procedure calls that are used in OLTP applications.
- *HP NonStop SQL/MP Reference Manual* describes the language elements and syntax for HP NonStop SQL/MP that are used in the TMF environment.
- Programming language manuals for NonStop systems describe I/O procedures that include accessing TMF protected data. These languages include COBOL85, PATHWAY SCREEN COBOL, FORTRAN, TAL, Pascal, C, C++, and SQL (NonStop SQL/MP implementation).

Notation Conventions

General Syntax Notation

The following list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words; enter these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

`file-name`

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name

INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list may be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
LIGHTS [ ON               ]
       [ OFF              ]
       [ SMOOTH [ num ] ]

K [ X | D ] address-1
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list may be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

… Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]…

[ - ] {0|1|2|3|4|5|6|7|8|9}…
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char…"
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be entered as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must enter as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In the following example, there are no spaces permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] CONTROLLER

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                      !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length           !i:i
                            , filename2:length ) ;       !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                        !i
                        , [ filename:maxlen ] ) ;        !o:i
```

Change Bar Notation

Change bars are used to indicate substantive differences between this edition of the manual and the preceding edition. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).

The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to docsfeedback@hp.com.

Include the document title, part number, and any comment, error found, or suggestion for improvement you have concerning this document.

1 TMF Programming Environment

The Transaction Management Facility (TMF) provides protection and concurrency control for HP NonStop SQL/MP and Enscribe files.

Database integrity is vital to the success of any online transaction processing (OLTP) environment. It is imperative that data not become corrupted as the result of hardware or software failures, or conflicting operations performed concurrently against the same data.

TMF protects the integrity of your database against failure of any of the following:

- A disk drive
- A disk controller
- A disk process
- An application process
- A CPU
- An entire system

In addition, TMF provides locking mechanisms with which you can achieve various levels of isolation from the effects of other concurrently running application processes.

As an integral part of the operating system, much of the TMF functionality is provided by system components such as the file system and the DP2 disk process. From the perspective of the application programmer, TMF and the protection it provides are often entirely transparent.

This section contains the following topics:

- The TMF Transaction on page 1-2
- Heterogeneous Transaction Processing on page 1-5
- The Requester/Server Model on page 1-6
- The Current Transaction on page 1-7
- Consistency and Concurrency on page 1-11

The TMF Transaction

Within the TMF programming environment, program operations that modify the content of a protected database must be grouped together into TMF transactions.

From a syntactical perspective, a TMF transaction is an executed sequence of code delimited by two unique programming statements: one indicating the start of the transaction and the other indicating the end of the transaction.

From a program design perspective, however, the decision as to what best constitutes a TMF transaction requires careful judgement; unfortunately, there is no simple formula.

Defining a Transaction

The following characteristics might be helpful in defining transactions.

Shared Destiny

All of the database operations within a particular transaction have a shared destiny: they are either all performed and their results committed (made permanent) or all aborted and their results backed out.

Independent of Other Database Operations

In general, database operations that are not dependent upon one another should not be bound together into a transaction. If the results of one operation leave the database in a consistent state regardless of the results of another operation, the two operations probably should not be bound together within a single TMF transaction.

Single Business Operation

Each TMF transaction often reflects a single business operation. For example, consider the situation in which a customer is performing a sequence of operations at an automated teller machine (ATM). The overall session at the ATM might include the following operations:

- A transfer of funds from the customer’s checking account to a savings account
- A transfer of funds from the checking account to a charge card account
- A cash withdrawal from the checking account

The session divides most effectively into three TMF transactions: one for the checking-to-savings transfer, one for the checking-to-charge transfer, and one for the cash withdrawal.

The primary reason for dividing the session into three transactions instead of making it a single TMF transaction is that each operation independently transforms the database from one consistent state to another consistent state. There is no logical reason why a

communications line failure during the cash withdrawal transaction, for example, should negate the successful outcome of the two preceding fund transfer operations.

In this particular example, the situation is further complicated in that each of the individual business operations results in the issuance of a printed receipt to the customer upon successful completion of the operation. If the entire session were to be backed out because of a line failure while processing the cash withdrawal, the customer would possess two printed receipts indicating successful fund transfers that, in fact, did not take place.

Completion Time

Occasionally, elapsed time is an issue to consider. If a transaction remains on the system too long, TMF aborts it. In most application environments, individual transactions exist for only a few seconds or minutes. In a very busy environment, however, a transaction that exists for more than an hour is in real danger of being aborted by TMF. This problem can be remedied somewhat at execution time by the use of operator commands. A preferable solution, if at all possible, is to design your transactions so they do not take too long to complete.

Initiating a Transaction

You initiate a TMF transaction by invoking the BEGINTRANSACTION procedure. The following table summarizes the statements that you use to do so, in each of the supported programming languages:

Programming Language	Statement Used for Invoking the BEGINTRANSACTION Procedure
C, C++	*status* = BEGINTRANSACTION (*trans-tag*);
COBOL85	ENTER “BEGINTRANSACTION” USING *trans-tag* GIVING *status*
FORTRAN	*status* = BEGINTRANSACTION (*trans-tag*)
NonStop SQL/MP	BEGIN WORK
Pascal	*status*:= BEGINTRANSACTION (*trans-tag*)
SCREEN COBOL	BEGIN–TRANSACTION
TAL	*status*:= BEGINTRANSACTION (*trans-tag*);

The BEGINTRANSACTION procedure generates a transaction identifier that uniquely identifies each transaction. Thereafter, the transaction identifier is transparently affixed to all database operations and interprocess communication performed within the domain of the particular transaction.

Any program module can initiate a TMF transaction; within each module that does so, however, you must always explicitly match every BEGINTRANSACTION call with a corresponding call to ENDTRANSACTION or ABORTTRANSACTION. This is true even if TMF or another process has already aborted the transaction.

Committing a Transaction

If a transaction completes successfully, the changes that it made to a protected database are made permanent; this is referred to as "committing the transaction."

You terminate a TMF transaction and commit the effects of that transaction by invoking the ENDTRANSACTION procedure. The following table summarizes the statements that you use to do so, in each of the supported programming languages:

Programming Language	Statement Used for Invoking the ENDTRANSACTION Procedure
C, C++	*status* = ENDTRANSACTION;
COBOL85	ENTER "ENDTRANSACTION" GIVING *status*
FORTRAN	*status* = ENDTRANSACTION
NonStop SQL/MP	COMMIT WORK
Pascal	*status*:= ENDTRANSACTION
SCREEN COBOL	END—TRANSACTION
TAL	*status*:= ENDTRANSACTION;

Aborting a Transaction

After a TMF transaction has been initiated but before it is committed, any application program module that is doing work on behalf of that transaction can terminate the transaction; this is referred to as "aborting the transaction."

When a transaction is aborted, all of the changes that were made to the database on behalf of the transaction are backed out.

You abort a transaction by invoking the ABORTTRANSACTION procedure. The following table summarizes the statements that you use to do so, in each of the supported programming languages:

Programming Language	Statement Used for Invoking the ABORTTRANSACTION Procedure
C, C++	*status* = ABORTTRANSACTION;
COBOL85	ENTER "ABORTTRANSACTION" GIVING status
FORTRAN	*status* = ABORTTRANSACTION
NonStop SQL/MP	ROLLBACK WORK
Pascal	*status*:= ABORTTRANSACTION
SCREEN COBOL	ABORT—TRANSACTION
TAL	*status*:= ABORTTRANSACTION;

Heterogeneous Transaction Processing

In heterogeneous transaction processing, TMF can start a transaction and then subcontract portions of the transaction (branches) to one or more foreign transaction management systems operating on platforms other than the HP system. Alternatively, a foreign transaction management system can begin a transaction and then subcontract parts to TMF. After completing the transaction, TMF and the foreign transaction manager participate in an agreement protocol to determine the outcome of the transaction. This cooperation between different transaction management systems running on different platforms relies on collections of routines called resource managers.

Resource managers encode the transactional semantics of foreign transaction management systems, or provide abstractions of foreign database management systems on the HP platform. They run in a process environment called a gateway process. A gateway process executes a resource manager routine to export a transaction branch from TMF to a foreign transaction management system, import a transaction branch from a foreign transaction management system to TMF, and participate in the agreement protocol between the two transaction management systems.

Note. Heterogeneous transaction processing, and the resource managers that support this processing, are used in HP products such as NonStop TUXEDO. Although TMFCOM and the TMFSERVE programmatic interface provide TMF commands for operating on resource managers, people issuing these commands do so in the context of issues involving multiple software subsystems and inter-platform considerations. Discussion of those commands and the context in which they are used is beyond the scope of this manual.

Readers who want more information about heterogeneous transaction processing are directed to the Open Group TRANSACTION PROCESSING Publications, available from X/Open Publications, PO Box 96, Witney, Oxon. OX8 6PG, U.K. Phone:+44 (0)1993 708731
Fax: +44 (0) 1993 708732, or from the following Web location:

http://www.rdg.opengroup.org/public/pubs/catalog/tp.htm

These publications can also be ordered through many bookstores.

The Requester/Server Model

Many business applications that run on HP systems use the **requester/server** application design model to obtain input from end users and apply it to a database.

In the most elementary case, a requester is the process that receives the initial request to perform a business operation requiring modification of a database. The requester then initiates a TMF transaction, calls one or more server processes to make the necessary changes to the database, terminates the transaction, and responds to the end user (or to the application module that represents an end user).

In reality, however, application modules are usually much more complex than that.

Requester and server modules can be designed to work on more than one transaction at a time; this is referred to as “multithreaded operation.” In addition to delegating the work for a particular transaction to one or more servers, a requester can update a database directly. Furthermore, a single application module can even function simultaneously as a requester for some transactions and as a server for others.

TMF requesters can be coded in C, C++, COBOL85, FORTRAN, Pascal, SCREEN COBOL, or TAL.

TMF servers can be coded in C, C++, COBOL85, FORTRAN, Pascal, or TAL.

Within the TMF environment, requesters and servers communicate with one another over the message system.

To communicate with a server, a requester must first open the server process by using the OPEN system procedure. The OPEN procedure includes, as its first parameter, the server process’ name instead of a file name. All necessary process names, in local and network form, can be predefined so that the particular processes are known throughout both the system and the network.

To send a request to a server and receive a reply, the requester uses the WRITEREAD procedure (specifying the *filenumber* returned by OPEN). Note that SCREEN COBOL requesters use the SEND verb to communicate with servers.

After opening a server, a requester will often leave it open indefinitely. Eventually, however, the requester terminates its communication path to the server by using the CLOSE system procedure.

To communicate with a requester, a server must open a special file named $RECEIVE. Thereafter, the server accepts work request messages from requesters by using the READUPDATE system procedure to read from $RECEIVE. When the server has completed all of the operations associated with a particular work request, it responds to the requester by using the REPLY system procedure.

Summary of Requester Actions

In general, TMF requester processes do the following:

- Open any necessary server processes.
- Initiate TMF transactions.
- Format work requests describing the necessary database manipulations and send them to the appropriate servers by way of the message system.
- Receive reply messages from the servers, indicating success or failure.
- Terminate transactions.

Summary of Server Actions

In general, TMF server processes do the following:

- Accept request messages from requesters by way of $RECEIVE.
- Obtain any necessary file or record locks.
- Once acquired, the locks on any records or rows that have been updated, inserted, or deleted are automatically maintained by the disk process until the transaction is either committed or aborted and backed out.
- Execute I/O statements to perform the requested actions against the database.
- Determine the success or failure of each I/O operation by way of the associated completion code or file system error code.
- Format reply messages and send them to the appropriate requesters.

The Current Transaction

An important concept within the TMF programming environment is that of the current transaction. The current transaction is an implicit identifier that specifies the particular transaction on behalf of which a process can perform operations on files being audited by the TMF subsystem.

At any given time, the current transaction for a requester or server is either of the following:

- A nil value, indicating that there currently is no transaction in progress
- The transaction identifier of the particular transaction on which the requester or server is currently working

The Nil State

Requesters and servers start execution with their current transaction in the nil state.

If a process attempts to lock or change the content of an audited file when the current transaction is in the nil state, the file system rejects the particular I/O request with a condition code of CCL and an error number 75 (no transaction identifier).

When a requester initiates a transaction, the current transaction for that requester automatically changes from the nil state to the transaction identifier of the new transaction.

The Current Transaction Identifier

When a requester sends a work request to a server process, the value of the requester's current transaction at the time of the transmission is extracted and used as the transaction identifier of the particular work request. Upon receiving the message, the server automatically inherits the transaction identifier as its current transaction.

Note that a process can always access nonaudited files regardless of the state of its current transaction.

When a process executes an ENDTRANSACTION or ABORTTRANSACTION statement, its current transaction automatically reverts to the nil state. This is also true if an attempted BEGINTRANSACTION fails.

Excluding a Server from a TMF Transaction

There are times when a requester must delegate work to a server and yet exclude that server from the bounds of a TMF transaction; it does so by either:

- Setting the current transaction to the nil state before sending the work request to the server
- Opening the server with a prior designation that the transaction identifier be omitted from all communication with the server

There are two reasons why a requester would want to exclude a server from a transaction:

- To enhance performance within a network environment
- To eliminate unnecessary dependencies between processes

Enhancing Performance in a Network Environment

When a work request associated with a transaction identifier is transmitted from a requester process on one node of a network to a server process on another node, TMF generates additional messages and internal overhead (when the transaction is being either committed or aborted) beyond that required for messages not associated with a transaction identifier.

If the remote server is one of several processes locking or updating records in a protected database as part of a single business function, then you will want the work of all the processes to be interrelated within the bounds of a single TMF transaction. In that scenario, the additional network overhead is not only unavoidable but is actually a positive factor because of the transaction-oriented database protection it helps provide.

If, however, the remote server manipulates the database independently of other processes, you should let the server initiate and terminate its own transaction for the work request. In that scenario, because the work request has no transaction identifier, you avoid unnecessary network overhead.

Eliminating Unnecessary Dependencies

The more processes that are explicitly involved in the processing of a transaction, the more opportunity there is for the transaction to be aborted. Consequently, you should minimize the number of server processes doing work under the same transaction identifier.

If work being delegated to a server process is logically external to the requester’s current transaction in that it does not involve protected portions of a database, you should exclude that work from the particular transaction identifier.

For example, suppose a requester needs to send a message to the spooler. Requests to the spooler occasionally time out. If that were to occur within the bounds of a transaction, the entire transaction would be aborted and all of the work associated with it would be backed out.

Because the spooler does not use audited database files, there is nothing to be gained by including its activities within a TMF transaction.

By setting the current transaction to the nil state before issuing a WRITEREAD system procedure call to the spooler, the requester’s transaction will not abort if a timeout occurs for the WRITEREAD system procedure.

Setting the Current Transaction to Nil

A requester or server process sets its current transaction to the nil state by issuing a RESUMETRANSACTION call with a tag value of zero. In TAL, the call is as follows:

```
status := RESUMETRANSACTION (0D);
```

After setting the current transaction to the nil state, however, you must then explicitly reset it back to the transaction identifier of a currently active transaction. To do that, you need to know the proper tag value for each transaction.

Within a requester, the tag for each transaction is returned by way of the *trans-begin-tag* variable in the BEGINTRANSACTION TMF procedure call. Requesters set their current transaction to a particular transaction by supplying the appropriate tag value in a RESUMETRANSACTION TMF procedure call. In TAL, the call is as follows:

```
status := RESUMETRANSACTION (tag);
```

Within a server, you obtain the tag for each transaction by issuing a LASTRECEIVE TMF procedure call immediately after each READUPDATE of $RECEIVE. Servers set their current transaction to a particular transaction by supplying the appropriate tag value in an ACTIVATERECEIVETRANSID TMF procedure call. In TAL, the call is as follows:

```
CALL ACTIVATERECEIVETRANSID (tag);
```

Marking an OPEN to Not Share a Transaction Identifier

A requester process can also exclude server processes from sharing its transaction identifier by issuing a SETMODE 117 call with *param1* = 1 after opening the servers. This parameter setting is the default for process subtypes 30 and 31 (device simulators and spoolers, respectively).

For any processes opened by the requester after such a SETMODE call, a transaction identifier is never associated with messages sent from the requester to those processes, even if one is in effect for the requester at the time the message is sent. Thus, if a server terminates abnormally while working on behalf of a transaction, the transaction itself will not be aborted because its transaction identifier was never included in the requester's messages.

Once this SETMODE call is issued, it remains in effect until it is explicitly reset by issuing a SETMODE 117 call with *param1* = 0.

The SETMODE 117 action is local to the process in which it is executed; no notification is passed to any other process.

If this SETMODE is invoked within one process of a process pair, provision must be made within the backup process to either call CHECKSETMODE or to reexecute the SETMODE 117 call at the time of a takeover.

Consistency and Concurrency

In the OLTP environment, there are three fundamental criteria that you can use to judge the integrity of a business database:

- The data must accurately reflect the results of all business operations performed against the database. In particular, all related data items must be accurate with regard to one another. Every total field, for example, must always equal the sum of all the detail items that it represents.
- The data must not violate the fundamental operating precepts of the particular business. Within the context of a financial institution, for example, the accrued debt against a line of credit must not exceed the customer's limit.
- Access to the same data by different program modules must result in the same view of that data.

When a database satisfies all of these criteria, it is said to be in a consistent state.

The job of maintaining database consistency can be divided into three tasks.

First, each application program module must be coded to properly change all pertinent data values and must be designed to do so in accordance with the rules of the applicable business environment. As an application designer or programmer, this is your responsibility.

Second, if a hardware or software failure occurs in the middle of a transaction, all of the items in the database that have been altered by the aborted transaction must be restored to their prior values. TMF does this for you. All you must do is properly group related database operations into TMF transactions.

Finally, when different processes are accessing the same database concurrently, they must be prevented from performing conflicting operations that distort one another's interpretation of the data. This is accomplished through the use of various types of locks on the data. For NonStop SQL/MP objects, you merely specify appropriate access options and NonStop SQL/MP obtains and releases the locks for you. For Enscribe files, you obtain and release locks, either explicitly or implicitly, through the use of file-system procedure calls.

Achieving Maximum Consistency

To achieve the highest degree of database consistency, do as follows:

1. If you have the choice of creating your database files as either NonStop SQL/MP objects or Enscribe files, choose NonStop SQL/MP because it provides the more effective (ANSI standard) concurrency control.
2. Design your TMF transactions so that all reads, updates, insertions, and deletions related to a particular business operation are within the bounds of a single TMF transaction.

3. When accessing NonStop SQL/MP objects, specify REPEATABLE ACCESS for all database operations.
4. When accessing Enscribe files, obtain a file lock at the beginning of the transaction and then let the DP2 disk process release the file lock when the transaction commits.

Note that the use of file locks with Enscribe files severely limits the amount of concurrent access, thereby reducing throughput and increasing the response time for end users. In most cases, it is impractical to use Enscribe file locks.

Levels of Consistency

Consistency and concurrency are inversely proportional to one another; as one increases, the other decreases. When using NonStop SQL/MP, you balance consistency and concurrency through the use of access options.

NonStop SQL/MP includes three access options, each corresponding to a general level of consistency. The access option specified by a particular transaction determines how much that transaction will be insulated from the effects of other concurrently running transactions.

The three access options, and the corresponding levels of consistency, are as follows:

- REPEATABLE ACCESS—Level-3 consistency
- STABLE ACCESS—Level-2 consistency
- BROWSE ACCESS—Level-1 consistency

The consistency level of a transaction depends on the lowest level of consistency provided by any access option used in the transaction. For example, three REPEATABLE ACCESS options and one STABLE ACCESS option used in a transaction result in second-level consistency because of the one STABLE ACCESS option.

REPEATABLE ACCESS

REPEATABLE ACCESS, sometimes referred to as “level-3 consistency,” provides maximum insulation from other transactions. With REPEATABLE ACCESS enabled, a transaction can run as though it were the only one active in the system. The transaction will not read uncommitted changes made by other transactions. This guarantees that any data you read is both stable and consistent; the data is neither in transition nor will it be backed out after you have read it.

Furthermore, other transactions cannot alter any uncommitted changes made by the transaction. This guarantees you the opportunity to change the database from one consistent state to another consistent state. If your transaction has updated a set of detail items, subtotals, and totals so that they are consistent with one another, no other transaction can disturb that consistency.

Finally, other transactions cannot update, delete, or insert anything within the entire range of rows accessed by the transaction. This guarantees you the opportunity to reread previously read rows and see exactly the same data values.

You should specify REPEATABLE ACCESS for transactions that make business decisions based on values that they have obtained by first reading the database. In particular, if the decision is based upon the content of a range of rows, you do not want anything to change within that range (including the addition of new rows) until the transaction has made the appropriate decision and changed the database to a new consistent state.

The effect of REPEATABLE ACCESS on concurrency is usually negligible compared to that of STABLE ACCESS. If you are unsure which access mode to use, choose REPEATABLE ACCESS.

STABLE ACCESS

STABLE ACCESS, sometimes referred to as “level-2 consistency,” provides less stringent concurrency control than REPEATABLE ACCESS. With STABLE ACCESS enabled:

- The transaction will not read uncommitted changes made by other transactions.
- Other transactions cannot alter any uncommitted changes made by the transaction.
- Other transactions cannot update or delete the row in which the transaction’s cursor is currently positioned. If you are doing processing in which you first read a row and then update it, this guarantees that data within a row will not change between reading and updating.

The essential difference between STABLE ACCESS and REPEATABLE ACCESS is the continuing stability of the entire range of committed data that you access. STABLE ACCESS provides short-term stability for the row in which the transaction’s cursor is currently positioned, while REPEATABLE ACCESS provides long-term stability (from initial access through the end of the transaction) for the entire range of rows accessed by the transaction.

For many types of transactions, this difference is not really an issue; sometimes, however, it can be critical.

Consider, for example, the case in which an airline has canceled a particular flight due to mechanical problems and is rescheduling all existing reservations for that flight to another. Assume that the application has a NonStop SQL/MP table in which each row represents a reservation, and that the rows are in ascending order by the passenger’s last name. Within each row is a flight number field, which is also a nonunique index for accessing the table.

The problem for the application is to sequentially step through the table, changing the flight number in each row from the old one to the new one, without having additional reservations for the old flight number be added to that part of the table that has already

been updated. A likely sequence of SQL statements for performing this operation would be as follows:

```
UPDATE passenger
  SET flight-number = 701
  WHERE flight-number = 429
  FOR REPEATABLE ACCESS;
```

With both REPEATABLE and STABLE access, every row that has been updated is protected against being deleted or changed by any other active transaction.

With STABLE ACCESS, however, another transaction could insert a new passenger row into the processed portion of the table while the first transaction is still in progress. If that were to happen, the table will not end up in a consistent state. With REPEATABLE ACCESS, that type of conflict cannot occur because REPEATABLE ACCESS prevents other transactions from updating, deleting, or inserting anything within the entire range of rows that has been accessed.

There are other ways that an application could approach this particular problem, but the example clearly illustrates a major advantage of REPEATABLE ACCESS over STABLE ACCESS.

BROWSE ACCESS

BROWSE ACCESS, sometimes referred to as “level-1 consistency,” applies only to read operations, allowing you to read through any type of existing lock. This means that you can gather information without waiting for the results of other transactions to commit; you would use BROWSE ACCESS, for example, if you want to generate an interim report very quickly. However, it also means that the data you read might be in transition or might even disappear altogether if another transaction aborts.

BROWSE ACCESS provides the least impact on other active transactions in that it imposes no locks on the rows that you read.

Enscribe Capabilities

Both NonStop SQL/MP and Enscribe guarantee level-1 consistency: no other transaction can delete or update any rows or records that you have inserted, updated, or deleted until your transaction commits or is backed out.

Unlike NonStop SQL/MP, Enscribe does not provide access options but rather a set of locking mechanisms with which you can approximate level-2 or level-3 consistency.

With Enscribe, you use the following system procedures to lock and unlock files, sets of records (for key-sequenced files using generic locking), or individual records:

- LOCKFILE locks an entire file.
- LOCKREC locks the current record, as determined by the most recent operation against the file. If generic locking is enabled for a key-sequenced file, the LOCKREC procedure also locks any other records in the file whose keys begin

with the same character sequence as the key of the referenced record. Generic locking applies only to key-sequenced files.

- READLOCK and READUPDATELOCK lock the record before reading it.
- WRITE locks the record that is being inserted.
- UNLOCKREC unlocks the current record (as determined by the most recent operation against the file). If generic locking is enabled for a key-sequenced file, calls to UNLOCKREC are ignored.
- UNLOCKFILE, ENDTRANSACTION, ABORTTRANSACTION, and CLOSE unlock all records that were locked through the transaction identifier associated with the call.

Note that locks on records that were read but not altered are released immediately; locks on inserted, updated, or deleted records, however, are never released until the transaction is either committed or aborted and backed out.

Unlike NonStop SQL/MP, all locks for Enscribe files are exclusive locks. Enscribe does not support shared locks (locks whose ownership is claimed jointly by two or more transactions).

Ensuring Level-1 Consistency

There are three ways that the DP2 disk process ensures level-1 consistency for audited Enscribe files.

Whenever a transaction inserts a new record into an audited file, the disk process automatically obtains a lock based on the inserted record's primary-key value. This lock prevents another transaction from inserting a record with the same primary key value as the newly inserted record, and from reading, locking, updating, or deleting the newly inserted record.

Before a transaction can update or delete an existing record in an audited file, the transaction must first lock either the record or its file. If the transaction has not obtained an appropriate lock, the attempt to update or delete the record is rejected immediately with a condition code of CCL and an error 79.

The disk process retains all locks on inserted, updated, or deleted records in audited files until the associated transaction is either committed or aborted and backed out.

Locking Modes

Enscribe provides six locking modes that determine what action occurs if you try to lock a file, or read or lock a record, when the file or record is already locked. Table 1-1 summarizes each mode. You use SETMODE 4 procedure calls to select the desired locking mode.

Note that the Enscribe locking modes apply only to read and lock requests. If you issue a WRITE request (to insert a new record) and the target file is locked by a different transaction identifier, the request is rejected with a condition code of CCL and an error

73 (file/record is locked). If a record with the same primary-key value already exists, the WRITE request is rejected with a condition code of CCL and an error 10 (file/record already exists).

Table 1-1. Enscribe Locking Modes

Mode	*param1*	Description
Normal mode	0	Any attempt to lock a file, or to read or lock a record, that is already locked through a different transaction identifier is suspended until the existing lock is released. This is the default locking mode.
Reject mode	1	Any attempt to lock a file, or to read or lock a record, that is already locked through a different transaction identifier is rejected with a file system error 73 (file/record is locked); no data is returned.
Read-through/ normal mode	2	READ and READUPDATE requests ignore existing record and file locks; encountering a lock does not delay or prevent reading the record. LOCKFILE, LOCKREC, READLOCK, and READUPDATELOCK are treated as in normal mode.
Read-through/ reject mode	3	READ and READUPDATE requests ignore existing record and file locks; encountering a lock does not delay or prevent reading the record. LOCKFILE, LOCKREC, READLOCK, and READUPDATELOCK are treated as in reject mode.
Read-warn/ normal mode	6	READ and READUPDATE requests ignore existing record and file locks; although an existing lock will not delay or prevent reading the record, it will cause a CCG completion code with a warning code of 9. LOCKFILE, LOCKREC, READLOCK, and READUPDATELOCK are treated as in normal mode.
Read-warn/ reject mode	7	READ and READUPDATE requests ignore existing record and file locks; although an existing lock will not delay or prevent reading the record, it will cause a CCG completion code with a warning code of 9. LOCKFILE, LOCKREC, READLOCK, and READUPDATELOCK are treated as in reject mode.

The Inserted Record Problem

The use of REPEATABLE ACCESS with NonStop SQL/MP objects prevents other transactions from updating, deleting, or inserting anything within the entire range of rows accessed by a transaction.

Because this protection applies to a range of rows rather than to specific existing rows, NonStop SQL/MP has in effect also locked the available (but not yet used) slots

between existing rows. While the transaction is in progress, no new rows can be inserted within the entire range of rows accessed by the transaction.

Unless you lock an entire file, Enscribe does not provide this protection; therefore, applications using Enscribe files can encounter what is called “the inserted record problem.”

Consider the case in which a transaction is locking and reading a sequence of employee records arranged in alphabetic order, using a read loop containing a READLOCK call. When the transaction (designated T1) does its first read, the file is as follows:

```
Record #1:  ANN        (locked by T1)
Record #2:  BEN
Record #3:  CHARLIE
Record #4:  EDDIE
Record #5:  FRANCO
EOF
```

When the read loop terminates, the file is as follows:

```
Record #1:  ANN        (locked by T1)
Record #2:  BEN        (locked by T1)
Record #3:  CHARLIE    (locked by T1)
Record #4:  EDDIE      (locked by T1)
Record #5:  FRANCO     (locked by T1)
EOF
```

At this point, another transaction inserts the record DYLAN into the file. The second transaction can do so because T1 did not lock the entire file. The file now is as follows:

```
Record #1:  ANN        (locked by T1)
Record #2:  BEN        (locked by T1)
Record #3:  CHARLIE    (locked by T1)
Record #4:  DYLAN
Record #5:  EDDIE      (locked by T1)
Record #6:  FRANCO     (locked by T1)
EOF
```

If T1 were then to reexecute its read loop, it would not get the same results it did the first time.

The Deleted Record Problem

If you specify REPEATABLE ACCESS, NonStop SQL/MP prevents a transaction from reading past rows that have been deleted by other incomplete transactions (the read operation halts at the first deleted row until the delete is either committed or backed out).

With Enscribe files, the SETMODE 4 normal mode and reject mode options prevent individual reads from accessing deleted records. If you read an Enscribe file by using a read loop, however, the read sequence always bypasses any records that are marked as deleted (this is true even if the delete has not yet been committed); therefore,

applications using Enscribe files can encounter what is called “the deleted record problem.”

Consider the case in which one transaction (designated T2) is using a loop to read a sequence of employee records arranged in alphabetic order and another transaction (designated T2) has previously deleted one of the records.

When T2 starts its read loop, the state of the employee file is as follows:

```
  Record #1:  ANN
  Record #2:  BEN
[ Record #3:  CHARLIE ]      (deleted by T1)
  Record #4:  DYLAN
  Record #5:  EDDIE
  Record #6:  FRANCO
  EOF
```

As T2 loops through the file, it accesses the following sequence of records:

```
ANN
BEN
DYLAN
EDDIE
FRANCO
```

If transaction T1 aborts while T2 is reading the record DYLAN, for example, the delete operation is backed out and the file now is as follows:

```
Record #1:  ANN
Record #2:  BEN
Record #3:  CHARLIE
Record #4:  DYLAN
Record #5:  EDDIE
Record #6:  FRANCO
EOF
```

If T2 were then to reexecute its read loop, it would not get the same results it did the first time.

Summary of Levels of Consistency in Enscribe Files

When you are using TMF to protect Enscribe files, you are always guaranteed at least level-1 consistency.

You can approximate level-2 consistency by:

- Not using any of the SETMODE 4 read-through options (all read operations must wait for existing locks to be released)
- Using the READLOCK and READUPDATELOCK procedures instead of READ and READUPDATE, explicitly releasing the record lock (UNLOCKREC) after each read is complete

You can achieve level-3 consistency by locking the entire file at the start of your transaction and then letting the disk process unlock it implicitly as part of the

ENDTRANSACTION function. In almost any interactive environment, however, this practice is unacceptable because it precludes concurrent access.

You can approximate level-3 consistency by:

- Not using any of the SETMODE 4 read-through options
- Using the READLOCK and READUPDATELOCK procedures instead of READ and READUPDATE
- Letting the disk process unlock all records implicitly as part of the ENDTRANSACTION function

These practices do not prevent the inserted record problem or the deleted record problem. If either problem could occur in your particular application environment, then you can only achieve level-1 consistency.

2 Designing Single-Threaded Processes

Application program modules are often designed as single-threaded processes. Single-threaded requesters can only participate in one transaction at a time. Having initiated one transaction, a single-threaded requester cannot initiate another transaction until it has terminated the existing transaction.

Similarly, single-threaded servers can do work on behalf of only one transaction at a time. Having accepted a work request for one transaction, a single-threaded server cannot accept another work request until it has completed all of its work for the current request.

This section contains the following topics:

- Single-Threaded Requesters on page 2-1
- Single-Threaded Servers on page 2-11

Single-Threaded Requesters

Within the TMF programming environment, a requester is an application program module that initiates and terminates a transaction. Although the requester usually delegates work to one or more servers, and any of the servers can abort a transaction for any reason at any time, the requester always remains the owner of the transaction and is responsible for explicitly terminating it.

Applicable System and TMF Procedures

Single-threaded TMF requesters use the following system and TMF procedures:

- ABORTTRANSACTION—aborts the current transaction
- BEGINTRANSACTION—initiates a transaction and returns the value of its tag
- FILE_CLOSE_—closes the TFILE or a server process
- ENDTRANSACTION—terminates the current transaction
- FILE_OPEN_—opens the TFILE or a server process
- RESUMETRANSACTION (0)—sets the current transaction to the nil state

- RESUMETRANSACTION (*tag*)—resets the current transaction from the nil state to the transaction ID of the transaction identified by the specified BEGINTRANSACTION *tag* value
- WRITEREAD—sends a work request to a server process and accepts a reply from the server

Delegating Work to Servers

To subcontract a unit of work to a server process, the requester must:

1. Open the server process by using the FILE_OPEN_ system procedure.
2. Format a work request message.
3. Send the work request message to the server by using the WRITEREAD system procedure.

The FILE_OPEN_ call includes, as its first parameter, the server process name instead of a file name. All necessary process names, in local and network form, can be predefined so they are known throughout both the system and the network.

The requester and any servers it communicates with must all use the same work request message format.

The WRITEREAD call includes, as its first parameter, the file number returned by the FILE_OPEN_ call.

WRITEREAD returns a condition code upon completion. The condition code CCE (=) indicates successful completion. The condition code CCL (<) indicates that an error occurred. For CCL completions, you can obtain the file system error code by calling the FILEINFO system procedure. Table 2-1 lists those error codes that apply to WRITEREAD calls.

If you are using nowait WRITEREAD calls, the error notification is returned with the associated AWAITIO call.

Table 2-1. WRITEREAD Error Numbers (page 1 of 2)

Error Number	Meaning
40	For nowait WRITEREAD operations, this code indicates that the AWAITIO call timed out before the server could reply.
201	A server process or its processor module failed while the server was working on the transaction.
211	A work request could not be delivered to a server because the processor module containing the server had already failed.
240	A line handler failure occurred (the request did not get started).
241	A network failure occurred (the request did not get started).
246	A **fiber optic extension (FOX)** network direct route failure occurred (the request was terminated).

Table 2-1. WRITEREAD Error Numbers (page 2 of 2)

Error Number	Meaning
248	A line handler failure occurred (the request was terminated).
249	A network failure occurred (the request was terminated).
250	All paths to a required system are down (the request did not get started).
251	A network protocol error occurred (the request was terminated).
252	A required Expand class is not available (the request did not get started).
255	A line handler came up too often.

Terminating Transactions

A requester must always match every BEGINTRANSACTION call with a corresponding ENDTRANSACTION or ABORTTRANSACTION call. This is true even if another process aborts the transaction.

ENDTRANSACTION

Requesters use ENDTRANSACTION to terminate transactions. ENDTRANSACTION directs TMF to make permanent all changes that the transaction made to audited files. For single-threaded requesters, ENDTRANSACTION normally suspends the requester until the transaction is either committed or aborted.

Because ENDTRANSACTION is a function-type procedure, it returns a file system error code in the *status* variable. The code 0 indicates successful completion; a nonzero code indicates that the transaction could not be committed. For a complete list of the file system error codes that apply to ENDTRANSACTION calls, see Parameters under the description of ENDTRANSACTION on page 4-15.

ABORTTRANSACTION

After issuing a BEGINTRANSACTION call and before issuing a corresponding ENDTRANSACTION call, a requester can abort the transaction by calling the ABORTTRANSACTION procedure.

The requester can abort a transaction even if there are servers that still have work in progress on behalf of the transaction.

If issued within a requester, an ABORTTRANSACTION call provides the necessary match for the associated BEGINTRANSACTION call (the requester does not have to also issue an ENDTRANSACTION call).

ABORTTRANSACTION backs out all changes made to audited files on behalf of the aborted transaction. ABORTTRANSACTION also prevents any further changes from being made to audited files on behalf of the aborted transaction (or at least guarantees that any such changes will be undone shortly after they occur).

After being called, ABORTTRANSACTION returns control immediately to the requester. Because the disk process holds locks on the affected database records until after the relevant changes have been backed out, the requester can safely initiate a new transaction involving the same database files without fear of encountering inconsistent data. If the new transaction requires access to a database record whose content was altered by the aborted transaction, the record lock prevents access to that record until the changes have been successfully backed out.

Because ABORTTRANSACTION is a function-type procedure, it returns a file system error code in the *status* variable. The code 0 indicates successful completion (the call has initiated transaction abort); 76 indicates that the transaction was already in the process of ending; 75 indicate that the transaction could not be aborted. For a complete list of the file system error codes that apply to ABORTTRANSACTION calls, see Parameters under the description of ABORTTRANSACTION on page 4-3.

Unilateral Aborts

Once a transaction is initiated, either of the following conditions can occur:

1. One of the servers doing work on behalf of the transaction issues an ABORTTRANSACTION call.
2. TMF or the file system aborts the transaction.

These types of transaction aborts are referred to as unilateral aborts because the requester that initiated the transaction has no recourse—the transaction is unconditionally aborted.

Any of the following situations will cause a unilateral abort:

- A requester or its processor module fails.
- A server or its processor module fails.
- A disk process or its processor module fails.
- A transaction remains on the system longer than TMF can tolerate.
- A waited WRITEREAD to a server process times out before the server can reply.
- An AWAITIO call times out before a server can reply.
- A requester cancels a WRITEREAD to a server process.
- A requester closes a server process that has not yet replied to a WRITEREAD.
- A requester closes its TFILE while a transaction is still in progress.
- A server issues an ABORTTRANSACTION call.
- A server closes $RECEIVE before having replied to a work request message.
- An operator manually aborts the transaction.

- A network failure occurs.
- A transaction is pinning a file on the MAT, and 45% of the MAT fills during the transaction's lifetime.
- The backup process of a process-pair or its processor fails after the transaction has been checkpointed (see Placement of Checkpoints on page 2-7).

When and how a requester finds out that a transaction has been aborted depends upon what the requester is doing at any particular time and the exact timing of various events relative to one another:

- If the requester is using waited WRITEREAD calls, the error completion occurs for the WRITEREAD call (a condition code of CCL and one of the error numbers shown in Table 2-1).
- If the requester is using nowait WRITEREAD calls, the error completion occurs for the AWAITIO call.

When a transaction aborts while the requester is issuing WRITEREADs to servers or reading, updating, or locking audited files, the requester starts receiving error completions. As the transaction is being aborted, the error number is one of those between 90 and 97, and listed in Table 2-2. When the transaction is completely aborted, the error number changes to 78 (invalid transaction ID).

When a transaction is unilaterally aborted by the system, the TFILE entry for the transaction still exists. To clear the entry from the TFILE, the transaction initiator must issue an explicit request to end or abort the transaction.

When the transaction initiator calls RESUMETRANSACTION or ENDTRANSACTION, the error returned depends on the cause of the abort, as reflected in the following table. For example, Error 93 is returned when the transaction is aborted because it spans more than 45% of the MAT capacity. The transaction initiator must either end or abort the transaction.

Table 2-2. Unilateral Abort Error Numbers (page 1 of 2)

Number	Cause of Error
90	BeginTransaction CPU failed.
92	Network-related problems occurred.
93	Transaction spanned more than 45% of the MAT capacity.
94	Operator initiated abort.

Table 2-2. Unilateral Abort Error Numbers (page 2 of 2)

Number	Cause of Error
95	One of the following occurred: • Audit loss • Backup disk process was not available for takeover • Backup disk process failed to takeover • Primary disk process failed because of internal error and backup disk process failed to takeover for this specific transaction • Double CPU or disk process failure
96	Transaction duration surpassed autoabort timeout interval
97	One of the following occurred: • Application requested abort • File system initiated abort • Server process involved in transaction failed • Unknown miscellaneous error occurred
1116	NonStop SQL/MP subsystem initiated abort

Checkpointing Strategy

Within the TMF programming environment, requesters can be coded as NonStop process pairs so that the backup process will take over automatically if the primary process should fail.

A full discussion of the design of NonStop process pairs is beyond the scope of this manual. The topics that follow do, however, present some requirements that TMF imposes upon the use of checkpoints and a strategy for the most effective placement of checkpoints within the primary requester process.

The Transaction Pseudofile (TFILE)

Every TMF requester process has its own transaction pseudofile, called a TFILE. The TFILE is not a physical I/O device file; it is never the target of actual I/O operations. Instead, the TFILE's access control block (ACB) serves as a mechanism whereby TMF can represent the state of transactions in a form that resembles a file.

Each entry in a TFILE corresponds to a single transaction and includes the transaction identifier and a status descriptor. The status descriptor is used at the completion of the transaction to indicate whether the transaction was committed or aborted.

For single-threaded requesters, the TFILE has only one entry: when the primary requester process issues a BEGINTRANSACTION call, the entry in the TFILE for the preceding transaction is overwritten by the information about the new transaction.

For single-threaded requesters that are not using checkpoints, the file system automatically creates a TFILE that contains one entry when the requester first calls

BEGINTRANSACTION. Such requesters can ignore the existence of the TFILE altogether.

If a requester is the primary process of a NonStop process pair, however, it must explicitly open the TFILE and then execute a CHECKOPEN call to create the backup requester's copy of the TFILE.

To open the TFILE, a single-threaded requester uses an FILE_OPEN_ call of the following form (illustrated in TAL).

```
CALL FILE_OPEN_ ( filename:4 , filenumber , , ,
                                          nowait-depth );
```

filename

is the name of a variable containing the logical device name of the TMF Management Process (TMP). The name is always $TMP. You can obtain the name programmatically by calling the GETTMPNAME procedure.

filenumber

is the name of a variable into which the FILE_OPEN_ system procedure returns a unique value identifying this instance of the TFILE. You use *filenumber* in CHECKPOINT calls to synchronize the content of the backup requester's TFILE with that of the primary requester's TFILE.

filenumber may be used only in calls to AWAITIO[X], CLOSE, and FILE_CLOSE_ or as a filenumber in a checkpoint procedure call. Using *filenumber* with any other file system procedures will fail with several different file error codes.

nowait-depth

is the name of a variable containing the FILE_OPEN_ *nowait-depth* parameter. For a single-threaded requester, this parameter is either a 0 or a 1 (both specifying that the requester can have only one transaction in existence at any given time). Zero (0) indicates that ENDTRANSACTION calls are to be waited, while 1 specifies that they are to be completed by an AWAITIO call (nowait).

A process can open the TFILE only once; attempts to open multiple instances of the TFILE, or to open the TFILE after having called BEGINTRANSACTION, will fail with a condition code of CCL and a file system error number 12 (file in use).

Placement of Checkpoints

If a primary requester process fails while a transaction is in progress, TMF automatically aborts the transaction. If the backup requester process fails while a transaction is in progress, and the transaction has been checkpointed, TMF automatically aborts the transaction.

The processing of a transaction never transfers in midstream from the primary requester to the backup: the backup requester, upon sensing a failure of the associated primary process, either restarts the aborted transaction and performs it in its entirety (if there was a transaction in progress at the time of the failure) or moves on to the next transaction (if the failure occurred between transactions).

The primary process periodically issues calls to the CHECKPOINT system procedure. Depending upon what parameters you supply in the CHECKPOINT call, the CHECKPOINT procedure causes either or both of the following actions to occur:

- TMF updates the backup requester's TFILE to synchronize its content with that of the primary requester's TFILE.
- The file system passes data from the primary requester's data stack and any combination of up to 13 separate data blocks and file synchronization blocks to the backup requester. Data blocks are usually file buffers that are not checkpointed as part of the stack and they can be from any location within the user data area (but not an extended data area).

Instead of CHECKPOINT, there are three other checkpointing procedures that you will sometimes need to use:

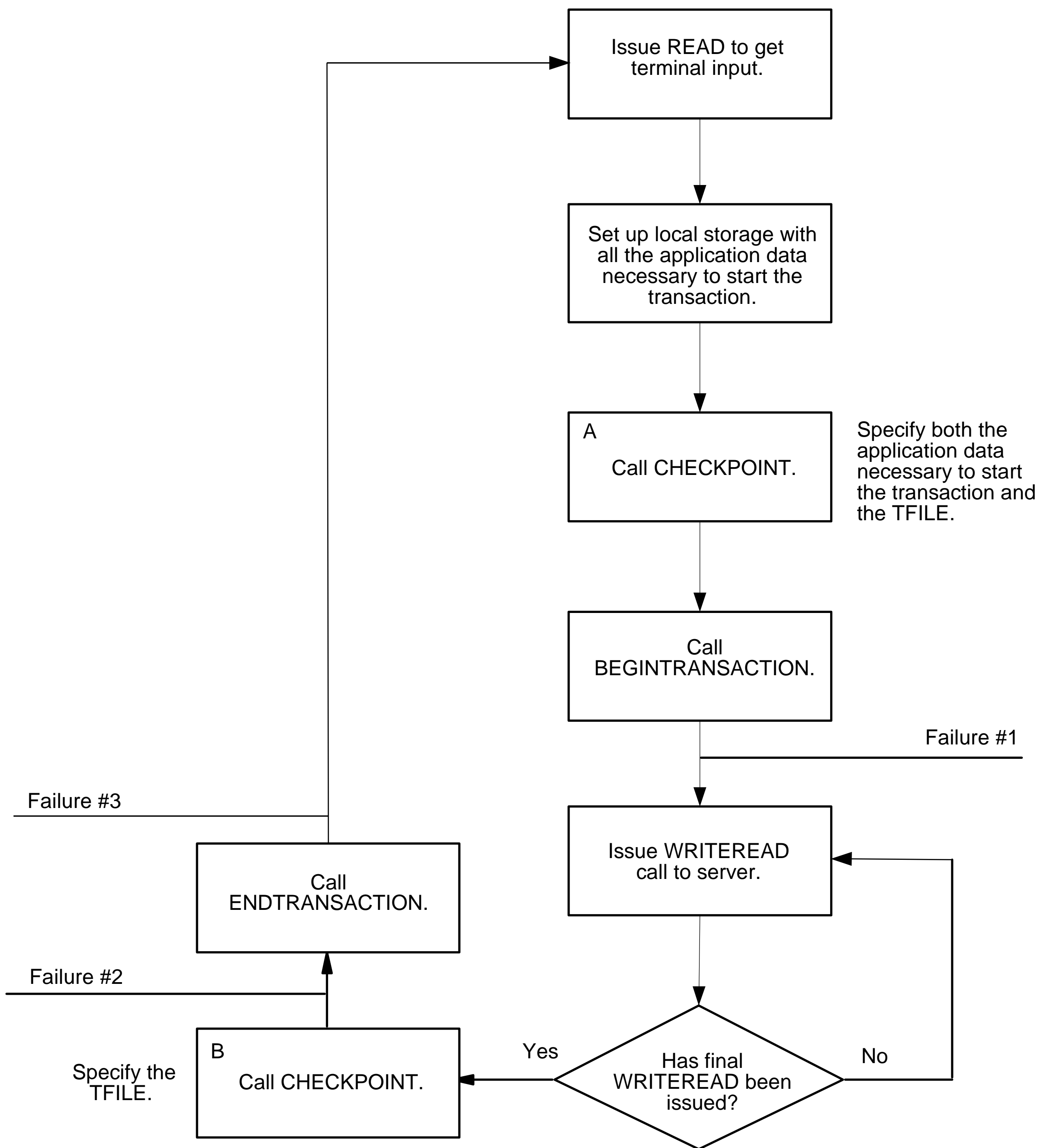
- You use the CHECKPOINTMANY procedure when you need to checkpoint more than 13 items from the user data area.
- You use the CHECKPOINTX procedure when you need to checkpoint items from an extended data area (up to five items).
- You use the CHECKPOINTMANYX procedure when you need to checkpoint more than five items from an extended data area.

Note that one of the pieces of information that must be in the data stack or one of the data blocks is the *tag* value returned by BEGINTRANSACTION. In the event of a primary-to-backup switch, the backup process might need that value to issue a RESUMETRANSACTION call.

Whenever the primary process issues a CHECKPOINT call, the backup process must issue a call to the CHECKMONITOR system procedure. This is often done in the form of a loop in which a single CHECKMONITOR call is issued repeatedly until a primary-to-backup switch occurs; when a switch occurs, control falls through the loop to the takeover code.

Figure 2-1 illustrates the proper placement of CHECKPOINT calls within the transaction processing loop of a single-threaded primary TMF requester process.

Figure 2-1. Checkpointing Within Single-threaded Requesters



The procedure call for checkpoint A occurs when the primary process is about to initiate a transaction but has not yet issued the BEGINTRANSACTION call. At that point, the backup requester's TFILE is updated to indicate that there is no current transaction. All of the application data necessary to start the new transaction is present in the data stack and data blocks that are copied to the backup process by the CHECKPOINT procedure.

The procedure call for checkpoint B occurs when the primary process is about to commit the transaction but has not yet issued the ENDTRANSACTION call. At that point, the backup requester's TFILE is updated to reflect the current transaction.

If the backup process is executing a single CHECKMONITOR call repeatedly, you might want to include a program variable in the checkpointed data that serves as a toggle switch identifying the current checkpoint (A or B); in the event of a primary-to-backup switch, the backup process would then know immediately at what point in the primary requester's transaction loop the failure occurred.

When the primary fails, the CHECKMONITOR loop in the backup process passes control to a section of takeover code. That code must first determine whether or not a transaction was in progress when the primary failed.

Consider the effects of a primary process failure at points 1, 2, and 3 in the transaction processing loop illustrated in Figure 2-1.

If the failure occurs at point 1, the backup process detects by the occurrence of checkpoint A that the primary was about to initiate a transaction and that the transaction either did not get started or was initiated and then aborted because of the failure. Therefore, without further investigation, the backup (now acting as a primary) starts a new backup process and then restarts the transaction using the checkpointed application data.

If the failure occurs at either point 2 or 3, the backup process knows by the occurrence of checkpoint B that the processing of a transaction was complete when the failure occurred but does not know whether the transaction was committed or aborted. The backup determines the outcome of the transaction by fetching the transaction's BEGINTRANSACTION *tag* value from the checkpointed data and then issuing a RESUMETRANSACTION call containing that *tag* value, followed by an ENDTRANSACTION or an ABORTTRANSACTION call.

If the transaction may have committed before the failure occurred, the RESUMETRANSACTION call returns the error number 76 (transaction ended) in the *status* variable. In this case, the backup (now acting as a primary) should call ENDTRANSACTION. If ENDTRANSACTION returns no error, the transaction is committed, and the application starts up a new backup process and then issues a READ to the terminal to accept the next operator request. If ENDTRANSACTION returns an error, the transaction is aborted, and the application starts a new backup process and restarts the transaction using the checkpointed application data.

If the transaction was aborted, however, the RESUMETRANSACTION call returns one of the errors 90 through 97 described in Table 2-2. In this case, the backup (now acting

as a primary) should call ABORTTRANSACTION, start a new backup process, and then restart the transaction using the checkpointed application data.

If RESUMETRANSACTION returns no error, TMF has not yet sufficiently processed the failure of the primary process to indicate that the transaction aborted. The application should call ABORTTRANSACTION, start a new backup process, and restart the transaction using checkpointed application data.

Single-Threaded Servers

Within the TMF programming environment, a server is an application program module that manipulates a database in response to work requests from a requester.

A single-threaded server can work on behalf of only one transaction at a time: having accepted a work request for one transaction, a single-threaded server cannot accept another work request until it has completed all of its work for the current request.

Note. In addition to the transaction inherited from a requester, a single-threaded server can initiate a new transaction by issuing a BEGINTRANSACTION call. If, however, the server attempts to issue a second BEGINTRANSACTION call without first issuing either an ENDTRANSACTION or ABORTTRANSACTION call, an error 83 occurs.

Applicable System Procedures

Single-threaded TMF servers use the following file system procedures:

- ABORTTRANSACTION—aborts the current transaction
- CLOSE—closes $RECEIVE or a subordinate server process
- LASTRECEIVE—obtains the *tag* value associated with a work request (this procedure call must immediately follow the READUPDATE call that accepted the particular work request)
- FILE_OPEN_—opens $RECEIVE or a subordinate server process
- READUPDATE—accepts an incoming work request from a requester
- REPLY—sends a reply message to a requester
- RESUMETRANSACTION (0)—sets the current transaction to the nil state
- ACTIVATERECEIVETRANSID (*tag*)—resets the current transaction from the nil state to the transaction identifier of the work request identified by the specified *tag* value
- WRITEREAD—sends a work request to a subordinate server process and accepts a reply from the server

Opening $RECEIVE

To open the $RECEIVE pseudofile, a single-threaded server uses an FILE_OPEN_ call of the following form (illustrated in TAL).

```
CALL FILE_OPEN_ ( filename:8 , filenumber , , ,
                                nowait-depth , 1 );
```

filename

specifies the character string $RECEIVE.

filenumber

is the name of a variable into which the FILE_OPEN_ procedure returns a unique value identifying the $RECEIVE pseudofile. You use *filenumber* in READUPDATE and REPLY calls to accept incoming messages and reply to the associated requesters, respectively.

nowait-depth

is the name of a variable containing the FILE_OPEN_ *nowait-depth* parameter. For a single-threaded server, this parameter is either a 0 or a 1. Zero (0) indicates that I/O operations are to be waited, while 1 specifies that I/O operations are to be completed by an AWAITIO call (nowait).

1

is the receive-depth value, specifying that the server can have only one incoming message in $RECEIVE at any given time. Note that if multiple incoming messages arrive within the same time period, none will be lost. The receive-depth of *1* means that, having read one message through a READUPDATE call, the server cannot read the next message until it has issued a reply to the first message.

A process can open $RECEIVE only once; attempts to open multiple instances of $RECEIVE will fail with a condition code of CCL and the error number 12 (file in use).

You close $RECEIVE by specifying *filenumber* in a CLOSE procedure call. Note that if the server has a transaction in progress (a work request message in $RECEIVE for which a reply has not yet been sent) when you close $RECEIVE, TMF automatically aborts the transaction associated with the work request.

Matching Each READUPDATE With a REPLY

Servers accept incoming work requests by way of the message system and the $RECEIVE file.

If the incoming message is a work request from a requester, the server must issue a corresponding reply message when it has finished doing the assigned tasks. If the server fails to reply, the only possible outcome for the associated transaction is

abortion: a requester cannot commit a transaction until all servers doing work under the transaction identifier have issued replies.

There are some types of interprocess messages—notably status messages sent by the operating system—that do not require a reply. Because there is no way to know in advance what type of message is in $RECEIVE, TMF servers must always use READUPDATE to accept incoming messages. If a particular message does not require a reply, the server completes the READUPDATE by issuing a REPLY call containing no parameters.

WRITEREAD to Another Server

A server can also subcontract work to other servers. The server does so in the same way as a requester: open the subordinate server, send a work request, and eventually close the server.

If a server is communicating with subordinate servers by using nowait WRITEREAD calls, however, the server must not reply to its own requester until all subordinate servers have issued their replies. If a server tries to issue a reply before having received replies from all of its subordinate servers, the file system rejects the REPLY call with a condition code of CCL and an error number 81 (nowait I/O pending).

The server's current transaction, if not preset to the nil state, is passed with the work request to the subordinate server. Note that a server sets its current transaction to the nil state exactly as a requester does—by issuing a RESUMETRANSACTION (0) call. Unlike requesters, however, a server restores the current transaction to the transaction identifier of an active transaction by issuing an ACTIVATERECEIVETRANSID (*tag*) call, where *tag* is the value obtained by using the LASTRECEIVE system procedure.

The Use and Implications of ABORTTRANSACTION

At any time after doing a READUPDATE of $RECEIVE and before issuing a corresponding reply, a server can abort the current transaction by issuing an ABORTTRANSACTION call.

ABORTTRANSACTION backs out all of the changes that were made to audited files on behalf of the aborted transaction. ABORTTRANSACTION also prevents any further changes from being made to audited files on behalf of the aborted transaction (or at least guarantees that any such changes will be undone shortly after occurring).

A server can abort the transaction even if there are subordinate servers that still have work in progress on behalf of the transaction.

ABORTTRANSACTION returns control to the server immediately. Even though the server has aborted the transaction, the server must still reply to its requester. A call to ABORTTRANSACTION merely aborts the current transaction; it does not complete the requester's WRITEREAD call, which is waiting for a reply.

A call to ABORTTRANSACTION automatically resets the server's current transaction to the nil state.

A server is only allowed to call ABORTTRANSACTION when it is participating in an active transaction: that is, when the server's current transaction is not nil. Once the server issues a reply to the requester, the server no longer has the ability to abort the transaction. If a server calls ABORTTRANSACTION when the current transaction is in the nil state, the call completes with an error number 75 (no transaction identifier) in the *status* variable.

Backout and Volume Recovery Anomalies

Anomalies occur when the backout process undoes certain changes made by an aborting transaction, or when volume recovery undoes certain changes made by an aborted or incomplete transaction.

- The insertion of a record at the end of an unstructured file is not undone. The inserted record stays in place, and the end-of-file (EOF) is unchanged. This procedure preserves records that other transactions might have written to the end of the file after the aborting transaction.
- The insertion of a record into an entry-sequenced file is undone by rewriting the record with a length of 0 bytes. This procedure preserves the positioning of records that other transactions might have inserted into the file after the aborting transaction.
- The insertion of a record at the end of a relative file is undone by deleting the record, but leaving the EOF unchanged. This procedure preserves records that other transactions might have written to the end of the file after the aborting transaction.

The Implications of REPLY

A reply message from a server indicates that the server is through with the work request. Furthermore, unless the server has previously called ABORTTRANSACTION, a reply message signifies that the server has agreed to commit the transaction. The server will not be notified of the transaction's eventual outcome.

When a single-threaded server issues a reply, the server's current transaction reverts to the nil state. If the server attempts to lock or change the content of an audited file while the current transaction is in the nil state, the file system rejects the particular I/O request with a condition code of CCL and an error number 75 (no transaction identifier). This situation will persist until the server either accepts another work request from $RECEIVE or (acting as a requester) initiates a new transaction by issuing a BEGINTRANSACTION call.

NonStop Servers

In the TMF environment, there is no need for NonStop servers.

If a server process fails while working on one or more transactions, TMF automatically aborts the affected transactions. When the associated requesters learn that their

transactions have been aborted, the requesters open a new instance of the failed server and then restart each transaction from the beginning.

As a result of this fundamental design characteristic of TMF, the processing of a failed server simply cannot be taken over in midstream by a backup process, and the use of checkpoints within a server is therefore meaningless.

Guarantees to Servers

TMF makes the following guarantees to server processes:

- Until the server issues a reply to the requester, the current transaction is not committed.
- All changes that a server has made to the database prior to issuing a reply participate in the eventual outcome of the transaction; the changes are either committed or backed out.
- If the server issues an ABORTTRANSACTION call, the current transaction unconditionally aborts and all changes made by anyone to the database on behalf of the aborted transaction are backed out.
- If a nonrecoverable hardware or software failure occurs while the server is working on the transaction, the transaction is unconditionally aborted and all changes made to the database on behalf of the aborted transaction are backed out.

Context-Sensitive Servers

Some TMF users have designed their applications to include multiple-message communication between a requester process and a particular server process. In that environment, the server process must maintain the context of a transaction throughout the interval spanned by successive interprocess messages. This type of server process is referred to as, “context-sensitive.” The use of context-sensitive servers requires that the requester observe certain constraints that can only be guaranteed through proper application design; TMF does not support this type of operation and cannot enforce the associated rules.

Suppose, for example, that a requester and server need to perform a transaction that requires five WRITEREAD-REPLY sequences. If fewer than all five sequences are performed, the transaction is not complete. According to the standard TMF rules, each time the server issues a reply, the server process is implicitly agreeing that the transaction can be committed. In a context-sensitive environment, however, that is true only for the final reply in the sequence.

A properly designed and coded requester ensures that the transaction gets aborted if anything prevents the completion of all five WRITEREAD-REPLY pairs. This includes a mechanism to detect a failure of the server process prior to the fifth reply, in which case the requester aborts the transaction, starts a new server using the old server’s name, and then restarts the entire transaction.

3

Designing Multithreaded Processes

Application program modules can be designed as multithreaded processes: for example, multithreaded requesters can have many transactions at the same time. Having initiated one transaction, a multithreaded requester can then initiate other transactions and switch from one transaction to another.

Similarly, a multithreaded server can work on many transactions at the same time. Having accepted a work request for one transaction, a multithreaded server can then accept work requests for other transactions and switch from one transaction to another.

A program module can even act simultaneously as a requester for one or more transactions and as a server for one or more other transactions.

Many concepts and characteristics of requesters and servers remain the same regardless of whether they are designed as single-threaded or multithreaded processes.

This section contains the following topics:

- Multithreaded Requesters on page 3-1
- Multithreaded Servers on page 3-12
- Multithreaded Requester/Server Processes on page 3-14

Multithreaded Requesters

Multithreaded requesters differ from single-threaded requesters in the following ways:

- Multithreaded requesters must always explicitly open their TFILE with more than one entry.
- Multithreaded requesters use nowait I/O (including nowait ENDTRANSACTION calls).
- Multithreaded requesters use a more complex checkpointing strategy.

These differences are detailed in the following subsections.

Opening the TFILE

To open the TFILE, a multithreaded requester uses FILE_OPEN_ call of the following form (illustrated in TAL).

```
CALL FILE_OPEN_ ( filename:4 , filenumber , , ,
                                          nowait-depth );
```

filename

is the name of a variable containing the logical device name of the TMF Management Process. The name is always $TMP. You can obtain the name programmatically by calling the GETTMPNAME procedure.

filenumber

is the name of a variable into which the OPEN procedure returns a unique value identifying this instance of the TFILE. You use *filenumber* in AWAITIO calls to recognize completions of nowait ENDTRANSACTION calls. In addition, if the requester is designed to run as a NonStop process pair, you also use *filenumber* in CHECKPOINT calls to synchronize the content of the backup requester's TFILE with that of the primary requester's TFILE.

filenumber may be used only in calls to AWAITIO[X], CLOSE, and FILE_CLOSE_ or as a filenumber in a checkpoint procedure call. Using *filenumber* with any other file system procedures will fail with several different file error codes.

nowait-depth

is the name of a variable containing the OPEN *nowait-depth* parameter. For a multithreaded requester, this parameter must be within the range 2 through 1000 (specifying the maximum number of transactions that the requester can have open concurrently).

A process can open the TFILE only once; attempts to open multiple instances of the TFILE, or to open the TFILE after having called BEGINTRANSACTION, will fail with a condition code of CCL and a file system error number 12 (file in use).

You close the TFILE by specifying *filenumber* in a CLOSE procedure call. Note that if there are any transactions in progress when you close the TFILE, TMF automatically aborts them.

Manipulating the Current Transaction

Multithreaded requesters must always include a variable name in the tag field of each BEGINTRANSACTION call. The BEGINTRANSACTION procedure returns a value, by way of that variable, that uniquely identifies each transaction.

After calling BEGINTRANSACTION, the newly initiated transaction automatically becomes the current transaction.

Thereafter, you use the returned tag values in RESUMETRANSACTION procedure calls to change the current transaction from one transaction to another.

As for single-threaded requesters, a successful call to ABORTTRANSACTION or ENDTRANSACTION automatically resets the current transaction to the nil state. Because a multithreaded requester is continually switching from one transaction to another, each portion of code that resumes the processing of a transaction must first set the current transaction to the proper value by calling RESUMETRANSACTION.

If the call to ABORTTRANSACTION or ENDTRANSACTION fails, the current transaction identifier remains unchanged and is the same identifier that was supplied to the procedure.

Nowait ENDTRANSACTION Calls

Because multithreaded requesters are designed to work on multiple transactions in an interleaved fashion, the requesters must be designed to use both nowait I/O and nowait ENDTRANSACTION calls. When the requester issues a nowait ENDTRANSACTION call, TMF does not suspend the requester but rather allows it to continue working on other transactions as the ENDTRANSACTION processing proceeds.

The current transaction identifier remains unchanged until the ENDTRANSACTION call completes successfully. Under certain circumstances governed by timing, nowaited READ operations executed immediately following ENDTRANSACTION calls can incorrectly report transaction id errors. To ensure correct behavior and error reporting, you should always issue a RESUMETRANSACTION(0d) call immediately after each ENDTRANSACTION call.

The requester uses the AWAITIO procedure to recognize completions of its nowait requests. Besides ENDTRANSACTION, the nowait requests can include WRITEREAD requests to servers, interprocess communication with other processes, and perhaps even updates to Enscribe database files.

When an AWAITIO call completes, it returns two pieces of information that identify the nowait call with which the completion is associated: values for *filenumber* and *tag*.

For WRITEREADs to servers, *filenumber* is returned by the OPEN procedure when the requester opened the particular process. For updates to Enscribe database files, *filenumber* is returned by OPEN when the requester opened the particular file.

For ENDTRANSACTION calls, however, *filenumber* is returned by OPEN when the requester opened its TFILE.

The tag value returned by AWAITIO identifies the particular transaction associated with the completed nowait operation; it is the same value that was returned by the BEGINTRANSACTION call that initiated the transaction.

Checkpointing Strategy

To describe the proper placement of checkpoints within multithreaded TMF requesters, the topics that follow examine the general flow of control within the sample requester illustrated in Figure 3-1, Figure 3-2 and Figure 3-3.

The sample requester accepts input from a set of terminals. The requester does that by displaying a data entry form on each terminal and then issuing nowait READ calls, one per terminal. After filling in the applicable fields on the form, the terminal operator presses a function key causing an AWAITIO completion and the initiation of a new transaction.

The sample requester responds to all error conditions by aborting the transaction and refreshing the READ to the affected terminal. The requester does not try to determine what type of error occurred or to make any decisions based on the error condition.

Individual Threads

As illustrated in Figure 3-1, the sequence of activities in a multithreaded requester is basically the same as that of a TMF transaction in a single-threaded requester. The major difference is that each thread in a multithreaded environment uses nowait procedure calls and, as a result, spends more time in a suspended state as the requester works on other concurrent transactions (threads).

The requester divides each thread into pieces, with each piece roughly corresponding to those sets of activities in Figure 3-1 that lie between the gray SUSPEND boxes. The requester does this asynchronously, essentially on a first come, first served basis; when an AWAITIO call completes, the requester uses *filenumber* and *tag* to determine what type of call completed and which transaction (thread) the completion is associated with. The requester then resumes processing for the particular transaction until the next point of suspension.

At any given time, only one thread is active; all others are temporarily suspended. Furthermore, each thread is at a particular point in its processing that is completely independent of all other threads. Within the requester itself, the threads have little impact on one another. Any contention between the threads occurs at the server level, where one transaction might have to wait momentarily for another transaction’s locks to be released.

Placement of Checkpoints

Figure 3-2 and Figure 3-3 together illustrate the flow of control within a primary multithreaded TMF requester process. Figure 3-2 shows the overall flow while Figure 3-3 shows the detailed functionality within each logical block of code.

Within the life of a transaction, there are three places at which information must be checkpointed:

1. Along with BEGINTRANSACTION, the primary process must issue a CHECKPOINT call to pass all application data necessary to restart the transaction to the backup process.
2. After BEGINTRANSACTION, and before issuing the corresponding ENDTRANSACTION call, the primary process must call CHECKPOINT to generate an entry for that transaction in the backup requester’s TFILE.
3. After either ENDTRANSACTION or ABORTTRANSACTION, the primary process must call CHECKPOINT to flush the transaction’s entry from the backup requester’s TFILE.

By properly placing your CHECKPOINT calls, however, you can actually accomplish all three of these checkpoint operations with just two calls.

As illustrated in the BEGINTRANSACTION box in Figure 3-2, a CHECKPOINT call following the BEGINTRANSACTION call can satisfy both items 1 and 2 in the preceding list. To do that, however, the parameter list of the CHECKPOINT call must include references to all of the following:

1. The local storage variable containing the filenumber of the TFILE
2. The list of active transactions
3. All blocks of application data that are necessary to restart the transaction

Figure 3-1. The Flow of an Individual Thread

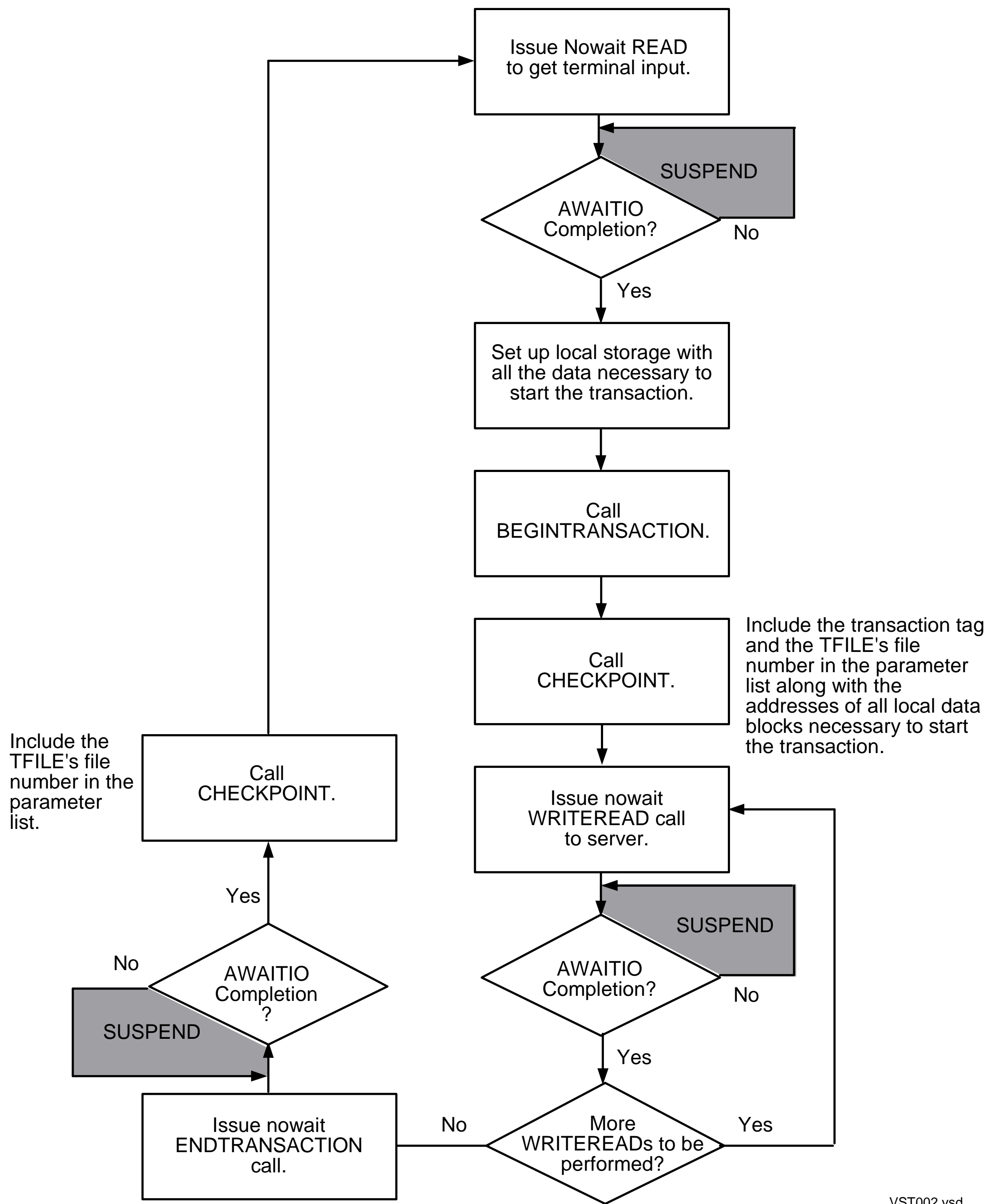


Figure 3-2. Multithreaded Requester Flow Chart

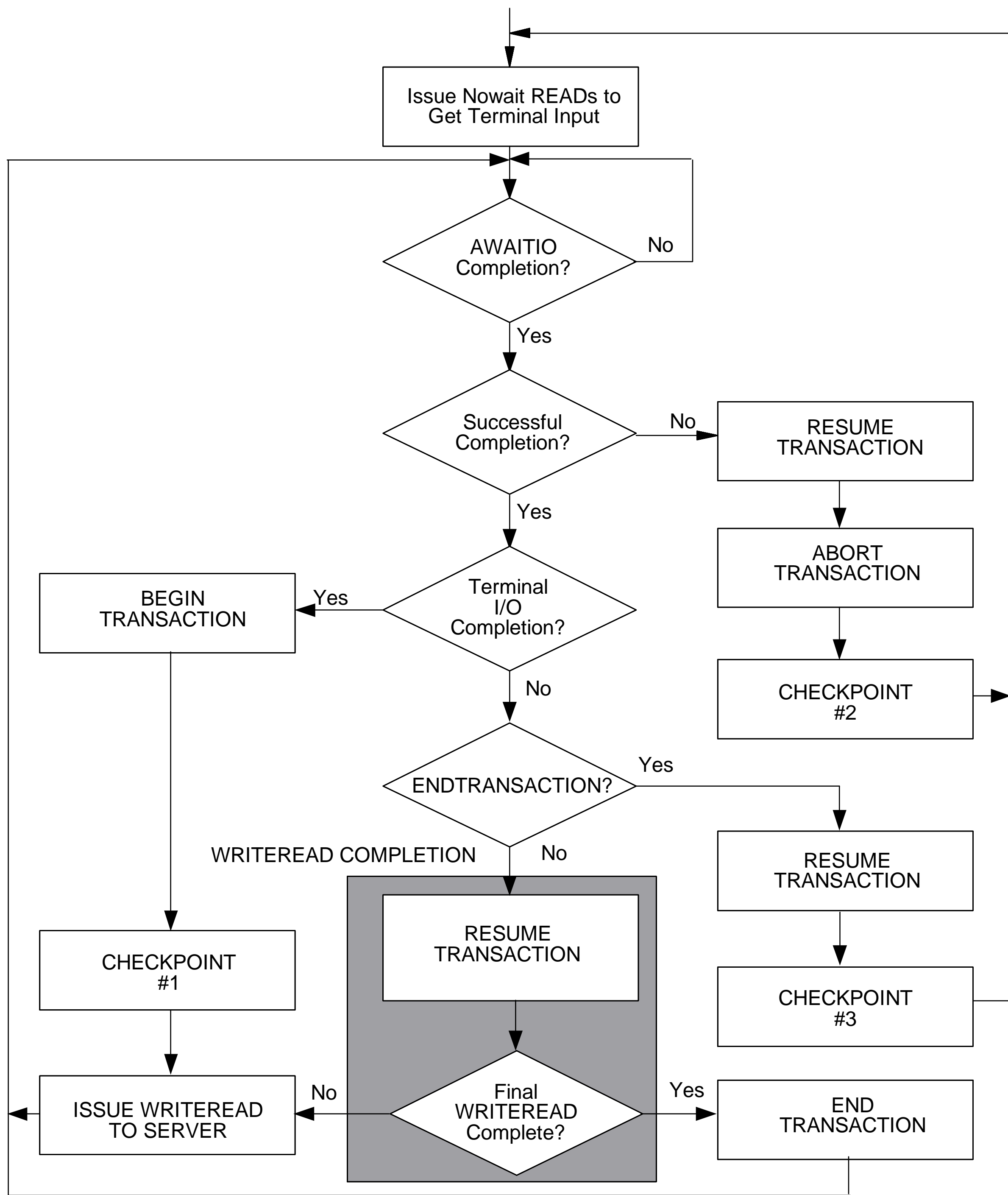
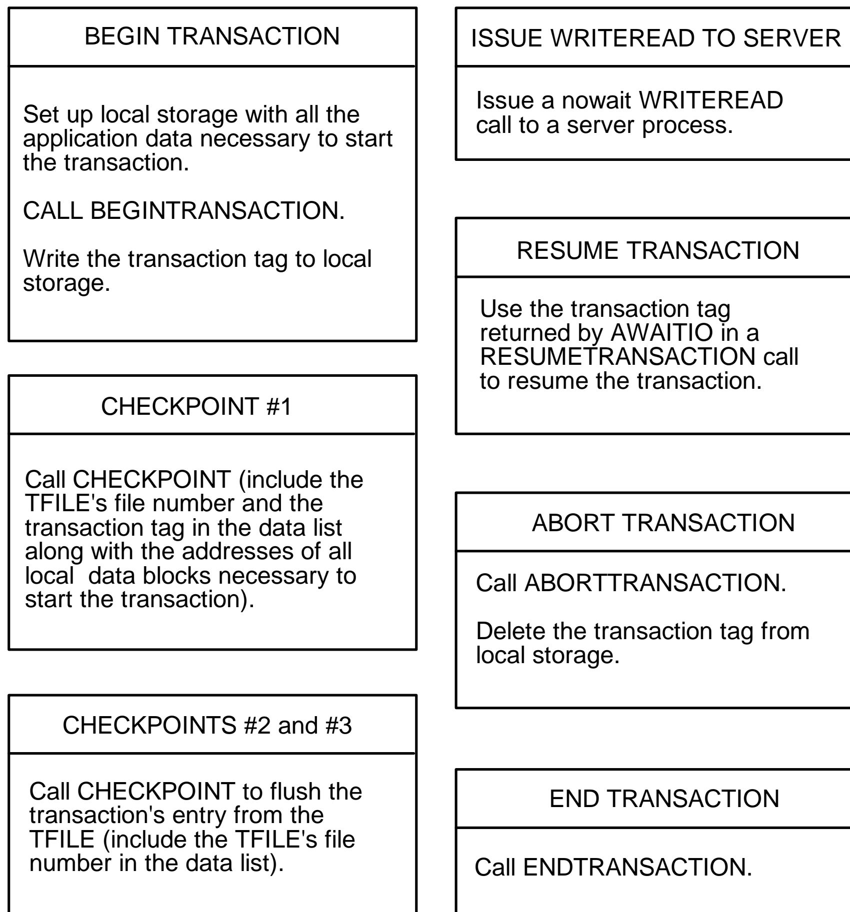


Figure 3-3. Multithreaded Requester; Detailed Functionality



Restarting Aborted Transactions on Takeover

If the primary requester fails for any reason, the backup process automatically takes over. When that happens, the backup process must determine which of the active transactions were successfully committed before the failure occurred, and which were aborted.

The processing of a transaction never transfers in midstream from the primary to the backup; any transactions that were aborted must be restarted and performed in their entirety.

One of the items that the primary requester must maintain in local storage is a list of the tag values associated with all active transactions. The primary process periodically passes the tag list to the backup process using a CHECKPOINT call. In general, it is most appropriate to do so by using the CHECKPOINT call that follows each BEGINTRANSACTION call.

When a primary-to-backup switch occurs, the backup process must determine which transactions were committed and which were aborted. The backup requester does this

by issuing a RESUMETRANSACTION call, followed by ENDTRANSACTION or ABORTTRANSACTION for each tag value in the tag list.

If a transaction may have committed before the failure occurred, the RESUMETRANSACTION call returns error number 76 (transaction ended) in the status variable. In that case, the backup process (now acting as a primary) should call ENDTRANSACTION. If ENDTRANSACTION returns no errors, the transaction is committed and the application starts a new backup process and then issues a READ to the terminal to accept the next operator request. If ENDTRANSACTION returns an error, the transaction is aborted. The application should start a new backup process and then restart the transaction using checkpointed application data.

If a transaction was aborted the RESUMETRANSACTION call returns one of the error numbers 90 through 97 described in Table 2-2 on page 2-5 of this manual. In this case, the backup process (now acting as a primary) should call ABORTTRANSACTION, start a new backup process, and then restart the transaction using the checkpointed application data.

If RESUMETRANSACTION returns no error, TMF has not yet sufficiently processed the failure of the primary process to indicate the aborted transaction. The application should call ABORTTRANSACTION, start a new backup, and restart the transaction using checkpointed application data.

The MAINLOOP Code

Initially, for each user terminal, the MAINLOOP code opens the terminal, displays the application data form on the terminal's screen, and issues a nowait READ call to detect when the operator presses a function key.

Thereafter, the MAINLOOP code continually checks for AWAITIO completions.

Upon detecting a successful AWAITIO completion, the MAINLOOP code determines what type of nowait call has completed:

- If the filenumber returned by the AWAITIO completion is that of a terminal, the MAINLOOP code passes control to the BEGINTRANSACTION code.
- If the filenumber returned by the AWAITIO completion is that of the TFILE, the completion is for a nowait ENDTRANSACTION call. In that case, control passes to the CHECKPOINTTFILEENTRY code.
- If the filenumber returned by the AWAITIO completion is that of a server process, the completion is for a nowait WRITEREAD call. In that case, the MAINLOOP code must determine whether or not this was the final WRITEREAD call to be issued for the transaction.

  If the completed WRITEREAD call is the final one for the transaction, control passes to the ENDTRANSACTION code.

  If the transaction requires another WRITEREAD call, control passes to the ISSUEWRITEREADTOSERVER code.

Upon return from either the ABORTTRANSACTION or CHECKPOINTTFILEENTRY code, the MAINLOOP code once again displays the application data form on the particular terminal's screen, issues a nowait READ for the terminal, and then continues checking for further AWAITIO completions.

The BEGINTRANSACTION Code

When the MAINLOOP code detects operator input, control passes to the BEGINTRANSACTION code. The major functions of the BEGINTRANSACTION code are as follows:

1. Save, in local storage, all of the application data necessary to start the new transaction.
2. Issue a BEGINTRANSACTION call to initiate the new transaction.
3. Add the transaction tag, returned by BEGINTRANSACTION, to a list of active transactions in local storage.
4. Issue a CHECKPOINT call. This call passes the startup data for the new transaction (saved in step #1) and the active transaction list (saved in step #2) to the backup requester and then generates an entry for the transaction in the backup requester's TFILE.
5. Pass control to the ISSUE WRITEREAD TO SERVER code.

The ISSUEWRITEREADTOSERVER Code

The ISSUE WRITEREADTOSERVER code formats a work request, sends the work request to the appropriate server process by issuing a nowait WRITEREAD call, and then returns control to the MAINLOOP code.

The CHECKPOINTTFILEENTRY Code

When the MAINLOOP code detects a successful AWAITIO completion for a nowait ENDTRANSACTION call, control passes to the CHECKPOINTTFILEENTRY code. The major functions of the CHECKPOINTTFILEENTRY code are as follows:

1. Issue a RESUMETRANSACTION call, specifying the tag value returned by the AWAITIO completion.
2. Issue a CHECKPOINT call (containing the filenumber of the TFILE in the parameter list) to flush the entry for the committed transaction from the backup requester's TFILE.
3. Delete the transaction tag from the list of active transactions in local storage.
4. Return control to the MAINLOOP code.

The ENDTRANSACTION Code

When the MAINLOOP code receives indication that the final nowait WRITEREAD call for a transaction has completed successfully, control passes to the ENDTRANSACTION code. The major functions of the ENDTRANSACTION code are as follows:

1. Issue a RESUMETRANSACTION call, specifying the tag value returned by the AWAITIO completion.
2. Issue an ENDTRANSACTION call to commit the transaction.
3. Return control to the MAINLOOP code.

The ABORTTRANSACTION Code

If the MAINLOOP code detects an unsuccessful AWAITIO completion, control passes to the ABORTTRANSACTION code. The major functions of the ABORTTRANSACTION code are as follows:

1. Issue a RESUMETRANSACTION call, specifying the tag value returned by the AWAITIO completion.
2. Issue an ABORTTRANSACTION call to abort the transaction.
3. Issue a CHECKPOINT call (containing the filenumber of the TFILE in the parameter list) to flush the entry for the aborted transaction from the backup requester's TFILE.
4. Delete the transaction tag from the list of active transactions in local storage.
5. Return control to the MAINLOOP code.

Multithreaded Servers

Multithreaded servers differ from single-threaded servers in the following ways:

- Multithreaded servers must open the $RECEIVE pseudofile with *receive-depth* greater than 1.
- Multithreaded servers must include the appropriate transaction tag in all REPLY calls.

Opening $RECEIVE

To open the $RECEIVE pseudofile, a multithreaded server uses an OPEN call of the following form (illustrated in TAL):

```
CALL FILE_OPEN_ ( filename:8 , filenumber , , , nowait-depth
                                             , receive-depth);
```

filename

is the name of a variable containing the character string $RECEIVE.

filenumber

is the name of a variable into which the OPEN procedure returns a unique value identifying the $RECEIVE pseudofile. You will use that filenumber in READUPDATE and REPLY calls to accept incoming messages and reply to the associated requesters, respectively.

nowait-depth

is the name of a variable containing the FILE_OPEN_ *nowait-depth* parameter. For a multithreaded server, this parameter is either a 0 or a 1. Zero (0) indicates that I/O operations are to be waited, while 1 specifies that I/O operations are to be completed by an AWAITIO call (nowait).

receive-depth

specifies the maximum number of incoming work requests that the server can have received but not yet replied to at any one time. For multithreaded servers, *receive-depth* must be within the range 2-2047.

A process can open $RECEIVE only once; attempts to open multiple instances of $RECEIVE will fail with a condition code of CCL and the error number 12 (file in use).

You close $RECEIVE by specifying the returned filenumber in a CLOSE procedure call. Note that if the server has any transactions in progress (messages queued in $RECEIVE for which a reply has not yet been sent) when you close $RECEIVE, TMF automatically aborts them.

Manipulating the Current Transaction

Immediately after accepting an incoming work request from $RECEIVE, the server must call the LASTRECEIVE procedure to obtain the tag value associated with that work request.

After each READUPDATE completion, the transaction identifier associated with the particular incoming work request becomes the server's current transaction.

Thereafter, you use the tag value (obtained by using LASTRECEIVE) in ACTIVATERECEIVETRANSID procedure calls to change the server's current transaction from one transaction to another.

When you are done processing $RECEIVE requests and wish to return control to a transaction that you previously initiated, you merely supply the tag value from the associated BEGINTRANSACTION call in a RESUMETRANSACTION procedure call.

As with single-threaded servers, a successful call to ABORTTRANSACTION automatically resets the current transaction to the nil state. Because a multithreaded server is continually switching its attention from one transaction to another, each portion of code that resumes the processing of a transaction must explicitly set the current transaction to the proper value by using an ACTIVATERECEIVETRANSID call.

If the call to ABORTTRANSACTION fails, the current transaction identifier remains unchanged and is the same identifier that was supplied to the ABORTTRANSACTION procedure.

Replying to Requesters

Whenever you have opened $RECEIVE with a receivedepth greater than 1, the REPLY procedure ignores the value of the server's current transaction and requires that you supply an explicit tag value in the procedure call.

NonStop Servers

In the TMF environment, there is no need for NonStop servers.

If a server process fails while working on one or more transactions, TMF automatically aborts the affected transactions. When the associated requesters receive indication that their transactions have been aborted, the requesters open a new instance of the failed server and then restart each transaction from the beginning.

As a result of this fundamental design characteristic of TMF, the processing of a failed server simply cannot be taken over in midstream by a backup process, and the use of checkpoints within a server is therefore meaningless.

Multithreaded Requester/Server Processes

You can design program modules that simultaneously act as a multithreaded requester for some transactions and as a multithreaded server for others.

These kinds of program modules must open both a TFILE and a $RECEIVE pseudofile (with *options* greater than 1 for the TFILE and *receive-depth* greater than 1 for $RECEIVE), but still have only one current transaction.

The TFILE *options* and $RECEIVE *receive-depth* parameters are completely independent of one another.

The TFILE *options* parameter specifies the maximum number of new transactions the program module can initiate (if the TFILE is opened with an *options* parameter of four, for example, the program module can issue up to four BEGINTRANSACTION calls without first issuing an ENDTRANSACTION or ABORTTRANSACTION call).

The $RECEIVE *receive-depth* parameter specifies the maximum number of transaction identifiers the program module can inherit from requester modules.

At any given time, the current transaction specifies one of the following:

- The transaction ID of a transaction in the TFILE
- The transaction ID of a work request message in $RECEIVE
- A nil value, indicating that there currently is no transaction in progress

The program module uses RESUMETRANSACTION (0) calls to set the current transaction to the nil state, RESUMETRANSACTION (*tag*) calls to set the current transaction to the transaction ID of a transaction in the TFILE, and ACTIVATERECEIVETRANSID (*tag*) calls to set the current transaction to the transaction ID of a work request message in $RECEIVE.

The program module must keep track of all the various tag values and use the proper procedure call (ACTIVATERECEIVETRANSID or RESUMETRANSACTION) for each individual tag value.

Note that a single program module could even act as both the requester and a server for the same transaction. Suppose that process A begins a transaction and does a WRITEREAD to process B. If process B then does a WRITEREAD back to process A, that very same transaction (which is already in process A's TFILE) could arrive in process A's $RECEIVE pseudofile as well. If that were to happen, the server component of process A must reply to process B (and process B must then reply to process A) before the requester component of process A can commit the transaction.

4 File System Procedures

This section presents detailed reference information for those file system procedure calls that pertain directly to TMF. The descriptions are in alphabetic order by procedure name.

For each procedure, the reference information includes:

- A brief description of what the procedure does
- The syntax of the procedure call (in both C and TAL)
- Parameter definitions
- Additional guidelines for using the call, where applicable

This section contains descriptions of the following procedures:

- ABORTTRANSACTION on page 4-3
- ACTIVATERECEIVETRANSID on page 4-5
- BEGINTRANSACTION on page 4-7
- BEGINTRANSACTION_EXT_ on page 4-10
- COMPUTETRANSID on page 4-12
- ENDTRANSACTION on page 4-15
- GETTMPNAME on page 4-17
- GETTRANSACTIONDETAILS on page 4-19
- GETTRANSID on page 4-24
- GETTRANSINFO on page 4-26
- INTERPRETTRANSID on page 4-28
- RESUMETRANSACTION on page 4-31
- STATUSTRANSACTION on page 4-34
- TEXTTOTRANSID on page 4-37
- TMF_BEGINTAG_FROM_TXHANDLE_ on page 4-40
- TMF_GETTXHANDLE_ on page 4-43
- TMF_GET_EXTTRANSID_ on page 4-44
- TMF_GET_TX_ID_ on page 4-46
- TMF_JOIN_ on page 4-48
- TMF_JOIN_EXT_ on page 4-50
- TMF_RESUME_ on page 4-53

- TMF_SETTXHANDLE_ on page 4-55
- TMF_SUSPEND_ on page 4-57
- TMF_SUSPEND_EXT_ on page 4-59
- TMF_TXBEGIN_ on page 4-61
- TMF_TXHANDLE_FROM_BEGINTAG_ on page 4-63
- TRANSIDTOTEXT on page 4-65
- TMF_VERSION_EXT_ on page 4-67

Note. For enabling 64-bit features, you can call the 64-bit APIs in the OSS environment using the **LP64** memory option. In the Guardian environment, call the 64-bit APIs using **Mix Mode**.

Mix mode programming uses 64-bit pointers generated by *SEGMENT_ALLOCATE64_* and the same pointers are used in the ILP32 memory model. This enables 64-bit APIs to be used in the Guardian environment.

ABORTTRANSACTION

This procedure aborts the current transaction (the transaction identified by the calling process' current transaction identifier). Any application process that is working on a transaction can abort that transaction at any time for any reason.

If a server process aborts a transaction, the requester (or its backup) that initiated the transaction must still explicitly terminate the transaction by calling either ENDTRANSACTION or ABORTTRANSACTION.

When ABORTTRANSACTION returns control to the calling process, the aborted transaction has not yet been backed out, but the transaction's state has changed from active to aborting. Later, the TMF backout process will back out the transaction by restoring the transaction's before-images to all affected disk files. When backout is complete, all locks held for the aborted transaction are released.

If the transaction is restarted under a new transaction identifier, the new transaction cannot access any records locked by the aborted transaction until transaction backout is completed for the aborted transaction and the locks held by the aborted transaction are released.

Syntax for C Programmers

Note. `This procedure can be called from 32-bit and 64-bit applications.`

```
#include <cextdecs(ABORTTRANSACTION)>
short ABORTTRANSACTION();
```

Syntax for TAL Programmers

```
status := ABORTTRANSACTION;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
30	System unable to obtain message block, or is already using its maximum number of RECEIVE or SEND message blocks.
75	Current transaction in nil state.

76	The requester called ABORTTRANSACTION while the transaction was already in the process of ending (the transaction therefore cannot be aborted or resumed).
82	TMF not running.
84	TMF not configured.
97	Transaction already aborted.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

ACTIVATERECEIVETRANSID

Servers use this procedure to set their current transaction to the transaction identifier of a particular work request queued in $RECEIVE. In the call, you supply a message tag that you initially obtained by calling the LASTRECEIVE procedure immediately after the READUPDATE call that accepted the message.

A multithreaded server uses ACTIVATERECEIVETRANSID either to resume processing for a particular transaction or to properly set the current transaction before replying to a requester. A single-threaded server uses ACTIVATERECEIVETRANSID to reset the current transaction from the nil state to the transaction identifier of the currently active transaction.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

```
#include <cextdecs(ACTIVATERECEIVETRANSID)>
_cc_status ACTIVATERECEIVETRANSID ( short message-tag );
```

Syntax for TAL Programmers

```
CALL ACTIVATERECEIVETRANSID ( message-tag );                  !i
```

Parameters

message-tag input

INT:value

is returned by LASTRECEIVE, identifying a particular work request queued in $RECEIVE. *message-tag* must be an integer between 0 and (*receive-depth* - 1), inclusive, that is currently associated with a work request queued in $RECEIVE.

ACTIVATERECEIVETRANSID returns a condition code only. For more information about error conditions, call the FILEINFO procedure specifying the *filenumber* of $RECEIVE.

The condition codes have the following meanings:

<< (CCL)	indicates that an error occurred (call FILEINFO).
= (CCE)	indicates successful completion.
>> (CCG)	indicates that $RECEIVE was not opened with a *receive-depth* equal to or greater than 1.

The function value returned by ACTIVATERECEIVETRANSID, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

BEGINTRANSACTION

This procedure initiates a new transaction. When you call this procedure, TMF creates a unique transaction identifier that distinguishes the new transaction from all other active transactions. The new transaction identifier becomes the current transaction of the calling process.

The form of the transaction identifier is as follows:

Transaction ID	Description
transid[0].<0:7>	contains 1 plus the Expand system number of the system in which BEGINTRANSACTION was called (this field contains 0 if the originating system is not part of a network). The system number identifies the **home node** of the transaction.
transid[0].<8:15>	contains the number of the processor in which the transaction originated.
transid[1:2]	contains a double-word sequence number to make the transaction identifier unique.
transid[3]	a number (zero or nonzero) representing flags used internally by TMF.

While the BEGINTRANSACTION procedure creates the transaction identifier, it does not return it during the course of the procedure. GETTRANSID returns the transaction identifier of the calling process' current transaction. For more information, see the description of GETTRANSID on page 4-24.

The value returned to the *trans-begin-tag* variable can be passed to the RESUMETRANSACTION procedure to restore to currency a transaction that was previously initiated by the calling process (or its backup).

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(BEGINTRANSACTION)>
short BEGINTRANSACTION ( [ long _near *trans-begin-tag ]);
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(BEGINTRANSACTION)>
short BEGINTRANSACTION ( [ int _ptr64 *trans-begin-tag ]);
```

Syntax for TAL Programmers

```
status := BEGINTRANSACTION ( [ trans-begin-tag ] );        ! o
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
30	System unable to obtain message block, or is already using its maximum number of RECEIVE or SEND message blocks.
82	TMF not running.
83	Too many transactions (a single-threaded requester tried to initiate a transaction while it still had one in progress, or a multithreaded requester attempted to initiate more concurrent transactions than there were TFILE entries)
84	TMF not configured.
86	Audit trail at *begin-trans-disable* threshold, or operator has disabled the BEGINTRANSACTION procedure.
98	TFILE was opened using a nonzero *sync-depth* and all transaction control blocks (TCBs) within the caller's CPU are currently occupied.

For a list of file system error numbers, see the *Operator Messages Manual*.

trans-begin-tag output

INT(32):ref

or

INT(32) .EXT64:ref

returns a value that identifies the new transaction, among other transactions, which the calling process has initiated. This parameter is optional for single-threaded requesters, and mandatory for multithreaded requesters.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

BEGINTRANSACTION_EXT_

This procedure initiates a new transaction. The procedure is the same as BEGINTRANSACTION except that it includes two new parameters to specify the transaction type, and a timeout value for the transaction.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(BEGINTRANSACTION_EXT_)>

short BEGINTRANSACTION_EXT_ ([ long _near *trans-begin-tag ],
                             [ long timeout ],
                               [ long long type_flags ] );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(BEGINTRANSACTION_EXT_)>

short BEGINTRANSACTION_EXT_ ([ int _ptr64 *trans-begin-tag ],
                             [ int timeout ],
                               [ long long type_flags ] );
```

Syntax for TAL Programmers

```
status := BEGINTRANSACTION_EXT_ ( [ trans-begin-tag ], ! o
                                  [ timeout ], ! i
                                  [ type_flags ] ); ! i
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
30	System unable to obtain message block, or is already using its maximum number of RECEIVE or SEND message blocks.
82	TMF not running.
83	Too many transactions (a single-threaded requester tried to initiate a transaction while a transaction is in progress, or a multithreaded requester attempted to initiate more concurrent transactions than there were TFILE entries).

84	TMF not configured.
86	Audit trail at *begin-trans-disable* threshold, or operator has disabled the BEGINTRANSACTION procedure.
98	TFILE was opened using a nonzero *sync-depth* and all transaction control blocks (TCBs) within the caller's CPU are currently occupied.
590	The parameter value is invalid.

For a list of file system error numbers, see the *Operator Messages Manual*.

trans-begin-tag output

INT(32) :ref

or

INT(32) .EXT64:ref

returns a value that identifies the new transaction, among other transactions, which the calling process has initiated. This parameter is optional for single-threaded requesters, and mandatory for multithreaded requesters.

timeout input

INT(32)

is the value in seconds that overrides the Auto Abort timer. A value 0 indicates use of Auto Abort timer. A value -1 indicates that the transaction will never time out.

The timeout parameter can have a value in the range of -1 through 21474836. This parameter is optional and if omitted, the configured AutoAbort value is used.

Type_Flags input

Fixed

is the transaction attribute. This parameter is optional.

The Transaction Typing parameter will be passed by the application to TMF and will be returned from TMF to the application as a FIXED (64-bit) value but will be treated as a structure within TMF.

```
STRUCT  Type_Flags_Struct (*) FIELDALIGN (SHARED2);
BEGIN
    Fixed           Flags[0:-1];

    UNSIGNED(8) User_Info;
      These bits may be set by the customer and are not the
     concern of TMF.
     Note: 8 bits allows 256 possibilities.  These bits will
     not be evaluated by TMF, they will simply be carried
     with the transaction.

    BIT_FILLER   53;
      Reserved bits for HP internal use. Should be
      initialized to zero.

    UNSIGNED(1) No_Capacity_Abort;
      If set to true, the transaction is not subject to TMF
      45% of audit-trail use limit.

    BIT_FILLER   2;
      Reserved bits for HP internal use. Should be initialized
      to zero.
END;
```

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

COMPUTETRANSID

This procedure converts the individual numeric parts of an externally formatted transaction identifier to internal form. All output parameters are undefined unless a status of 0 (successful) is returned. If omitted, the `tm-flags` parameter defaults to 0.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(COMPUTETRANSID)>

short COMPUTETRANSID ( long long _far *transid
                     , long system
                     , short cpu
                     , long sequence
                     , [ short tm-flags ] );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(COMPUTETRANSID)>

short COMPUTETRANSID ( long long _ptr64 *transid
                     , int system
                     , short cpu
                     , int sequence
                     , [ short tm-flags ] );
```

Syntax for TAL Programmers

```
status := COMPUTETRANSID ( transid                  ! o
                         , system                   ! i
                         , cpu                      ! i
                         , sequence                 ! i
                         , [tm-flags ] );           ! i
```

Parameters

status returned value

INT

is a TMF or file system error number:

-4	Invalid system (TMF).
-3	Invalid TMF flags value (TMF).
-2	Invalid CPU (TMF).
-1	Invalid sequence (TMF).
0	Successful completion.
22	Invalid reference parameter.
29	Missing parameter(s).
82	TMF not running.
84	TMF not configured.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

transid output

FIXED .EXT:ref:1 or INT .EXT:ref:4

or

FIXED .EXT64:ref:1 or INT .EXT64:ref:4

is the internal name of the transaction identifier.

system input

INT(32):value

is a required system number in the range 0 through 254.

cpu input

INT:value

is a required CPU number in the range 0 through 15.

sequence input

INT(32):value

is a required sequence number greater than 0.

tm-flags input

INT:value

is a number representing flags used internally by TMF.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

ENDTRANSACTION

This procedure commits the database changes associated with the current transaction. The only application process that can execute an ENDTRANSACTION call is the requester process that initiated the particular transaction.

When a requester calls ENDTRANSACTION, TMF attempts to commit the changes made to the database by the transaction. If the action completes successfully, the changes made by the transaction are permanent and the locks held for the transaction are released. Locks on inserted, updated, or deleted items are held until TMF has written a commit record to the audit trail.

If the calling process did not explicitly open its TFILE, calls to ENDTRANSACTION are waited operations: ENDTRANSACTION does not return control to the calling process until TMF has written a commit record to the audit trail, thereby ensuring that the transaction will commit. ENDTRANSACTION does not, however, wait for the transaction's locks to be released.

If the calling process explicitly opened its TFILE, calls to ENDTRANSACTION are nowait operations: ENDTRANSACTION returns control immediately to the calling process and the request must be completed by a subsequent call to the AWAITIO system procedure.

Syntax for C Programmers

Note. `This procedure can be called from 32-bit and 64-bit applications.`

```
#include <cextdecs(ENDTRANSACTION)>
short ENDTRANSACTION();
```

Syntax for TAL Programmers

```
status := ENDTRANSACTION;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
30	System unable to obtain message block, or is already using its maximum number of RECEIVE or SEND message blocks.
75	Current transaction in nil state.

78	Invalid or obsolete transaction identifier.
81	Outstanding nowaited I/O requests exist for the current transaction.
82	TMF not running.
84	TMF not configured.
90	Transaction's parent process failed.
91	The TMF subsystem failed and the transaction final outcome is unknown. When TMF is re-started, if the transaction has not been completely committed, it will automatically be aborted and all changes will be removed.
92	Path to a participating network node failed.
93	TMF aborted the transaction because the transaction remained on the system long enough to span too many audit-trail files.
94	Transaction aborted by operator command.
95	TMF aborted the transaction because a CPU failure caused a disk process primary-to-backup switch.
96	Transaction aborted by Autoabort threshold.
97	Transaction aborted by ABORTTRANSACTION call.
730	One or more processes are still joined to the transaction.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

GETTMPNAME

This procedure returns the logical device name of the TMF Management Process (TMP) defined during system configuration. You need to know the TMP name, which is always $TMP, in order to open the TFILE.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(GETTMPNAME)>
short GETTMPNAME ( short _near *devname );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(GETTMPNAME)>
short GETTMPNAME ( short _ptr64 *devname );
```

Syntax for TAL Programmers

```
status := GETTMPNAME ( devname );                              ! o
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
29	Missing parameter(s).
82	TMF not running.
84	TMF not configured.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

devname output

INT:ref:12

or

INT .EXT64:ref:12

is the name of a 12-word array into which GETTMPNAME returns the device name of the TMP. This name can then be passed to the OPEN procedure to open the TFILE. To call the FILE_OPEN_ procedure, pass a byte-addressable array that has the same location as *devname* along with a length value of 4. If the TMP is not configured, the returned device name is all blanks.

A process can open the TFILE only once. Attempts to open more than one instance of the TFILE return an error number 12 (file in use). For multithreaded requesters, the OPEN TFILE call must be performed before the first call to BEGINTRANSACTION. For single-threaded requesters, the first call to BEGINTRANSACTION implicitly opens the TFILE for you.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

GETTRANSACTIONDETAILS

This procedure returns transaction related information like `TransID`, `TXID`, `TxHandle`, type flags, begin tag, and process handle of the process that had begun the transaction.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(GETTRANSACTIONDETAILS)>
short  GetTransactionDetails (short,
                              long long _far*,
                              long long _far*,
                              short _far*,
                              long long _far*,
                              short _far*,
                              long _far*);
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(GETTRANSACTIONDETAILS)>
short  GetTransactionDetails (short,
                              long long _ptr64*,
                              long long _ptr64*,
                              short _ptr64*,
                              long long _ptr64*,
                              short _ptr64*,
                              int _ptr64*);
```

Syntax for TAL Programmers

```
error := GetTransactionDetails (InputID,
                                TransID,
                                TXID,
                                TXHandle,
                                TypeFlags,
                                Phandle,
                                BeginTag);
                                CALLABLE, EXTENSIBLE;
```

Parameters

error returned value

INT

is an error number indicating the outcome of an operation. The following are the possible errors:

FEOK	Successful completion.
FEMISSPARAM	Missing parameter(s).
FENOTRANSID	Current transaction in NIL state.
FEINVTRANSID	Invalid or obsolete transaction identifier.
FETMFNOTRUNNING	TMF not running.
FETMFNOTCONFIGURED	TMF not configured.
FEBADPARMVALUE	InputID is invalid.
FEINVALIDTXHANDLE	Invalid transaction handle.

For a list of file system error numbers, see *Operator Messages Manual*.

InputID input

INT

indicates whether TransID, TXID, TxHandle, or none of these are passed as input. Valid values are 0, 1, 2, and 3.

If InputID = 0

It indicates that no input is passed. In this case, the API retrieves the following details of the current transaction of the process:

- TransID
- TXID

- `TxHandle`
- `TypeFlags`
- `Phandle`
- `BeginTag`

At least one of `TransID`, `TXID`, `TxHandle`, `TypeFlags`, or `Phandle` must be passed as output parameter.

If `InputID = 1`

It indicates that `TransID` is passed as input. In this case, the API retrieves the following details based on the `TransID`:

- `TXID`
- `TxHandle`
- `TypeFlags`
- `Phandle`
- `BeginTag`

At least one of `TXID`, `TxHandle`, `TypeFlags`, or `Phandle` must be passed as output parameter.

If `InputID = 2`

It indicates that `TXID` is passed as input. In this case, the API retrieves the following details based on the `TXID`:

- `TransID`
- `TxHandle`
- `TypeFlags`
- `Phandle`
- `BeginTag`

At least one of `TransID`, `TxHandle`, `TypeFlags`, or `Phandle` must be passed as output parameter.

If `InputID = 3`

It indicates that `TxHandle` is passed as input. In this case, the API retrieves the following details based on the `TxHandle`:

- `TXID`
- `TransID`
- `TypeFlags`

- `Phandle`
- `BeginTag`

At least one of `TransID`, `TXID`, `TypeFlags`, or `Phandle` must be passed as output parameter.

`TransID` input/output

FIXED .EXT:REF:1
or
FIXED .EXT64:REF:1

`TransID` can be passed as an input parameter by setting the `InputID` to 1. `TransID` must be local when it is passed as an input parameter. This is an optional parameter.

If `TransID` is passed as an output parameter, then one of the following is returned:

- The `TransID` of the corresponding `TXID` or `TxHandle` that was passed when the `InputID` is set to 2 or 3.
- The current transaction when the `InputID` is set to 0.

`TXID` input/output

FIXED .EXT:REF:1
or
FIXED .EXT64:REF:1

`TXID` can be passed as an input parameter by setting the `InputID` to 2. This is an optional parameter.

If `TXID` is passed as an output parameter, then one of the following is returned:

- The `TXID` of the corresponding `TransID` or `TxHandle` that was passed when the `InputID` is set to 1 or 3.
- The current transaction when the `InputID` is set to 0.

`TxHandle` input/output

INT .EXT:REF:10
or
INT .EXT64:REF:10

`TxHandle` can be passed as an input parameter by setting the `InputID` to 3. This is an optional parameter.

If `TxHandle` is passed as an output parameter, the `TxHandle` of the current transaction when the `InputID` is set to 0 is returned or when the `InputID` is set to 1 or 2, `TxHandle` is not returned.

`TypeFlags` output

FIXED .EXT:REF:1
or
FIXED .EXT64:REF:1

This is an optional parameter.

If this attribute is passed, one of the following is returned:

- The type flags of the transaction of the corresponding `TransID`, `TXID`, or `TxHandle` that is passed when `InputID` is set to 1, 2, or 3.
- The type flags of the current transaction when the `InputID` is set to 0.

`Phandle` output

INT .EXT:REF:10
or
INT .EXT64:REF:10

This is an optional parameter.

If this attribute is passed, one of the following is returned:

- The process handle of the transaction beginner process of the corresponding `TransID`, `TXID`, or `TxHandle` that is passed when `InputID` is set to 1, 2, or 3.
- The process handle of the current transaction when the `InputID` is set to 0.

`BeginTag` output

INT(32) .EXT:REF:1
or
INT(32) .EXT64:REF:1

This is an optional parameter.

If this attribute is passed, one of the following is returned:

- The beginner tag associated with the corresponding `TransID`, `TXID`, or `TxHandle` that is passed when the `InputID` is set to 1, 2, or 3.
- The beginner tag associated with the current transaction when the `InputID` is set to 0.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

GETTRANSID

This procedure returns the transaction identifier of the calling process' current transaction..

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(GETTRANSID)>
short GETTRANSID ( short _near *transid );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(GETTRANSID)>
short GETTRANSID ( short _ptr64 *transid );
```

Syntax for TAL Programmers

```
status := GETTRANSID ( transid );                              ! o
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
75	Current transaction in nil state.
78	Invalid or obsolete transaction identifier.
82	TMF not running.
84	TMF not configured.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

transid output

INT:ref:4

or

INT .EXT64:ref:4

is a four-word array into which GETTRANSID returns the transaction identifier of the current transaction. Its form is as follows:

Transaction ID	Description
transid[0].<0:7>	contains 1 plus the Expand system number of the system in which BEGINTRANSACTION was called (this field contains 0 if the originating system is not part of a network). The system number identifies the home node of the transaction.
transid[0].<8:15>	contains the number of the processor in which the transaction originated.
transid[1:2]	contains a double-word sequence number to make the transaction identifier unique.
transid[3]	contains *tm-flags*, a number (zero or nonzero) representing flags used internally by TMF.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

GETTRANSINFO

This procedure returns the exttransid and transaction type flags. Use GETTRANSINFO when the calling process wants to pass exttransid to another process in the same Expand network so that the process can call TMF_JOIN_EXT_.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(GETTRANSINFO)>

short GETTRANSINFO ( long long _far *transid,
                       long long _far *type_flags );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(GETTRANSINFO)>

short GETTRANSINFO ( long long _ptr64 *transid,
                    long long _ptr64 *type_flags );
```

Syntax for TAL Programmers

```
status := GETTRANSINFO ( transid, ! o
                         type_flags ) ! o
                                CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

FEOK	Successful completion.
FEMISSPARM	Required parameter exttransid is missing.
FEBOUNDSERR	One or more of the supplied parameters is out of bounds.
FENOTRANSID	No current transaction.
FEINVTRANSID	Invalid transaction identifier.

For a list of file system error numbers, see the *Operator Messages Manual*.

`transid` output

FIXED .EXT:ref:1

or

FIXED .EXT64:ref:1

is the transaction identifier in internal format as obtained from GETTRANSID.

type_flags output

FIXED .EXT:ref:1

or

FIXED .EXT64:ref:1

the transaction type flags for the current transaction.

This parameter is optional.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

INTERPRETTRANSID

This procedure converts the supplied transaction identifier from internal form to its individual external numeric parts. If the conversion fails (*status* <> 0), all output parameters are undefined.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(INTERPRETTRANSID)>

short INTERPRETTRANSID ( long long transid
                       , long _far *system
                       , short _far *cpu
                       , long _far *sequence
                       ,[ short _far *tm-flags ] );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(INTERPRETTRANSID)>

short INTERPRETTRANSID ( long long transid
                       , int _ptr64 *system
                       , short _ptr64 *cpu
                       , int _ptr64 *sequence
                       ,[ short _ptr64 *tm-flags ] );
```

Syntax for TAL Programmers

```
status := INTERPRETTRANSID ( transid               ! i
                           , system                ! o
                           , cpu                   ! o
                           , sequence              ! o
                         [ , tm-flags ] );         ! o
```

Parameters

status returned value

INT

is a TMF or file system error number:

-2	Invalid internal transaction identifier (TMF).
-1	Missing `tm-flags` parameter (TMF).
0	Successful completion.
22	Invalid reference parameter.
29	Missing parameter(s).
82	TMF not running.
84	TMF not configured.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

transid input

FIXED:value

is the transaction identifier in internal format as obtained from GETTRANSID.

system output

INT(32) .EXT:ref:1

or

INT(32) .EXT64:ref:1

is the system number.

cpu output

INT .EXT:ref:1

or

INT .EXT64:ref:1

is the CPU number.

sequence output

INT(32) .EXT:ref:1

or

INT .EXT64:ref:1

is a sequence number greater than 0.

tm-flags output

INT .EXT:ref:1

or

INT .EXT64:ref:1

is a number representing flags used internally by TMF.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

RESUMETRANSACTION

This procedure sets the current transaction of the calling process either to the nil state or to the transaction identifier of the specified transaction. You identify the desired transaction by supplying the *trans-begin-tag* returned by a previous call to BEGINTRANSACTION.

RESUMETRANSACTION makes it possible for you to write multithreaded requesters that can process two or more transactions at the same time: the requester can only work on one transaction at a time. By using RESUMETRANSACTION to manipulate the current transaction, however, the requester can switch from one active transaction to another.

To use RESUMETRANSACTION, your program must include *trans-begin-tag* in all BEGINTRANSACTION calls and must remember which tag applies to which active transaction.

If RESUMETRANSACTION returns an error, this is an indication that future attempts to use the transaction for updates to the database will result in an error. The requester (or its backup) that initiated the transaction must still explicitly terminate the transaction by calling either ENDTRASACTION or ABORTTRANSACTION. The final outcome of the transaction will be indicated by the result of one of the calls.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(RESUMETRANSACTION)>
short RESUMETRANSACTION ( long trans-begin-tag );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(RESUMETRANSACTION)>
short RESUMETRANSACTION ( int trans-begin-tag );
```

Syntax for TAL Programmers

```
status := RESUMETRANSACTION ( trans-begin-tag );          ! i
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
76	Transaction ending or aborting.
78	Invalid or obsolete transaction identifier.
82	TMF not running.
84	TMF not configured.
90	Transaction's parent process failed.
92	Path to a participating network node failed.
93	Transaction aborted because it was pinning a file on the MAT, and more than 45% of the MAT filled during the transaction's lifetime.
94	Transaction aborted by operator command.
96	Transaction aborted by Autoabort threshold.
97	Transaction aborted by ABORTTRANSACTION call.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

Non-zero errors indicate that the transaction cannot be used to update the database.

Note that for all errors except 31, 36, and 78, the caller must still call ABORTTRANSACTION or ENDTRANSACTION to clean up data structures and determine the final outcome of the transaction.

trans-begin-tag input

INT(32)

is the value returned by the optional *trans-begin-tag* parameter of BEGINTRANSACTION. If the value of this parameter is 0D, the current transaction identifier of the calling process is reset to the nil state.

If the transaction identified by *trans-begin-tag* was initiated by either the calling process or its backup, the calling process' current transaction is set to the associated transaction identifier, even if RESUMETRANSACTION returns an error number.

When issued by the primary requester process of a process pair, RESUMETRANSACTION does not change the current transaction identifier of the backup process.

STATUSTRANSACTION

In addition to determining the status of a transaction, this procedure optionally accepts a transaction identifier as an input parameter. If you omit the transaction identifier from the call, STATUSTRANSACTION returns the status of the current transaction.

Syntax for C Programmers

Note. `This procedure can be called from 32-bit and 64-bit applications.`

- Syntax for TNS Programmers

```
#include <cextdecs(STATUSTRANSACTION)>

short STATUSTRANSACTION ( short _far *trans-status
                        ,[ long long transid ] );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(STATUSTRANSACTION)>

short STATUSTRANSACTION ( short _ptr64 *trans-status
                        ,[ long long transid ] );
```

Syntax for TAL Programmers

```
status := STATUSTRANSACTION ( trans-status                ! o
                              [ , transid ] );            ! i
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
53	File system internal error.
75	No such transaction.

78	Calling process’ current transaction in the nil state.
82	TMF not running.
84	TMF not configured.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

trans-status output

INT .EXT:*

or

INT .EXT64:ref:1

is the status of the transaction, as follows:

1 ACTIVE

2 PREPARED

3 COMMITTED

4 ABORTING

5 ABORTED

6 HUNG

transid input

FIXED:value

is an optional transaction identifier (in internal form). If this parameter is omitted, STATUSTRANSACTION returns the status of the current transaction.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TEXTTOTRANSID

This procedure converts a transaction identifier from external ASCII form to internal form. If the conversion fails (status <> 0), all output parameters are undefined.

If the string pointed to by the text parameter does not include a system name or number and the local system is a named system, TEXTTOTRANSID uses the local Expand node number as the system number.

If the string pointed to by the text parameter does not include a TMF flags value, TEXTTOTRANSID generates a TMF flags field containing 0.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TEXTTOTRANSID)>

short TEXTTOTRANSID ( char _far *text
                      , short text-byte-length
                      , long long _far *transid );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TEXTTOTRANSID)>

short TEXTTOTRANSID ( char _ptr64 *text
                      , short text-byte-length
                      , long long _ptr64 *transid );
```

Syntax for TAL Programmers

```
status := TEXTTOTRANSID ( text                              ! i
                        , text-byte-length                  ! i
                        , transid );                        ! o
```

Parameters

status returned value

INT

is a TMF or file system error number:

-6	Additional characters in transaction identifier (TMF).
-5	*text-byte-length* out of range (TMF).
-4	Invalid system name or number (TMF).
-3	Invalid TMF flags value (TMF).
-2	Invalid CPU (TMF).
-1	Invalid sequence (TMF).
0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
82	TMF not running.
84	TMF not configured.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

text input

STRING .EXT:ref:*

or

STRING .EXT64:ref:*

is the name of a string variable containing the external ASCII transaction identifier in one of the following formats:

```
\system-name(tm-flags).cpu.sequence
\system-number(tm-flags).cpu.sequence
\system-name.cpu.sequence
\system-number.cpu.sequence
cpu.sequence
```

text-byte-length input

INT:value

is the length of text, from 0 to 80 bytes. *text-byte-length* must specify the exact length of the external transaction identifier contained in the string pointed to by text.

transid output

FIXED:ref:1

or

FIXED .EXT64:ref:1

is the transaction identifier in internal format.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_BEGINTAG_FROM_TXHANDLE_

This procedure returns the begin-transaction-tag associated with the specified transaction handle. The returned tag matches the one returned by the BEGINTRANSACTION procedure, if the calling process began the transaction. It also matches the tag returned by AWAITIO[X] to indicate completion of a nowaited ENDTRANSACTION call.

If the specified transaction handle does not refer to a transaction begun or resumed by the calling process, the file system error number 715 (invalid transaction handle) is returned.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TMF_BEGINTAG_FROM_TXHANDLE_)>

short TMF_BEGINTAG_FROM_TXHANDLE_
                            ( short _far *tx-handle ,
                              long _far *trans-begin-tag);
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_BEGINTAG_FROM_TXHANDLE_)>

short TMF_BEGINTAG_FROM_TXHANDLE_
                            ( short _ptr64 *tx-handle ,
                             int _ptr64 *trans-begin-tag);
```

Syntax for TAL Programmers

```
status := TMF_BEGINTAG_FROM_TXHANDLE_ ( tx-handle ,       ! i
                                        trans-begin-tag ) ! o
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
82	TMF not currently running.
84	TMF not configured.
715	Invalid transaction handle.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

tx-handle input

INT .EXT:ref:10

or

INT .EXT64:ref:10

specifies the transaction handle associated with the transaction whose begin-transaction-tag is to be returned.

trans-begin-tag output

INT(32) .EXT:ref:1

or

INT(32) .EXT64:ref:1

returns the begin-transaction-tag associated with the supplied transaction handle.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_GETTXHANDLE_

This procedure returns the transaction handle of the current transaction.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TMF_GETTXHANDLE_)>
short TMF_GETTXHANDLE_ ( short _far *tx-handle );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_GETTXHANDLE_)>
short TMF_GETTXHANDLE_ ( short _ptr64 *tx-handle );
```

Syntax for TAL Programmers

```
status := TMF_GETTXHANDLE_ ( tx-handle )                    ! o
          CALLABLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
75	No current transaction.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

tx-handle output

INT .EXT:ref:10

or

INT .EXT64:ref:10

specifies the transaction handle of the current transaction of the calling process. The size of the returned handle is 20 bytes.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_GET_EXTTRANSID_

This procedure returns the `exttransid` associated with the current transaction of the process.

Note. `This procedure cannot be called by TNS applications.`

Syntax for C Programmers

Note. `This procedure can be called from 32-bit and 64-bit applications.`

Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_GET_EXTTRANSID_)>

short TMF_GET_EXTTRANSID_ ( long long _ptr64 *exttransid[2],
                            long long _ptr64 *type_flags );
```

Syntax for TAL Programmers

```
status := TMF_GET_EXTTRANSID_ ( exttransid,                  ! i
                                type_flags  )
! o
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
75	No current transaction
78	Transaction no longer valid.

For a list of file system error numbers, refer to the *Operator Messages Manual*

exttransid output

FIXED .EXT:ref:2

or

FIXED .EXT64:ref:2

the transactional identifier that can be used for a subsequent TMF_JOIN_EXT_ call, either in the calling process or in another process in the same EXPAND network.

type_flags output

FIXED .EXT:ref:1

or

FIXED .EXT64:ref:1

the transaction type flags for the current transaction.

This parameter is optional.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_GET_TX_ID_

This procedure extracts a transactional identifier (txid) from a transaction handle. The identifier is only valid within the lifetime of the calling process or until TMF is either shut down or crashes. The returned identifier can be used in calls to TMF_RESUME_ to specify the transaction to be resumed by the process. The same transactional identifier is returned by TMF_SUSPEND_.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TMF_GET_TX_ID_)>

short TMF_GET_TX_ID_ ( short _far *tx-handle ,
                       long long _far *txid );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_GET_TX_ID_)>

short TMF_GET_TX_ID_ ( short _ptr64 *tx-handle ,
                       long long _ptr64 *txid );
```

Syntax for TAL Programmers

```
status := TMF_GET_TX_ID_ ( tx-handle ,                    ! i
                           txid  )                        ! o
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
78	Transaction no longer valid.

82	TMF not running.
84	TMF not configured.
715	Invalid transaction handle.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

tx-handle input

INT .EXT:ref:10

or

INT .EXT64:ref:10

specifies the transaction handle associated with the transaction whose transactional identifier is to be returned.

txid output

FIXED .EXT:ref:1

or

FIXED .EXT64:ref:1

a transactional identifier to be used in a subsequent call to TMF_RESUME_ or TMF_JOIN_.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_JOIN_

This procedure allows a process to participate in a transaction initiated by another process without taking over as the ENDTRANSACTION caller. TMF_JOIN_ can be called by more than one process in the same Expand node at the same time. The BEGINTRANSACTION caller need not call TMF_SUSPEND_ before another process calls TMF_JOIN_ and can participate in the transaction at the same time.

Joining a transaction means:

- Making the transaction the current transaction.
- The process calling TMF_JOIN_ is not allowed to call ENDTRANSACTION.

To relinquish control of a joined transaction, the calling process must issue a TMF_SUSPEND_ call; otherwise, the process that initiated the transaction receives error 730 when it tries to run ENDTRANSACTION.

TMF_JOIN_ has the same semantics as the implicit transaction propagation provided by the file system and can provide the same function when file system propagation is not appropriate to the application.

The transaction identifier passed to this procedure is the one returned by TMF_GET_TX_ID_.

This procedure cannot be used for transactions that were begun on a different EXPAND node.

A joined transaction uses one of the entries in the calling process' TFILE. If the process has not explicitly opened the TFILE, the joined transaction uses the one TFILE entry that exists for all processes. If there are no available TFILE entries when the call to TMF_JOIN_ is issued, the call is rejected with file system error 83 (too many transactions).

If the calling process is the primary process of a process pair and the TFILE is being checkpointed, the calling process should checkpoint the TFILE after calling TMF_JOIN_ to modify the backup process' TFILE accordingly.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

```
#include <cextdecs(TMF_JOIN_)>
short TMF_JOIN_ ( long long txid );
```

Syntax for TAL Programmers

```
status := TMF_JOIN_ ( txid  )                           ! i
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
76	Transaction ending or aborting.
78	Invalid transaction identifier or transaction not started on this EXPAND node.
82	TMF not running.
83	Too many transactions (a single-threaded requester tried to initiate a transaction while it still had one in progress, or a multithreaded requester attempted to initiate more concurrent transactions than there were TFILE entries).
84	TMF not configured.
721	BEGINTRANSACTION not completed.
731	The process has already called TMF_JOIN_ for this transaction.
732	The process has already called TMF_RESUME_ for this transaction.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

txid input

FIXED

the transactional identifier returned by TMF_GET_TX_ID_.

TMF_JOIN_EXT_

This procedure allows a process to participate in a transaction, initiated by another process, without taking over as the ENDTRANSACTION caller. TMF_JOIN_EXT_ can be called by more than one process in any node in the beginner’s Expand network at the same time. The BEGINTRANSACTION caller need not call TMF_SUSPEND_EXT_ before another process calls TMF_JOIN_EXT_ and can participate in the transaction at the same time. TMF_JOIN_EXT_ is interchangeable with TMF_JOIN_ in the beginner's node.

Joining a transaction means:

- Making the transaction the current transaction.
- The process calling TMF_JOIN_EXT_ is not allowed to call ENDTRANSACTION.

TMF_JOIN_EXT_ must be completed in the calling process by a call to TMF_SUSPEND_EXT_.

Note. This procedure cannot be called by TNS applications.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_JOIN_EXT_)>
short TMF_JOIN_EXT_ ( long long _ptr64 *exttransid );
```

Syntax for pTAL Programmers

```
status := TMF_JOIN_EXT_ ( exttransid  )                  ! i
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).

76	Transaction ending or aborting.
78	Invalid transaction identifier.
82	TMF not running.
83	Too many transactions (a single-threaded requester tried to initiate a transaction while a transaction is in progress, or a multithreaded requester attempted to initiate more concurrent transactions than there were TFILE entries).
84	TMF not configured.
721	BEGINTRANSACTION not completed.
731	The process has already called TMF_JOIN_ or TMF_JOIN_EXT_ for this transaction. This error now always returns when TMF_JOIN_EXT_ is called and the calling process has a current transaction.
732	The process has already called TMF_RESUME_ for this transaction.

For a list of file system error numbers, see the *Operator Messages Manual*.

exttransid input

FIXED:ref:2

or

FIXED .EXT64:ref:2

the transactional identifier returned by TMF_SUSPEND_EXT_ or GETTRANSINFO.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

Usage Considerations

Applications using TMF_JOIN_EXT_ must also use TMF_SUSPEND_EXT_. TMF_SUSPEND_EXT_ and GETTRANSINFO return `exttransid`, which is used in subsequent calls to TMF_JOIN_EXT_.

TMF_JOIN_EXT_ has the same semantics as the implicit transaction propagation provided by the file system and provides the same function when the file system propagation is not appropriate to the application.

A joined transaction uses one of the entries in the calling process' TFILE. If the process has not explicitly opened the TFILE, the joined transaction uses one TFILE entry that exists for all processes. If there are no available TFILE entries when the call

to TMF_JOIN_EXT_ is issued, the call is rejected with file system error FETOOMANYTRANSBEGINS (too many transactions).

If the calling process is the primary process of a process pair and the TFILE is being checkpointed, the calling process should checkpoint the TFILE after calling TMF_JOIN_EXT_ to modify the backup process' TFILE.

If the calling process has not relinquished control of a joined transaction by issuing a TMF_SUSPEND_EXT call, one of the following takes place when the process that initiated the transaction calls ENDTRANSACTION:

- If the calling process is on the beginner's node, then a retry error FEJOINOUTSTANDING will be returned.

  OR

- If the calling process is on a remote node, the transaction will be aborted, and FETRANSABORTED will be returned.

Note. If the Joiners CPU fails while the join is active, the transaction is aborted with abort flags set to AbortDueToServerFailure.

TMF_RESUME_

This procedure allows a process to resume a previously suspended transaction.

Resuming a transaction means:

1. Making the transaction the current transaction
2. Taking over as the ENDTRANSACTION caller for the transaction

After successfully resuming a transaction, the calling process can then issue an ENDTRANSACTION call. The process that suspended the transaction and the one that resumes it need not reside in the same CPU, but they must reside in the same EXPAND node. Only one process will succeed in resuming a previously suspended transaction.

The transaction identifier passed to this procedure is the one returned by TMF_SUSPEND_; it can also, however, be obtained by calling TMF_GET_TX_ID_.

This procedure cannot be used for transactions that were begun on a different EXPAND node.

Resuming a transaction uses one of the entries in the calling process' TFILE. If the process has not explicitly opened the TFILE, the resumed transaction uses the one TFILE entry that exists for all processes. If there are no available TFILE entries when the call to TMF_RESUME_ is issued, the call is rejected with file system error 83 (too many transactions).

If the calling process is the primary process of a NonStop process pair and the TFILE is being checkpointed, the calling process should checkpoint the TFILE after calling TMF_RESUME_ to modify the backup process' TFILE accordingly.

Syntax for C Programmers

Note. `This procedure can be called from 32-bit and 64-bit applications.`

```
#include <cextdecs(TMF_RESUME_)>

short TMF_RESUME_ ( long long txid );
```

Syntax for TAL Programmers

```
status := TMF_RESUME_ ( txid )                              ! i
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
76	Transaction ending or aborting.
78	Invalid transaction identifier or transaction not started on this EXPAND node.
82	TMF not running.
83	Too many transactions (a single-threaded requester tried to initiate a transaction while it still had one in progress, or a multithreaded requester attempted to initiate more concurrent transactions than there were TFILE entries).
84	TMF not configured.
717	Transaction not suspended.
721	BEGINTRANSACTION not completed.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

txid input

FIXED

the transactional identifier returned by either a TMF_GET_TX_ID_ or TMF_SUSPEND_ call.

TMF_SETTXHANDLE_

This procedure sets the current transaction to the one associated with the specified transaction handle.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TMF_SETTXHANDLE_)>
short TMF_SETTXHANDLE_ ( short _far *tx-handle );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_SETTXHANDLE_)>
short TMF_SETTXHANDLE_ ( short _ptr64 *tx-handle );
```

Syntax for TAL Programmers

```
status := TMF_SETTXHANDLE_ ( tx-handle )                    !i
          CALLABLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
75	No current transaction.
78	Invalid transaction identifier or transaction not started on this EXPAND node.
82	TMF not currently running.
715	Invalid transaction handle.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

tx-handle input

INT .EXT:ref:10

or

INT .EXT64:ref:10

specifies the transaction handle of the transaction that is to become the current transaction.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_SUSPEND_

This procedure allows the calling process to relinquish its duties as ENDTRANSACTION caller for the current transaction, provided the transaction was begun by or resumed by the calling process. A successful call to TMF_SUSPEND_ is referred to as suspending the transaction.

TMF_SUSPEND_ is also used by the calling process to relinquish control of a transaction to which it has been joined; otherwise, the process that initiated the transaction receives an error 730 when it tries to run ENDTRANSACTION.

The suspended transaction is no longer the current transaction of the calling process and the calling process is no longer allowed to call ENDTRANSACTION for the transaction (until it subsequently calls TMF_RESUME_). The TFILE entry previously occupied by the suspended transaction is vacated.

A different process can resume the transaction by calling TMF_RESUME_. The resuming process can reside in a different CPU than the process that began the transaction, but it must reside in the same EXPAND node. The suspending process must pass the transaction identifier to the process that subsequently resumes the transaction. The transaction identifier is returned by TMF_SUSPEND_; alternatively, it can be obtained by calling TMF_GET_TX_ID_ prior to calling TMF_SUSPEND_.

This procedure cannot be used for transactions that were not begun by or are not currently resumed by the calling process.

If the calling process is the primary process of a NonStop process pair and the TFILE is being checkpointed, the calling process should checkpoint the TFILE after calling TMF_SUSPEND_ to modify the backup process' TFILE accordingly.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TMF_SUSPEND_)>
short TMF_SUSPEND_ ( long long _far *txid );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_SUSPEND_)>
short TMF_SUSPEND_ ( long long _ptr64 *txid );
```

Syntax for TAL Programmers

```
status := TMF_SUSPEND_ ( txid )                  ! o
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
75	No current transaction.
76	Transaction ending or aborting.
78	Invalid transaction identifier or transaction not started on this EXPAND node.
81	Outstanding nowaited I/O requests exist for the specified transaction.
82	TMF not running.
84	TMF not configured.
716	Calling process is not the beginner, importer, or resumer of the specified transaction.

For a list oflist of file system error numbers, refer to the *Operator Messages Manual*.

txid output

FIXED .EXT:ref:1

or

FIXED .EXT64:ref:1

the transactional identifier that can be used for a subsequent TMF_RESUME_ call, either in the calling process or in another process in the same EXPAND node.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_SUSPEND_EXT_

This procedure allows the calling process to relinquish control of a transaction— the current transaction of the calling process. Applications that use TMF_JOIN_EXT_ must also use TMF_SUSPEND_EXT_.

If the calling process does not relinquish control of a joined transaction by issuing a TMF_SUSPEND_EXT call, one of the following takes place when the process that initiated the transaction calls ENDTRANSACTION:

- If the calling process is on the beginner's node, a retry error FEJOINOUTSTANDING will be returned.

  OR

- If the calling process is on a remote node, the transaction will be aborted, and FETRANSABORTED will be returned.

Semantically, TMF_SUSPEND_EXT_ is similar to TMF_SUSPEND_ and must be used in conjunction with TMF_JOIN_EXT_ in applications where transactions can span EXPAND nodes. TMF_SUSPEND_ or TMF_JOIN_ or TMF_RESUME_ work only within an EXPAND node.

The suspended transaction is no longer the current transaction of the calling process and the calling process is no longer allowed to call ENDTRANSACTION for the transaction. The TFILE entry previously occupied by the suspended transaction is vacated.

If the calling process is the primary process of a NonStop process pair and the TFILE is being checkpointed, the calling process should checkpoint the TFILE after calling TMF_SUSPEND_EXT_ to modify the backup process' TFILE accordingly.

Note. This procedure cannot be called by TNS applications.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_SUSPEND_EXT_)>

short TMF_SUSPEND_EXT_ ( long long _ptr64 *exttransid );
```

Syntax for pTAL Programmers

```
status := TMF_SUSPEND_EXT_ ( exttransid )                  ! o
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
75	No current transaction.
76	Transaction ending or aborting.
78	Invalid transaction identifier or transaction not started on this EXPAND node.
81	Outstanding nowaited I/O requests exist for the specified transaction.
82	TMF not running.
84	TMF not configured.
716	Calling process is not the beginner, importer, or resumer of the specified transaction.

For a list of file system error numbers, see the *Operator Messages Manual*.

exttransid output

FIXED .EXT:ref:2

or

FIXED .EXT64:ref:2

the transactional identifier that can be used for a subsequent TMF_JOIN_EXT_ call, either in the calling process or in another process in the same EXPAND network.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_TXBEGIN_

This procedure is the same as BEGINTRANSACTION except that it includes a new parameter to specify a timeout value.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TMF_TXBEGIN_)>

short TMF_TXBEGIN_ ( [ long timeout ] ,
                     [ long _near *trans-begin-tag ] );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_TXBEGIN_)>

short TMF_TXBEGIN_ ( [ int timeout ] ,
                     [ int _ptr64 *trans-begin-tag ] );
```

Syntax for TAL Programmers

```
status := TMF_TXBEGIN_ ( [ timeout ]                 ! i
                         [ , trans-begin-tag ] )     ! o
          CALLABLE, RESIDENT, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
30	System unable to obtain message block, or is already using its maximum number of RECEIVE or SEND message blocks.
82	TMF not running.
83	Too many transactions (a single-threaded requester tried to initiate a transaction while it still had one in progress, or a multithreaded requester attempted to initiate more concurrent transactions than there were TFILE entries).

84	TMF not configured.
86	Audit trail at *begin-trans-disable* threshold, or operator has disabled the BEGINTRANSACTION procedure.
98	TFILE was opened using a nonzero *sync-depth* and all transaction control blocks (TCBs) within the caller's CPU are currently occupied.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

timeout input

INT(32)

the number of seconds that the transaction should be allowed to exist before it is aborted due to a timeout. If the specified value is greater than the configured AutoAbort attribute, the AutoAbort value takes precedence.

This parameter is optional. If omitted, the configured AutoAbort value is used.

trans-begin-tag output

INT(32) .EXT:ref:1

or

INT(32) .EXT64:ref:1

returns a value that identifies the new transaction among other transactions that the calling process has initiated. This parameter is optional for single-threaded requesters and process pairs, but it is required for multithreaded requesters and process pairs.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_TXHANDLE_FROM_BEGINTAG_

This procedure returns the transaction handle associated with the specified begin-transaction-tag.

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TMF_SETTXHANDLE_FROM_BEGINTAG_)>

short TMF_TXHANDLE_FROM_BEGINTAG_
                             ( long trans-begin-tag ,
                                short _far *tx-handle );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_SETTXHANDLE_FROM_BEGINTAG_)>

short TMF_TXHANDLE_FROM_BEGINTAG_
                             ( int trans-begin-tag ,
                                short _ptr64 *tx-handle );
```

Syntax for TAL Programmers

```
status := TMF_TXHANDLE_FROM_BEGINTAG_ ( trans-begin-tag,  ! i
                                        tx-handle )       ! o
          CALLABLE, EXTENSIBLE;
```

Parameters

status returned value

INT

is a file system error number:

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
78	Invalid transaction identifier or transaction not started on the EXPAND node.
84	TMF not configured.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

trans-begin-tag input

INT(32)

specifies the begin-transaction-tag associated with the transaction whose transaction handle is to be returned.

tx-handle output

INT .EXT:ref:10

or

INT .EXT64:ref:10

returns the transaction handle associated with the supplied begin-transaction-tag. The size of the returned handle is 20 bytes.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TRANSIDTOTEXT

This procedure converts a transaction identifier from internal form to external ASCII form. If the conversion fails (*status* <> 0), all output parameters are undefined.

Note. Attempts to pass a transactional identifier (*txid*) to the TransIdToText procedure instead of a transaction identifier (*transid*) will fail with a return code of -1. For information about *txId*s, see the TMF_GET_TX_ID_ procedure description.

text-byte-length specifies how many bytes in the string variable pointed to by *text* are available for use by TRANSIDTOTEXT. *bytes-used* parameter specifies how many of those bytes TRANSIDTOTEXT actually used.

If the system number of the transaction identifier can be converted to a system name, the transaction identifier is formatted in either of the following ways (depending upon whether or not the transaction identifier contains a nonzero TMF flags value):

```
\system-name(tm-flags).cpu.sequence
\system-name.cpu.sequence
```

If the system number of the transaction identifier cannot be converted to a system name, the transaction identifier is formatted in either of the following ways (depending upon whether or not the transaction identifier contains a nonzero TMF flags value):

```
\system-number(tm-flags).cpu.sequence
\system-number.cpu.sequence
```

If the system is not named, the transaction identifier is formatted as follows:

```
cpu.sequence
```

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TRANSIDTOTEXT)>

short TRANSIDTOTEXT ( long long transid
                    , char _far *text
                    , short text-byte-length
                    , short _far *bytes-used );
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TRANSIDTOTEXT)>

short TRANSIDTOTEXT ( long long transid
                    , char _ptr64 *text
                    , short text-byte-length
                    , short _ptr64 *bytes-used );
```

Syntax for TAL Programmers

```
status := TRANSIDTOTEXT ( transid                          ! i
                        , text                             ! o
                        , text-byte-length                 ! i
                        , bytes-used );                    ! o
```

Parameters

status returned value

INT

is a TMF or file system error number:

-2	String too short (TMF).
-1	Invalid internal transaction identifier (TMF).
0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
82	TMF not running.
84	TMF not configured.

For a list of file system error numbers, refer to the *Operator Messages Manual*.

transid input

FIXED:value

is the name of the variable containing the transaction identifier in internal form.

text output

STRING .EXT:ref:*

or

STRING .EXT64:ref:*

is the name of the string variable into which TRANSIDTOTEXT is to store the external ASCII form of the transaction identifier.

text-byte-length input

INT:value

specifies the maximum number of bytes available in the string variable pointed to by the text parameter. Any value greater than 3 is acceptable.

bytes-used output

INT .EXT:ref:1

or

INT .EXT64:ref:1

specifies the number of bytes in the string variable pointed to by text that were actually used by TRANSIDTOTEXT.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

TMF_VERSION_EXT_

Syntax for C Programmers

Note. This procedure can be called from 32-bit and 64-bit applications.

- Syntax for TNS Programmers

```
#include <cextdecs(TMF_VERSION_EXT_)>
short  TMF_VERSION_EXT_ ( short _far *
                        , short _far *
                        , char _far *
                        , short
                        , char _far *
                        , short
                        , char _far *
                        , short
                        , short _far *
                        , short _far *
                        , short _far *
                        , short _far *);
```

- Syntax for TNS/E Programmers

```
#include <cextdecs(TMF_VERSION_EXT_)>

short  TMF_VERSION_EXT_ ( short _ptr64*
                        , short _ptr64 *
                        , char _ptr64 *
                        , short
                        , char _ptr64 *
                        , short
                        , char _ptr64*
                        , short
                        , short _ptr64*
                        , short _ptr64 *
                        , short _ptr64 *
                        , short _ptr64 *);
```

Syntax for TAL Programmers

```
Error :=  TMF_VERSION_EXT_ ( Version
                           , SubVersion
                           , Platform_String:Platform_Max_Len
                           , Release_String:Release_Max_Len
                   , Release_Date_String:Release_Date_Max_Len
                           , Platform_Return_Len
                           , Release_Return_Len
                           , Release_Date_Return_Len
                           , Error_Detail );
```

Parameters

error returned value

INT

is a File System error number indicating the outcome of an operation:.

0	Successful completion.
22	Bounds error.
29	Missing parameter(s).
84	TMF is not configured.
590	Parameter value invalid or inconsistent.

For a list of file system error numbers, see *Operator Messages Manual*.

Version output

Int .EXT:REF:1

or

Int .EXT64:REF:1

is an optional output parameter. `Version` returns the current version of the TMF installed on a NonStop system. For example, if TMF 3.7 is installed on a NonStop system, `Version` returns 3.

SubVersion output

Int .EXT:REF:1

or

Int .EXT64:REF:1

is an optional output parameter. `SubVersion` returns the current subversion of the TMF installed on a NonStop system. For example, if TMF 3.7 is installed on a NonStop system, `SubVersion` returns 7.

Platform_String output

String .EXT:REF:*

or

String .EXT64:REF:*

is an optional output parameter. `Platform_String` returns the release version installed on a NonStop system. For example, if T8607H01^AKV is installed on a NonStop system, `Platform_String` returns H01.

Platform_Max_Len input

INT

is an optional input parameter. This option is required when `Platform_String` is passed. `Platform_Max_Len` must be set to the maximum size in bytes of the `Platform_String` output buffer.

Release_String output

String .EXT:REF:*

or

String .EXT64:REF:*

is an optional output parameter. `Release_String` returns the software product revision installed on a NonStop system. For example, if T8607H01^AKV is installed on a NonStop system, `Release_String` returns AKV.

Release_Max_Len input

INT

is an optional input parameter. This option is required when Release_String is passed. Release_Max_Len must be set to the maximum length in bytes of the Release_String output buffer.

Release_Date_String output

String .EXT:REF:*

or

String .EXT64:REF:*

is an optional output parameter. Release_Date_String returns the release date of the software product revision installed on a NonStop system. For example, if T8607H01^AKV is installed on a system, Release_Date_String returns 20Feb2013.

Release_Date_Max_Len input

INT

is an optional input parameter. This option is required when Release_Date_String is passed. Release_Date_Max_Len must be set to the maximum size in bytes of the Release_Date_String output buffer.

Platform_Return_Len output

Int .EXT:REF:1

or

Int .EXT64:REF:1

is an optional output parameter. This option is required when Platform_String is passed. Platform_Return_Len returns the length in bytes of the Platform_String output parameter.

Release_Return_Len output

Int .EXT:REF:1

or

Int .EXT64:REF:1

is an optional output parameter. This option is required when Release_String is passed. Release_Return_Len returns the length in bytes of the Release_String output parameter.

Release_Date_Return_Len output

Int .EXT:REF:1

or

Int .EXT64:REF:1

is an optional output parameter. This option is required when `Release_Date_String` is passed. `Release_Date_Return_Len` returns the length in bytes of the `Release_Date_String` output parameter.

Error_Detail output

Int .EXT:REF:1

or

Int .EXT64:REF:1

is an optional output parameter. For a non-zero value of error, `Error_Detail` returns the position of the parameter, which caused the first error. For example, if `Release_Date_Max_Len` has caused an error, `Error_Detail` returns the value 8.

Note. EPTAL callers can pass 64-bit pointers by specifying the compiler directives __EXT64 and SETTOG _64BIT_CALLS before sourcing from the EXTDECS file.

5

TMF ARLIB2 Audit-Reading Procedures

This section describes the ARLIB2 audit-reading interface that is available in the H06.03 and later release version updates (RVUs) of the TMF product.

The TMF ARLIB2 audit-reading interface allows you to programmatically examine TMF audit records for SQL/MX, SQL/MP, and Enscribe objects. The interface consists of two parts: a set of procedures that you bind into your program and a set of data definitions that describe the information returned by the procedures. Table 5-2 summarizes the ARLIB2 audit-reading procedures.

Information about how to include the TMF ARLIB2 audit-reading procedures in your application is described under How to Include Audit Reading in an Application on page 5-110.

This section contains the following topics:

- ARLIB2 Compared to ARLIB on page 5-2
- Cursors on page 5-3
- Restoring Audit-Trail Files From Audit Dumps on page 5-4
- Retrieving Information From Audit Records on page 5-5
- Error Reporting on page 5-6
- Audit Compression on page 5-7
- Reading Active Audit Files on page 5-8
- Reading a Range of Audit-Trail Files on page 5-9
- Reading a Merged Audit Trail With a MERGE Cursor on page 5-10
- Reading a Merged Audit Trail Without a MERGE Cursor on page 5-11
- Distributed Transactions on page 5-12
- Auxiliary Audit Trails on page 5-15
- NonStop SQL/MP Internal Field Formats on page 5-16
- Audit Records on page 5-19
- Procedure Calls on page 5-32
- Error Codes on page 5-103
- How to Include Audit Reading in an Application on page 5-110

Once TMF audit-reading procedures are bound into your program, you can use them to open an audit-trail file and read individual records from that file. Rather than return entire audit records, the procedures return only those fields that contain information useful to you from only those records useful to you. Fields and records that are only of use to TMF are not accessible. By ignoring such information, the procedures make it less likely that future changes to the format of TMF audit trails will adversely affect your audit-reading programs.

TMF audit-reading procedures do no filtering or analysis; they simply provide selected information from each accessible audit record. If you want any filtering, analysis, or correlation of the returned data, your program must do it.

Audit records do not include a user ID or a terminal name. You can incorporate this type of information into an audit trail, however, by having your application write a record to an audited security log file after each BEGINTRANSACTION process; this would insert an accessible audit record into the audit trail. The format and content of such records could be customized to your specific application. To conserve disk space, you could then delete the security record immediately from the audited security log file.

ARLIB2 Compared to ARLIB

The ARLIB2 audit-reading procedures read audit records for SQL/MX objects as well as for Enscribe and SQL/MP objects. The ARLIB audit-reading procedures read audit records only for Enscribe and SQL/MP objects.

The changes necessary to support SQL/MX objects are significant, including several new procedures and changes to many of the record formats. Programs bound with previous RVUs of ARLIB must not be used on systems that use SQL/MX objects. Such programs must be rebound with a current version of ARLIB, or they must be changed to use ARLIB2.

ARLIB2 is a native object file. To use this product, existing programs must be changed to use the native compiler for their language and to link with NLD. The PASCAL compiler cannot generate native code, however, so ARLIB2 is not usable by programs written in PASCAL.

To read and process audit records for SQL/MX objects, you use combinations of the following ARLIB2 procedures:

ARGETMXCOLUMNINFO

Returns column-format information for the SQL/MX object indicated by the current audit record.

ARFETCHMXBEFOREDATA

Copies the before-image field from an SQL/MX audit record to the application buffer (uses bit maps).

ARFETCHMXBEFOREDATA2

Copies the before-image field from an SQL/MX audit record to the application buffer (uses byte maps).

ARFETCHMXAFTERDATA

Copies the after-image field from an SQL/MX audit record to the application buffer (uses bit maps).

ARFETCHMXAFTERDATA2

Copies the after-image field from an SQL/MX audit record to the application buffer (uses byte maps).

The structure of the ARRECORD for ARLIB2 differs from the structure for ARLIB, as follows:

- A field has been added to identify audit records for SQL/MX objects.
- The rba field used to return the location of the audit record in the audit file has been expanded from 32 to 64 bits.

△ **Caution.** Relative byte address (rba) values for audit-trail files might exceed the size allowed by C INT or pTAL INT(32) data types.

- The before-image and after-image length fields have been expanded from 16 to 32 bits.
- Most of the fields have been reordered to allow for better performance.

Note. Use ARLIB2 APIs to read and process SQL/MX fields as described in the section Reading Audit Records for SQL/MX Objects on page 5-12. Some of the data types are read as ASCII string as described in the section *Using Host Variables in a C/C++ Program* of *SQL/MX Programming Manual for C and COBOL*.

Cursors

A cursor provides a reference into an audit trail. Each cursor reflects the generic name of an audit trail, the sequence number of the file in the trail, the relative byte address (RBA) of a record in the file, and the direction in which the cursor is to move through the file (forward or reverse).

Cursor Declaration

You declare a cursor by using the AROPEN procedure. Each AROPEN call passes the generic name of the desired audit trail and returns a cursor number that you use for future references to that cursor. You can only open audit trails on the local node. Until released by a call to the ARCLOSE procedure, each cursor continues to refer to the specified audit trail. You can have up to 30 cursors declared simultaneously, any number of which can refer to the same audit trail.

Cursor Positioning

You declare the location and direction of a cursor by using the ARPOSITION procedure. Each ARPOSITION call supplies the cursor number, the sequence number of the audit-trail file, an initial byte offset into the file, and the direction in which the cursor is to move for successive reads. If the supplied offset is not at the beginning of a record recognized by the audit-reading interface, then the first recognized record in the current cursor direction is retrieved. Attempts to position the cursor beyond the end-of-file (EOF) of an audit-trail file cause an error return.

The cursor position changes dynamically. Each read for a particular cursor causes the position of that cursor to be advanced to the next recognized record. When a cursor crosses from one audit-trail file to another, the new file is automatically opened and the cursor's sequence number and RBA are updated accordingly.

Restoring Audit-Trail Files From Audit Dumps

When you declare a cursor, you can indicate whether or not audit-trail files that are not on disk should be restored from audit dumps. When you are reading from a cursor that is positioned at a missing audit-trail file, a request is sent to the TMP process to restore the file from the appropriate audit dump; this is true whether you do an explicit positioning to a missing file, or successive reads cause the cursor to cross over into a missing file.

Missing files are restored to an audit-restore volume, a disk on the local system used to store audit-trail files that are restored from an audit dump as part of the recovery procedure. This minimizes the impact of file restoration on normal transaction processing.

You can also elect to have successive audit-trail files restored ahead of time. When you do that and the next audit-trail file in the current cursor direction is missing, the missing file is restored automatically: this saves time when reading sequentially through several audit files.

You specify the desired audit-trail file restoration information by using the TMFCOM ADD AUDITTRAIL and ALTER AUDITTRAIL commands. Tape mount requests, if necessary, are handled by the labeled tape subsystem.

When unable to restore a requested file, the TMP returns an error indication.

To use this feature:

- You must be reading audit-trail files that were generated on the local system.
- The associated audit dumps must be included in the TMF catalog.
- TMF must be running.

Retrieving Information From Audit Records

Once you position a cursor, each call to ARREAD reads the audit record at the cursor position. ARREAD returns the fixed-length fields and common attributes in a record structure. The audit-trail file sequence number and RBA returned by ARREAD permit you to position the cursor directly at that record if you need to later.

For distributed transactions, commit and abort records are written to the master audit trail on the parent node when the parent node releases its locks. Such records can be retrieved by calls to ARREAD, and are identified by the record types NETWORK-COMMIT and NETWORK-ABORT (RECTYPE constants 18 and 19, respectively). To retrieve NETWORK-COMMIT and NETWORK-ABORT records, you call ARGETNETWORKRECS prior to the first call to ARREAD. Conversely, the ARSTOPNETWORKRECS procedure disables the returning of NETWORK-COMMIT and NETWORK-ABORT records. (Alternatively, you can use ARSETOPTIONS to change the network options.)

For Enscribe and SQL/MP, you read the variable-length fields from audit records separately by calling the ARFETCH *objectname* procedure immediately after using ARREAD. The seven object names are AFTERIMAGE, AUXPOINTER, BEFOREIMAGE, CHILDNODELIST, FIELDVALUE, FRAGMENT, and RECORDKEY.

For SQL/MX, you read the variable-length fields from audit records separately by calling the ARFETCHMXBEFOREDATA, ARFETCHMXBEFOREDATA2, ARFETCHMXAFTERDATA, or ARFETCHMXAFTERDATA2 procedures.

The before-image, after-image, and record key are copied into your application buffer byte-for-byte. For audit records that contain variable-length fields, the byte length of such fields is specified in the record returned by ARREAD. The auxiliary pointer is a structure specifying a range of audit in an audit trail. The child node list is an array of 32-bit system numbers that starts in the first word of the specified application buffer. For audit records that include a child node list, the number of elements (child nodes) in the list is specified in the record returned by ARREAD. Fragment is a pair of before- and after-image fragments from a compressed Enscribe update record. If you issue a fetch procedure call for a variable-length field that is not present in the record most recently returned by ARREAD, the procedure returns an error -17 (ARE-FIELD-NOT-PRESENT).

To fetch one or more variable-length fields from an audit record, you must first have read the record by using the ARREAD procedure. ARREAD saves information about each record it reads and the fetch procedures use that information to retrieve the associated variable-length fields. Calls to AROPEN, ARCLOSE, or ARPOSITION erase that information; after such calls, further fetch attempts will fail with an error -9 (ARE-NO-CURRENT-RECORD) until the next successful call to ARREAD.

How you go about retrieving the key or record address of a data record whose modification is recorded in the audit trail depends upon the file type and the type of audit record.

- For key-sequenced files, the key of the record is present in the before-image and after-image of the record. Because these images are not available for the UPDATE AUDITCOMP record, the key is stored separately in the audit record and must be fetched by using the ARFETCHRECORDKEY procedure. For an UPDATE FIELDCOMP record, you use the ARFETCHFIELDVALUE procedure to retrieve the before-image and after-image fields.
- For entry-sequenced, relative, and unstructured files, you use the ARGETRECADDR and ARGETRECADDR64 procedures to retrieve the record address.

Error Reporting

The TMF audit-reading interface has two error reporting mechanisms: return codes and messages printed to the operator terminal.

Return Codes

The first parameter of each procedure call, which is always required, is the name of a variable to receive the return code. If that parameter is not passed to a procedure, the procedure returns immediately without taking any action. The various types of return codes have the following general meanings:

- Zero indicates that the procedure completed successfully.
- Negative numbers indicate that an error occurred.
- Positive numbers indicate warnings: although the procedure was successful, some unusual condition occurred. Warnings take the form of a bit map. This allows new warnings to be easily added and multiple warnings to be returned simultaneously.

Applicable warning codes are described in the syntax definition of each individual procedure call, later in this section; the error codes are described in Table 5-3.

Messages Printed to the Operator Terminal

There are times when a simple error code is insufficient. Two examples are the detection of a corrupt block in an audit file and the retry messages associated with failed audit restore attempts.

In such cases, a message is printed on the operator terminal specified in the call to ARSTART. If the problem causes the procedure to exit, an error code indicating the general type of error is also returned to the calling program. Examples of this kind of error are -800 (cursor error) and -900 (error reading the audit).

Procedural Retrieval of Message Text

When messages corresponding to error codes are printed on the operator terminal, the error information is normally remembered by the associated cursor. The text of the most recent message generated for a given cursor is available by calling the ARPRINTMESSAGE or ARGETMESSAGELINE procedures.

ARPRINTMESSAGE copies the message text to a file whose filenumber is passed to the procedure. ARGETMESSAGELINE allows you to retrieve lines of message text into your application buffer. Both procedures return a warning if no error information is currently recorded for the specified cursor.

Audit Compression

There are circumstances under which the disk process generates audit records containing less than the complete before-image and after-image of an updated data record.

Enscribe

When audit compression is enabled for an Enscribe file, the before-images and after-images of updated records are recorded as compactly as possible to save space in the audit trail (only those bytes in the data record that were actually modified are represented).

If you want to be able to retrieve complete before-images and after-images for Enscribe updates, do not enable audit compression.

NonStop SQL/MP

By default, NonStop SQL/MP applies a form of audit compression when updating tables (only the individual fields that changed are recorded).

If you want to be able to retrieve complete before-images and after-images for NonStop SQL/MP updates, you must first specify the NOAUDITCOMPRESS file attribute.

If you are reading the audit trail for a NonStop SQL/MP table whose audit records are compressed, you can use the ARFETCHFIELDVALUE procedure to retrieve the before-image and after-image of those fields that are present in field-compressed update records (both updated and key fields). The ARGETFIELDINFO procedure provides information about the fields that are present.

Reading Active Audit Files

The TMF audit-reading procedures use large physical transfers when reading audit files from disk. The audit-trail file is read from the disk into cache memory and records are then retrieved from cache into your application buffer.

Note. Updates belonging to different volumes might not be written to the audit trail in the same sequence as the updates happen in the actual transaction. For example, assume that SQL/MP (non-referential integrity) data is being replicated to SQL/MX (referential integrity) by a software tool reading the audit trail and that a parent-child RI relationship exists for two tables in SQL/MX. Assume also that the SQL/MP application starts a transaction, inserts the parent row first, then inserts the child row, and then commits. Because the DP2 disk process flush intervals can vary, this sequence of events might show up in the audit trail as insert of the child, followed by insert of the parent, followed by the commit. Only when the partitions are on the same volume is order within the audit trail assured.

If audit records are added to an audit file after the audit-reading procedures have detected the EOF, your application can retrieve them by issuing more ARREAD calls (provided you specified 999999 as the maximum sequence number in the AROPEN call that opened the cursor).

The procedure for reading audit records from an active-audit trail is as follows:

1. Open a cursor for the audit trail by calling AROPEN.

   Use a fully qualified file name or the appropriate audit-trail ID (MAT, AUX01, AUX02, ...) as the *generic-name*. When using the fully qualified file name, the volume name is meaningless because ARLIB2 consults the TMP to determine the location of the audit trail, the subvolume name must be ZTMFAT, and the filename should be the appropriate two-character identifier (AA, BB, CC, ...).

   Use the desired starting audit-file sequence number as the *min-seqno* parameter value and 999999 as the *max-seqno* parameter value.

   ARLIB2 correctly processes multiple active-audit volumes, overflow-audit volumes, and restore-audit volumes. If audit restore is necessary and enabled, the TMP uses the TMF dump/restore capability to restore audit-trail files to one of the configured restore-audit volumes.

2. Set the position at which to start reading by calling ARPOSITION.

   To read forward from the beginning of the audit file specified in the AROPEN call, set the *audit-file-seqno* parameter to the same value as the AROPEN *min-seqno* parameter, the *rel-byte-addr* parameter to zero, and the *cursor-direction* parameter to zero.

   To read in reverse from the end of the current active audit file back to the beginning of the audit file specified in the AROPEN call, set the *audit-file-seqno* parameter to 999999, the *audit-file-rba* parameter to -1, and the *cursor-direction* parameter to any value greater than zero.

3. Retrieve successive audit records by repeatedly calling ARREAD.

   When reading forward, ARE-END-OF-AUDIT indicates that you have reached the end of the current active audit file. To retrieve newly added audit records, pause briefly and then issue another ARREAD. That call will return either a new audit record (if available) or an ARE-END-OF-AUDIT message. If the call returns a new audit record, issue another ARREAD. If the call returns an ARE-END-OF-AUDIT, then pause once again and issue another ARREAD, and so forth.

   When reading in reverse, ARE-END-OF-AUDIT indicates that you have reached the beginning of the audit file specified by the *min-seqno* parameter in the AROPEN call.

Reading a Range of Audit-Trail Files

The procedure for reading a range of audit records that does not include the active audit-trail file is as follows:

1. Open a cursor for the audit trail by calling AROPEN.

   Use a fully qualified file name or the appropriate audit-trail ID (MAT, AUX01, AUX02, ...) as the *generic-name*. When using the fully qualified file name, the volume name is meaningless because ARLIB2 consults the TMP to determine the location of the audit trail, the subvolume name must be ZTMFAT, and the filename should be the appropriate two-character identifier (AA, BB, CC, ...).

   Use the desired starting audit-file sequence number as the *min-seqno* parameter value and the desired ending audit-file sequence number as the *max-seqno* parameter value.

   ARLIB2 correctly processes multiple active-audit volumes, overflow-audit volumes, and restore-audit volumes. If audit restore is necessary and enabled, the TMP uses the TMF dump/restore capability to restore audit-trail files to one of the configured restore-audit volumes.

2. Set the position at which to start reading by calling ARPOSITION.

   To read forward from the beginning of the audit file specified by the *min-seqno* parameter, set the ARPOSITION *audit-file-seqno* parameter to the same value as the *min-seqno* parameter, the ARPOSITION *rel-byte-addr* parameter to zero, and the ARPOSITION *cursor-direction* parameter to zero.

   To read in reverse from the end of the audit file specified by the *max-seqno* parameter back to the beginning of the audit file specified by the *min-seqno* parameter, set the ARPOSITION *audit-file-seqno* parameter to the same value as the *max-seqno* parameter, the ARPOSITION *rel-byte-addr* parameter to -1D, and the ARPOSITION *cursor-direction* parameter to any value greater than zero.

3. Retrieve successive audit records by repeatedly calling ARREAD.

   When reading forward, ARE-END-OF-AUDIT indicates that you have reached the end of the audit file specified by the *max-seqno* parameter in the AROPEN call.

   When reading in reverse, ARE-END-OF-AUDIT indicates that you have reached the beginning of the audit file specified by the *min-seqno* parameter in the AROPEN call.

Reading a Merged Audit Trail With a MERGE Cursor

When using the ARLIB2 MERGE cursor, the application never sees AuxPointer records. ARLIB2 merely uses those records to locate and return records from the appropriate auxiliary audit trails.

The procedure for reading the MAT and all configured auxiliary audit trails as a merged audit trail using the MERGE cursor is as follows:

1. Call AROPEN with “MERGE” as the *generic-name* for the cursor. Note that “MERGE” functions as the audit-trail designator even if no auxiliary audit trails are configured.

   Specify the desired MAT low and high file-sequence numbers (*min-seqno* and *max-seqno*). To designate the current active audit file as the high file-sequence number, specify 999999 as the *max-seqno*.

2. Set the position in the MAT at which to start reading by calling ARPOSITION.

   When you call ARPOSITION, you can also specify the desired starting position within one of the configured auxiliary audit trails. If you specify both a MAT and auxiliary position, however, they should be the values returned in an ARRECORD or else unpredictable results may occur.

3. Retrieve successive audit records by repeatedly calling ARREAD.

   When ARE-END-OF-AUDIT is returned, all audit records from the master and auxiliary audit trails within the range determined by the ARPOSITION and AROPEN calls have been returned.

   If *max-seqno* in the AROPEN call was 999999, you have read to the end of the merged audit trail. To retrieve newly added audit records, pause briefly and then issue another ARREAD. That call will return either a new audit record (if available) or an ARE-END-OF-AUDIT message. If the call returns a new audit record, issue another ARREAD. If the call returns an ARE-END-OF-AUDIT, then pause once again and issue another ARREAD, and so forth. Note that a MERGE cursor cannot determine that additional data is written to an auxiliary audit trail until the TMP writes an Aux Pointer record to the MAT.

Reading a Merged Audit Trail Without a MERGE Cursor

The procedure for reading the MAT and all configured auxiliary audit trails as a merged audit trail without using a MERGE cursor is as follows:

1. Open cursors for the MAT and all configured auxiliary audit trails by calling AROPEN.

   Within each call, use a fully qualified file name or the appropriate audit-trail ID (MAT, AUX01, AUX02, ...) as the *generic-name*. When using the fully qualified file name, the volume name is meaningless because ARLIB2 consults the TMP to determine the location of the audit trail, the subvolume name must be ZTMFAT, and the filename should be the appropriate two-character identifier (AA, BB, CC, ...).

   Within the AROPEN call for the MAT, use the desired starting audit-file sequence number as the *min-seqno* parameter value and the desired ending audit-file sequence number or 999999 as the *max-seqno* parameter value. Within the AROPEN call for each configured auxiliary audit trail, use 1 as the *min-seqno* parameter value and 999999 as the *max-seqno* parameter value to achieve the easiest implementation.

   ARLIB2 correctly processes multiple active-audit volumes, overflow-audit volumes, and restore-audit volumes. If audit restore is necessary and enabled, the TMP uses the TMF dump/restore capability to restore audit-trail files to one of the configured restore-audit volumes.

2. Using the MAT cursor, call ARREAD.

   If any record other than an AuxPointer record is returned, process the record and call ARREAD again.

   If an AuxPointer record is returned, then do as follows for each configured auxiliary audit trail:

   a. Call ARFETCHAUXPOINTER.

   b. Call ARPOSITION2 using the position information returned by the ARFETCHAUXPOINTER call. The ARPOSITION2 call provides the starting and ending positions within the auxiliary audit trail for the records to be merged before the next MAT record.

   c. Call ARREAD until ARE-END-OF-AUDIT is returned. At that point call ARREAD for the MAT again.

   If *max-seqno* in the AROPEN call was 999999 and ARE-END-OF-AUDIT is returned for the MAT, you have read to the end of the merged audit trail. To retrieve newly added audit records, pause briefly and then issue another ARREAD. That call will return either a new audit record (if available) or an ARE-END-OF-AUDIT message.

Reading Audit Records for SQL/MX Objects

The general steps for processing SQL/MX audit records are as follows:

1. Read the audit record (ARREAD).
2. Verify that the audit record is for an SQL/MX object (OBJECTTYPE field in ARRECORD).
3. Obtain the before-image and after-image column format (ARGETMXCOLUMNINFO).
4. Obtain the before image (ARFETCHMXBEFOREDATA[2]). Process the various fields using the column information returned by ARGETMXCOLUMNINFO.
5. Obtain the after image (ARFETCHMXAFTERDATA[2]). Process the various fields using the column information returned by ARGETMXCOLUMNINFO.

Distributed Transactions

The audit-reading procedures only allow you to read local audit trails. If your application is performing distributed transactions, you must incorporate the audit-reading procedures into a server that can be invoked on a remote node. You could then receive the results either by way of messages or by creating a file on the remote node and reading it across the network.

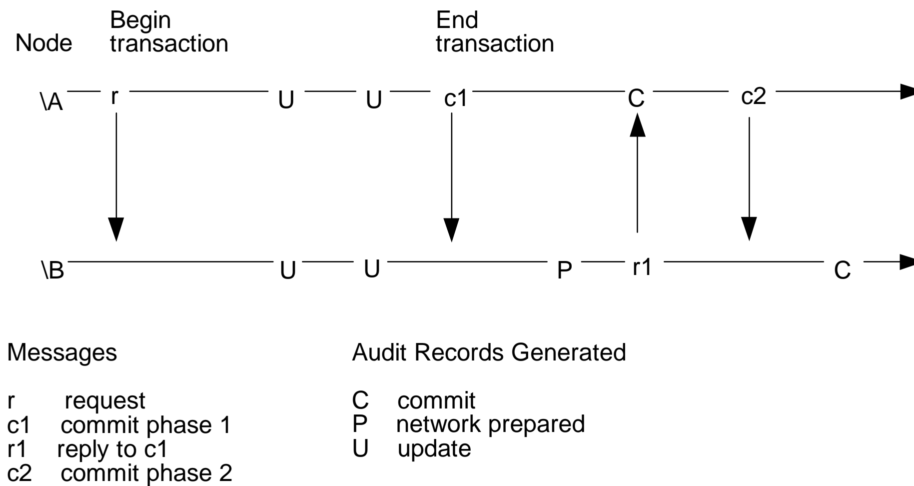
Tracing network transactions is somewhat difficult. At the local node, the first and only indication that a transaction involved changes on a remote node is found in the commit record, which contains a list of any child nodes to which it sent requests.

Because the home node is recorded in every audit record for which it is relevant, it is fairly easy to determine the node at which the transaction was initiated. This is not sufficient, however, to identify the node that relayed the transaction to the current node (that is, the parent node) because other nodes might have been involved between the home node and the current node. The network-prepared record and the commit record list the parent node (as well as any child nodes) of the current node.

To trace a distributed transaction completely, you must have servers available on all nodes that were involved in the transaction. In addition, you must be familiar with the audit trails on the other nodes (at a minimum, you must know their generic names and sequence numbers).

Figure 5-1 and Figure 5-2 show relative time lines for two distributed transactions. Figure 5-1 shows the basic relationship between a parent node and a child node.

Figure 5-1. Basic Parent-Child Relationship



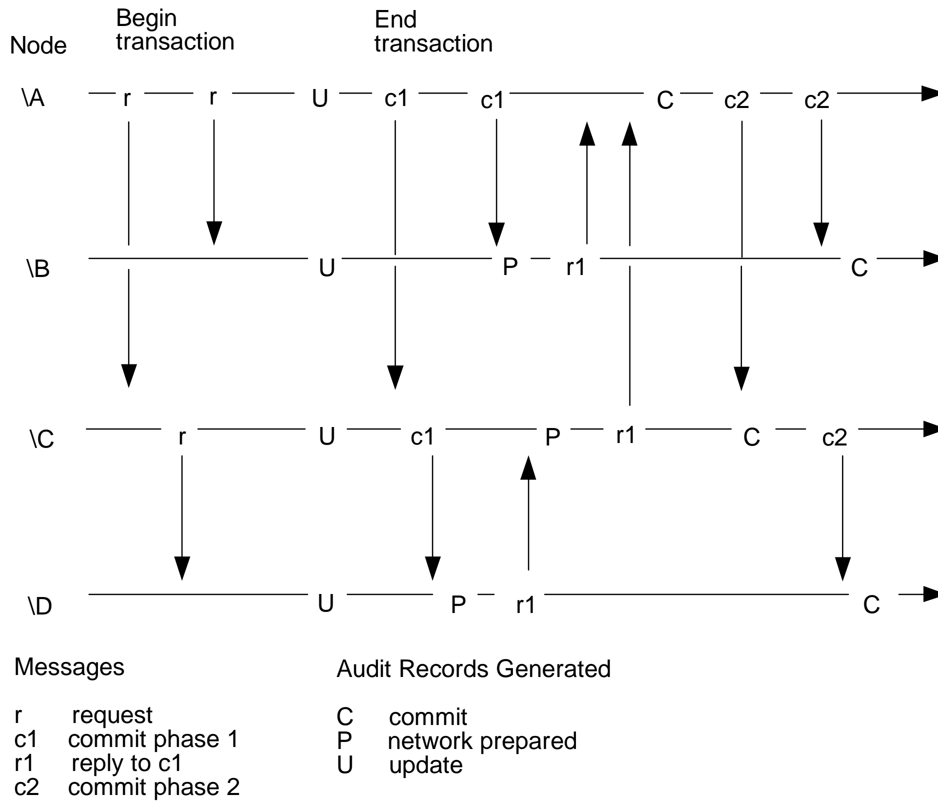
In Figure 5-1, the transaction begins at node \A. The requestor running in node \A sends a work-request message to a server in node \B. The processes in both nodes generate updates to audited data files. When the requestor in node \A is ready to commit the transaction, a phase one commit message is sent to node \B.

When node \B is ready for the transaction to be committed, the audit on node \B is flushed to disk and a reply is sent back to node \A. The transaction then commits on node \A, and node \B is notified that the transaction is to be committed. At that point the transaction also commits on node \B.

The commit record on node \A contains a list of its offspring, node \B in this case. You can retrieve this list of child nodes by calling the ARFETCHCHILDNODELIST procedure after reading the commit record.

Figure 5-2 shows how the basic relationship between a parent and a child node gets expanded for more complex transactions.

Figure 5-2. Layered Offspring Relationships



In Figure 5-2, the transaction again begins at node \A. The requestor running in node \A sends work-request messages to servers in nodes \B and \C. The server in node \C sends a further work-request message to a server in node \D. The processes in all four nodes generate updates to audited data files. When the requestor in node \A is ready to commit the transaction, a phase one commit message is sent to nodes \B and \C.

When node \B is ready for the transaction to be committed, the audit on node \B is flushed to disk and a reply is sent back to node \A. Before it can commit the transaction, node \C must first ascertain that its child at node \D is prepared to commit. After node \D flushes its audit and replies to node \C, the audit on node \C is flushed and node \A is notified that node \C is ready to commit.

Once node \A receives positive replies from both of its children, the transaction commits on node \A and a message is sent to the offspring in nodes \B and \C telling them that the transaction is to be committed. Then nodes \B and \C commit the transaction and node \D is notified that the transaction is to be committed. Finally, the transaction commits on node \D.

Auxiliary Audit Trails

Although most applications use only a master audit trail, the TMF audit-reading procedures support both master and auxiliary audit trails. The following types of audit records can appear in either the master audit trail or an auxiliary audit trail:

delete	file purge	update
file alter	file rename	update auditcomp
file create	insert	update fieldcomp

All other types of audit records can appear only in the master audit trail.

Tracing transactions that involve volumes whose audit records are directed to an auxiliary audit trail can be difficult. There is no indication within the master audit trail that a transaction also involves an auxiliary audit trail. If you are tracing a transaction that involves auxiliary audit trails, you will have to search every trail that includes the transaction and correlate the audit records by using the transid.

Subset Audit Records

When several records within the same data block are all deleted or updated in the same manner, the disk process generates a single audit record containing the before-images and after-images of all of the affected data records. Each delete or update represented within such an audit record is called a “subset audit record.”

The use of subset audit records has the following consequences:

- The SEQNO and RBA values returned by ARREAD are the same for all related subset audit records.
- If you try to position the cursor at a particular subset audit record, the cursor actually gets positioned at the beginning of the entire audit record (or at the end if you are reading in the reverse direction).

NonStop SQL/MP Internal Field Formats

Certain fields in NonStop SQL/MP records are represented on disk in a way that does not directly correspond to the external view of the data. Because TMF audit-reading procedures return record and field images as they are on disk, you must understand how the fields are represented on disk in order to interpret them.

This topic attempts to cover as many of the special cases as are currently known. The on-disk record format is subject to change from RVU to RVU because it is not considered to be an external feature of NonStop SQL/MP.

In the examples that follow, brackets ([ ]) denote a byte of storage, while an asterisk (*) denotes an undefined byte.

Field Alignment

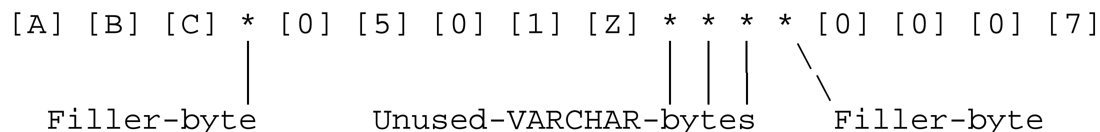
To conserve disk space, NonStop SQL/MP records are stored in a packed format. This means that bytes not needed to represent the values of fields in the record are not generally stored on disk. Each field in the record immediately follows the preceding field with no filler bytes. This means, for example, that numeric fields following odd-length fields are not necessarily aligned on word boundaries. The field can begin on either an even or odd byte, depending upon whether the preceding odd-length fields compensate for one another.

You must also consider the presence of variable-length character (VARCHAR) fields. The length of a VARCHAR field can vary between records in the same table. This means that, because the information in a record might not be at a fixed offset, you cannot simply overlay the before-image or after-image buffer with the record definition or use the offset that you might expect from an external point of view.

For example, assume that a table has the following Data Definition Language (DDL) definition:

```
RECORD REC.
 02  A  TYPE CHARACTER 3.          ! SQL type: CHARACTER
 02  B  TYPE BINARY 16,0.          ! SQL type: SMALLINT
 02  C.                            ! SQL type: VARCHAR
   04  LENGTH  TYPE BINARY 16,0.
   04  VALUE   TYPE CHARACTER 4.
 02  D  TYPE BINARY 32,0.          ! SQL type: INT
END
```

Assume that the values of a row in the table are ABC, 5, Z, and 7. The external representation would be as follows:



However, the data record image on disk and in a before-image or after-image from the audit trail actually is as follows:

```
[A] [B] [C] [0] [5] [0] [1] [Z] [0] [0] [0] [7]
```

The offset of field B is fixed (namely, the length of field A). Although field B is a numeric field, it is not word-aligned relative to the beginning of the image. Notice also that the offset of field D depends upon the length of field C (the VARCHAR field) and is not fixed for all rows in the same table.

Variable-Length Character (VARCHAR) Fields

VARCHAR fields, which store character-type data, are declared with a certain maximum length. However, any given entry can actually contain from 0 characters up to the maximum number of characters that the particular field declaration can hold.

The external DDL declaration for a VARCHAR field is as follows:

```
02  VARCHAR.
  04  LENGTH  TYPE BINARY 16,0.
  04  VALUE   TYPE CHARACTER max-length.
```

If you store the character string FRED in a VARCHAR field whose maximum length is 8, the external view of the field is then as follows:

```
[0] [4] [F] [R] [E] [D] * * * *
```

On disk and in most images in audit records, however, the unneeded bytes are dropped and the image is actually as follows:

```
[0] [4] [F] [R] [E] [D]
```

Within field-compressed update records, VARCHAR fields that belong to the primary key of a table are always represented using their maximum length with blanks appended as necessary. The character string FRED within an 8-character field in a field-compressed update record would therefore be represented as follows:

```
[0] [4] [F] [R] [E] [D] [ ] [ ] [ ] [ ]
```

You can always determine the stored length of a VARCHAR field by looking in the image of the field. The first word contains the length in bytes of the character portion. If the audit record is a field-compressed update, then you can also get this information from parameters returned by the ARFETCHFIELDVALUE or ARGETFIELDINFO procedures.

DATETIME and INTERVAL Fields

DATETIME and INTERVAL fields in NonStop SQL/MP tables store time-related information using varying units. DATETIME fields contain a value that designates a point in time. INTERVAL fields contain a value that represents a time interval or duration. Information is represented on disk depending on how the field is declared.

A DATETIME field stores values for a logically contiguous subset of the following subfields:

Subfield	Value Range	Storage
YEAR	1 to 9999	2 bytes
MONTH of year	1 to 12	1 byte
DAY of month	1 to 31	1 byte
HOUR of day	0 to 23	1 byte
MINUTE of hour	0 to 59	1 byte
SECOND of minute	1 to 59	1 byte
FRACTION of second	0 to 9(*n*)*	4 bytes

* *n* is the number of decimal digits of precision and is in the range 1 to 6. For example, FRACTION(3) has the range of values 0 to 999, which represent .000 to .999 seconds.

The length of a DATETIME field is determined by its declaration. For example, a field declared as DATETIME HOUR TO FRACTION(1) will be of length 1 + 1 + 1 + 4 = 5 bytes.

The DATE, TIME, and TIMESTAMP types are special cases of DATETIME:

DATE	is equivalent to DATETIME YEAR TO DAY
TIME	is equivalent to DATETIME HOUR TO SECOND
TIMESTAMP	is equivalent to DATETIME YEAR TO FRACTION(3)

An INTERVAL field stores an integer whose units are those of the smallest subfield in its declaration. For example, a field of the type INTERVAL HOUR(1000) TO SECOND will contain an integer number of seconds while a field of the type INTERVAL MINUTE TO FRACTION(3) will contain an integer number of microseconds.

The length of an INTERVAL field is determined by the space required to represent the largest value that can be accommodated by the declared field and is either 2, 4, or 8 bytes (that is, the field contains a 16-bit, 32-bit, or 64-bit integer).

Null Fields

A field in a NonStop SQL/MP table that can be set to a null value consists of a 1-word (2-byte) null indicator followed by the normal field itself. If the null indicator is set, then the value in the field is undefined (except that VARCHAR fields retain their length word). If the null indicator is not set, then the value in the field is valid. The value -1 in the null indicator specifies that the field is null, while the value 0 specifies that the field is not null.

Null VARCHAR fields are stored using a 0-length character part. The field length is 4 bytes, consisting of the 2-byte null indicator (set to -1) and the 2-byte VARCHAR length (set to 0), as follows:

```
[-1] [-1] [0] [0]
```

Audit Records

The various types of audit records that are accessible, and the particular fields from which you can retrieve them, are as follows.

Record Types

Table 5-1 summarizes the constants that identify each record type and the related record format name. The appropriate constant appears in the RECTYPE field of each audit record.

Table 5-1. RECTYPE Constants

Constant	Record Type	Record Format Name
1	ABORT	ABORTREC
2	COMMIT	COMMITREC
3	DELETE	DELETEREC
5	INSERT	INSERTREC
6	NETWORK-PREPARED	PREPAREDREC
7	TMF SHUTDOWN	TMF SHUTDOWNREC
10	UPDATE	UPDATEREC
11	UPDATE AUDITCOMP	UPDATEAUDITCOMPREC
12	FILE ALTER	FILE ALTERREC
13	FILE CREATE	FILECREATEREC
14	FILE PURGE	FILEPURGEREC
15	UPDATE FIELDCOMP	UPDATE FIELDCOMPREC
16	FILE RENAME	FILE RENAMEREC
17	AUX POINTER	AUX POINTERREC
18	NETWORK-COMMIT	NETWORKCOMMITREC
19	NETWORK-ABORT	NETWORKABORTREC
20	ARTYPE-CURRENTPOS	ARRECORD Header

Record Formats

Included in each of the following record descriptions is the DDL representation of those fields that are returned by the ARREAD procedure and a list of any variable-length fields that you can access by using one of the ARFETCHxxx procedures.

Each record has the following general form:

```
record ARRECORD.

 02 RECTYPE    type binary 16,0.  ! record type constant
 02 FILLER     pic x(2).          ! for alignment
 02 SEQNO      type binary 32,0.  ! audit file sequence #
 02 RBA        type binary 64,0.  ! rel byte addr of record
 02 FILLER     pic x(2).          ! for alignment
 02 AUX-INDEX  type binary 16,0.  ! for MERGE cursor
 02 AUX-SEQNO  type binary 32,0.  ! for MERGE cursor
 02 AUX-RBA    type binary 64,0.  ! for MERGE cursor
 02 TIMESTAMP  type binary 64,0.
 02 FILLER     pic x(24).
 02 BODY       pic x(136).

end  ! of record ARRECORD.
```

where the BODY field is redefined for each specific record type.

Within some of the record definitions, the DATAFILE entry refers to the following FILESPEC definition:

```
definition FILESPEC.

 02 VOLUME       type character 8.  ! $volume-name
 02 VOLUME-I     type binary 16,0 occurs 4 times
                 redefines VOLUME.
 02 SUBVOL       type character 8.  ! subvolume-name
 02 SUBVOL-I     type binary 64,0.
                 redefines SUBVOL.
 02 FILENAME     type character 8.  ! disk-filename
 02 FILENAME-I   type binary 16,0 occurs 4 times
                 redefines FILENAME.

end  ! of definition FILESPEC.
```

Within some of the record definitions, the EXTERNALNAME entry refers to the following DSMSMFILESPEC definition:

```
definition DSMSMFILESPEC.

 02 FILENAME    type character 36. ! \sys.$vdp.svol.file
                                   ! blank-filled on right
 02 FILENAME-I  type binary 16,0 occurs 18 times
                redefines FILENAME.

end  ! of definition DSMSMFILESPEC.
```

ABORT (1)

This record is generated whenever a transaction is aborted. Abort records can occur only in the master audit trail.

Variable-length fields: none

```
02 ABORTREC redefines BODY.

   04 TRANSID       type binary 64,0.
   04 HOMENODE      type binary 32,0.  ! system number
   04 FILLER        type x(4).
```

AUX POINTER (17)

This record shows ranges of audit for each auxiliary audit-trail file, creating a logical ordering among all audit records in audit trails on a given system. Aux pointer records can occur only in the master audit trail.

Variable-length fields: aux pointers

```
02 AUXPOINTERREC redefines BODY.

   04 NUMAUXTRAILS type binary 16,0.  ! # of auxiliary
                                      ! audit-trail files
```

When an AUX POINTER record is returned by ARREAD, the NUMAUXTRAILS value may increase. This is because, during the time between two AUX POINTER records, one or more new auxiliary audit trails could have been added to the audit trail configuration. When another audit trail is added to the configuration, a range of audit corresponding to that new trail will exist in every subsequent AUX POINTER record. If that happens, you should determine if it is necessary to retrieve audit from the new audit trail for the task at hand and, if so, open a new cursor(s) to read the additional audit trail(s).

It is also possible for TMF to be configured as a MAT-only environment and for the first auxiliary audit trail to be added after TMF is started. You must be able to handle and properly process an AUX POINTER record from the master audit trail, and retrieve audit, if deemed necessary, from a new auxiliary audit trail.

If the “MERGE” generic audit trail name is used in ARLIB2, then ARLIB2 will handle the new auxiliary audit trail(s) and return the audit records from the newly added trail(s).

COMMIT (2)

This record is generated whenever a transaction is made permanent by a call to ENDTRANSACTION. Commit records can occur only in the master audit trail.

Variable-length fields: child node list

```
02 COMMITREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 HOMENODE     type binary 32,0.  ! system number
  04 PARENTNODE   type binary 32,0.  ! system number
  04 NUMCHILDREN  type binary 16,0.  ! # of child nodes
  04 FILLER       type x(6).
```

DELETE (3)

This record indicates the deletion of a record from an audited file. Delete records can occur in either the master audit trail or an auxiliary audit trail.

Variable-length fields: before-image

```
02 DELETEREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 DATAFILE     type FILESPEC.
  04 EXTERNALNAME type DSMSMFILESPEC.
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
  04 HOMENODE     type binary 32,0.  ! system number
  04 BEFORELEN    type binary 32,0.  ! before-image length
```

FILE ALTER (12)

This record indicates that an attribute of an audited file (such as the file's security, the audit attribute, or maxextents) was modified. No indication of which attribute was modified is returned.

For Enscribe files, the value of transid is meaningless because attribute modifications are not done within a transaction.

Variable-length fields: none

```
02 FILEALTERREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 DATAFILE     type FILESPEC.
  04 EXTERNALNAME type DSMSMFILESPEC.!
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
  04 PURGEDATA    type binary 16,0.  ! TRUE if EOF altered
                                     ! to zero
```

FILE CREATE (13)

This record signifies the creation of an audited file.

For Enscribe files, the value of transid is meaningless because file creation is not done within a transaction.

Variable-length fields: none

```
02 FILECREATEREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 DATAFILE     type FILESPEC.
  04 EXTERNALNAME type DSMSMFILESPEC.
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
```

FILE PURGE (14)

This record indicates that an audited file was purged.

For Enscribe files, the value of transid is meaningless because file purging is not done within a transaction.

Variable-length fields: none

```
02 FILEPURGEREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 DATAFILE     type FILESPEC.
  04 EXTERNALNAME type DSMSMFILESPEC.
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
```

FILE RENAME (16)

This record indicates that an audited file has been renamed. Since Enscribe does not allow the rename of audited files, this record for all practical purposes indicates the renaming of an SQL table.

Variable-length fields: none

```
02 FILERENAMEREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 DATAFILE     type FILESPEC.
  04 EXTERNALNAME type DSMSMFILESPEC.
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
  04 NEWFILENAME  type FILESPEC.
```

INSERT (5)

This record signifies the insertion of a new record into an audited file. Insert records can occur in either the master audit trail or an auxiliary audit trail.

Variable-length fields: after-image

```
02 INSERTREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 DATAFILE     type FILESPEC.
  04 EXTERNALNAME type DSMSMFILESPEC.
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
  04 HOMENODE     type binary 32,0.  ! system number
  04 AFTERLEN     type binary 16,0.  ! after-image length
```

NETWORK-ABORT (19)

This record is generated whenever a network transaction aborts, and it only appears in the master audit trail on the parent node. The record indicates that the parent node has released its locks while simultaneously delivering abort instructions to all child nodes. A subsequent abort record (FORGOTTEN) is then written after replies to the abort instructions have been received from all child nodes.

Variable-length fields: child node list

```
02 NETWORKABORTREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 HOMENODE     type binary 32,0.  ! system number
  04 PARENTNODE   type binary 32,0.  ! system number
  04 NUMCHILDREN  type binary 16,0.  ! # of child nodes
  04 FILLER       type x(6)
```

NETWORK-COMMIT (18)

This record is generated whenever a network transaction commits, and it only appears in the master audit trail on the parent node. The record indicates that the parent node has released its locks while simultaneously delivering commit instructions to all child nodes. A subsequent commit record (FORGOTTEN) is then written after replies to the commit instructions have been received from all child nodes.

Variable-length fields: child node list

```
02 NETWORKCOMMITREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 HOMENODE     type binary 32,0.  ! system number
  04 PARENTNODE   type binary 32,0.  ! system number
  04 NUMCHILDREN  type binary 16,0.  ! # of child nodes
  04 FILLER       type x(6)
```

NETWORK-PREPARED (6)

This record is written into the master audit trail by every child (non-home) node participating in a network transaction. The record indicates that this node and all of its children, if any, are prepared to commit the transaction (child nodes generate a network-prepared record at the end of Phase 1 of the two-phase commit). Network-prepared records are sometimes referred to as “non-home flush records.”

Variable-length fields: child node list

```
02 PREPAREDREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 HOMENODE     type binary 32,0.  ! system number
  04 PARENTNODE   type binary 32,0   ! system number
  04 NUMCHILDREN  type binary 16,0.  ! # of child nodes
  04 FILLER       type x(6)
```

TMF SHUTDOWN (7)

This record signifies the successful shutdown of TMF. TMFSHUTDOWN records can only occur in the master audit trail.

Variable-length fields: none

```
02 TMFSHUTDOWNREC redefines BODY.

  04 FILLER       PIC X(2).
  ! tmf shutdown record contains the header info only
```

UPDATE (10)

This record signifies the modification of an existing record in an audited file. An update record can occur in either the master audit trail or an auxiliary audit trail.

When audit compression is enabled, update records are generated only for those records in the audited file, which are not compressed by DP2. However, for records that are compressed by DP2, the disk process generates the `update auditcomp` records for the Enscribe data files; and the `update fieldcomp` records for the NonStop SQL/MP files.

Variable-length fields: before-image and after-image

```
02 UPDATEREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 DATAFILE     type FILESPEC.
  04 EXTERNALNAME type DSMSMFILESPEC.
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
  04 HOMENODE     type binary 32,0.  ! system number
  04 BEFORELEN    type binary 32,0.  ! before-image length
  04 AFTERLEN     type binary 32,0.  ! after-image length
  04 FILLER       type x(4)
```

UPDATE AUDITCOMP (11)

This record reflects updates to an Enscribe file when audit compression is enabled. The major difference between this record and an update record is that the before-image and after-image within an update auditcomp record are encoded. Call ARFETCHFRAGMENT to get the offset, length, before-image, and after-image of each data record fragment. The recordkey field identifies the modified record.

Variable-length fields: recordkey and fragments

```
02 UPDATEAUDITCOMPREC redefines BODY.

  04 TRANSID      type binary 64,0
  04 DATAFILE     type FILESPEC
  04 EXTERNALNAME type DSMSMFILESPEC.
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
  04 HOMENODE     type binary 32,0.   ! system number
  04 KEYLEN       type binary 32,0.   ! record key length
  04 NUMFRAGS     type binary 16,0    ! #datarec frags in
                                      ! audit rec
  04 FILLER       type x(6)
```

UPDATE FIELDCOMP (15)

This record reflects the form of audit compression used by NonStop SQL/MP. Compression within this type of record is achieved by omitting fields that were not updated and that do not belong to the primary key of the file. Information about the fields contained in the audit record is available by using the ARGETFIELDINFO procedure, and the before-image and after-image of each field are available by using the ARFETCHFIELDVALUE procedure.

Variable-length fields: before-images and after-images of all updated fields and primary key fields

```
02 UPDATEFIELDCOMPREC redefines BODY.

  04 TRANSID      type binary 64,0.
  04 DATAFILE     type FILESPEC.
  04 EXTERNALNAME type DSMSMFILESPEC.
  04 OBJECTTYPE   type binary 16,0.  ! to id SQL/MX objects
  04 UNDOFLAG     type binary 16,0.
  04 HOMENODE     type binary 32,0.  ! system number
  04 FILLER       type x(4)
```

Field Descriptions

The following paragraphs briefly describe the fields returned by the ARREAD and ARFETCHxxx procedures. Note that not every field discussed is present in every type of audit record.

After-image

An after-image is a byte-for-byte copy of a record (as it was after being inserted or modified) in an audited file.

The AFTERLEN field returned by the ARREAD procedure specifies the length (in bytes) of the associated after-image field.

You use the ARFETCHAFTERIMAGE procedure to copy the after-image field from the audit record into your application buffer.

For key-sequenced files that have the DCOMPRESS (key compression in data blocks) attribute set, it is possible that an after-image retrieved from an audit record using the ARFETCHAFTERIMAGE procedure will include the leading compression-count byte.

In most cases the audit-reading procedures can detect that key compression is being used and remove its effects from the record image. There are situations, however, in which it is not possible to detect compression; in particular, when the record is involved in an insertion or update that caused a block split and the compression count for the record is 1. Whenever this occurs, ARFETCHAFTERIMAGE returns a warning code.

Aux Pointer

This 16-byte structure specifies a range of audit in an auxiliary audit trail that is logically ordered at this point relative to the records in the master audit trail. When all aux pointer records are combined, they show the logical ordering of all audit records in all audit trails on a given system.

The NUMAUXTRAILS field returned by the ARREAD procedure specifies the number of auxiliary audit ranges represented in the record.

You use the ARFETCHAUXPOINTER procedure to copy the auxiliary audit trail ranges into your application buffer. The format of the output is like the following:

```
record AUX-POINTER.
  02 BEGIN-SEQNO        type binary 32,0.
  02 FILLER             type binary 32,0.
  02 BEGIN-RBA          type binary 64,0.
  02 END-SEQNO          type binary 32,0.
  02 FILLER             type binary 32,0.
  02 END-RBA            type binary 64,0.
```

Audit records in an auxiliary audit trail are considered “in range” if the audit record begins at or after the beginning SEQNO and RBA, and ends before the ending SEQNO and RBA.

Before-image

A before-image is a byte-for-byte copy of a record (as it was before being deleted or modified) in an audited file.

The BEFORELEN field returned by the ARREAD procedure specifies the length (in bytes) of the associated before-image field.

You use the ARFETCHBEFOREIMAGE procedure to copy the before-image field from the audit record into your application buffer.

Child Node List

Each entry in this list specifies the system number of a node that is the child of the current node in a network transaction, represented as a 32-bit integer.

The NUMCHILDREN field returned by the ARREAD procedure specifies the number of entries in the associated child node list.

You use the ARFETCHCHILDNODELIST procedure to copy the child node list field from the audit record into your application buffer.

DATAFILE

This is the 12-word (24-character) local internal form of the name of the audited file upon which the action was performed ("$DATA   JOB100 PARTS   ", for example).

For files managed by the HP NonStop Storage Management Foundation (SMF) product, DATAFILE contains the logical name of the affected file. For files not managed by the SMF product, DATAFILE contains the physical filename.

Users of the TMF audit-reading procedures who change to the SMF environment need only re-compile their applications using the new audit-reading procedures to take advantage of this capability.

FRAGMENTS

This is a pair of before- and after-image fragments from a compressed Enscribe update record.

The NUMFRAGS field returned by the ARREAD procedure specifies the number of fragments in an audit-compressed update record.

You use the ARFETCHFRAGMENTS procedure to copy the before- and after-image fragments into your application buffer.

HOMENODE

This is the system number of the home node of a network transaction.

PARENTNODE

This field specifies the system number of the node that was the immediate parent of the current node in a network transaction.

For a local commit record, the value of PARENTNODE is -1D, indicating that there is no parent node.

RBA

This field specifies the relative byte address (byte offset) of the audit record within the audit-trail file. RBA is part of the cursor positioning information.

Record Key

This field is present only in update auditcomp records. The KEYLEN field returned by the ARREAD procedure specifies the length (in bytes) of the associated record key field.

For key-sequenced files, the record key field is a byte-for-byte copy of the record key. For key-sequenced files, you use the ARFETCHRECORDKEY procedure to copy the record key field from the audit record into your application buffer.

For entry-sequenced, relative, or unstructured files, the record key field represents the two-word internal form of the data record address, which is not particularly useful. You can use the ARGETRECADDR procedure to copy the record address (in external form) from the audit record into your application buffer when you encounter an update auditcomp record for one of these file types.

RECTYPE

This field specifies the type of audit record that was read. Table 5-1 shows the applicable values.

SEQNO

This field specifies the sequence number of the current audit file within the audit-trail. SEQNO is part of the cursor positioning information.

TIMESTAMP

This is a 4-word Julian timestamp, stored in the audit trail, that represents Greenwich Mean Time (GMT). If necessary, you are responsible for converting the timestamp to Local Standard Time (LST) or Local Civil Time (LCT).

When an audit record contains a timestamp, the value returned by the ARREAD procedure is the Julian value stored in the audit record. Note that many of the audit records do not contain a timestamp. For such records, TIMESTAMP refers to the timestamp in the header of the audit block containing the record; that timestamp indicates when the entire block was written to disk.

Timestamps provide only a coarse means of placing audited events in time. There are several reasons for this, one of which is the use of block header timestamps, as previously mentioned. Audit records are stored in a disk process buffer for some period of time before the block(s) containing them are actually written to disk. This means that the TIMESTAMP value from the block header will indicate a time that is later than the time when the record was generated.

Another reason for an irregular time track is that timestamps are based upon the system time as set by the system operator. It is possible for an operator to set the system time or date to an incorrect value.

Finally, separate disk processes using independent buffering can all write records to a single audit trail. This scenario could result in records that have later timestamps appearing before records that were actually generated earlier by a different disk process.

TRANSID

The TRANSID is the unique 4-word transaction identifier assigned to the particular transaction.

UNDOFLAG

The UNDOFLAG field, when set to a nonzero value, indicates that the audit record was produced by a TMF recovery process (transaction backout, volume recovery, or file recovery) as it undid the effects of a transaction. You use the UNDOFLAG field to distinguish such records from those produced on behalf of an application program.

The value of the UNDOFLAG field has the following meaning:

<> 0 the audit record was generated as part of an undo operation by a TMF recovery process (backout, volume recovery, or file recovery)

0 the audit record does not represent undo work

VOLUME

This is the 4-word (8-character) name of the affected disk volume ("$DATA ", for example).

Procedure Calls

This section provides the format of each TMF audit-reading procedure call. The descriptions are presented in alphabetic order by procedure name.

Note. The TMF audit-reading procedures cannot be called from highpin user processes. Any attempt to do so will cause a run-time error.

Table 5-2 summarizes the audit-reading procedures.

Table 5-2. TMF Audit-Reading Procedures (page 1 of 3)

Procedure Name	Function
ARCLOSE	Closes a cursor to an audit trail.
ARCOMPLETEIO	ARCOMPLETEIO is for compatibility, the application will not see ARLIB2 I/O's.
ARFETCHAFTERIMAGE	Copies the after-image field from an SQL/MP or Enscribe audit record to the application buffer.
ARFETCHAUXPOINTER	Retrieves information about the audit ranges in auxiliary audit trails.
ARFETCHBEFOREIMAGE	Copies the before-image field from an SQL/MP or Enscribe audit record to the application buffer.
ARFETCHCHILDNODELIST	Copies the child node list from an audit record to the application buffer.
ARFETCHFIELDVALUE	Retrieves both the before-image and after-image from field-compressed update record.
ARFETCHFRAGMENT	Returns information about updated data record fragments from field-compressed update record.

Table 5-2. TMF Audit-Reading Procedures (page 2 of 3)

Procedure Name	Function
ARFETCHMXAFTERDATA	Copies the after-image field from an SQL/MX audit record to the application buffer (uses bit maps).
ARFETCHMXAFTERDATA2	Copies the after-image field from an SQL/MX audit record to the application buffer (uses byte maps).
ARFETCHMXBEFOREDATA	Copies the before-image field from an SQL/MX audit record to the application buffer (uses bit maps).
ARFETCHMXBEFOREDATA2	Copies the before-image field from an SQL/MX audit record to the application buffer (uses byte maps).
ARFETCHRECORDKEY	Copies the record key field from the audit record into the application's buffer.
ARGETANSINAME	Returns the ANSI name of an SQL/MX object.
ARGETAUDRECHEADERINFO	Returns certain information from the audit-record header.
ARGETFIELDINFO	Returns information about the individual fields contained in a field-compressed update record.
ARGETMESSAGELINE	Reproduces error messages for user logging.
ARGETMXCOLUMNINFO	Returns column-format information for the SQL/MX object indicated by the current audit record.
ARGETNETWORKRECS	Enables the returning of network-related audit records when the ARREAD procedure is called.
ARGETNONDATACHNGRECS	Causes certain audit records that do not reflect changes to customer data to be returned instead of being discarded by the ARREAD procedure.
ARGETRECADDR	Returns the 32-bit record address of the modified data record (for FORMAT1 or non-oversized FORMAT2 entry-sequenced, relative, and unstructured files.).
ARGETRECADDR64	Returns the 64-bit record address of the modified data record (for FORMAT1, FORMAT2, or oversized FORMAT2 entry-sequenced, relative, and unstructured files).
AROPEN	Opens a cursor for reading audit records from a particular audit trail.
ARPOSITION	Positions a cursor within an audit trail.
ARPOSITION2	Uses the *aux-trail-range* returned by an ARFETCHAUXPOINTER call to position the cursor for the specified auxiliary audit trail.
ARPRINTMESSAGE	Reproduces error messages for user logging.
ARREAD	Reads a single record from the audit trail and retrieves selected fields from the record.

Table 5-2. TMF Audit-Reading Procedures (page 3 of 3)

Procedure Name	Function
ARSETOPTIONS	Sets several different options that control audit-record processing.
ARSTART	Initiates audit reading.
ARSTOP	Terminates audit reading.
ARSTOPNETWORKRECS	Disables the returning of network-related audit records when the ARREAD procedure is called.
ARSTOPNONDATACHNGRECS	Causes certain audit records that do not reflect changes to customer data to be discarded instead of being returned by the ARREAD procedure.

ARCLOSE

This procedure closes an open cursor. Closing a cursor ends its association with a particular audit trail and makes that cursor available for reassignment by a subsequent call to the AROPEN procedure.

```
CALL ARCLOSE     ( return-code                              ! o
                 , cursor-number                            ! i
                 , [sub-system]                             ! o
                 , [ar-error] );                            ! o
```

return-code — output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

cursor-number — input

INT:value

is the number identifying the particular open cursor to be closed.

sub-system — output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error — output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARCOMPLETEIO

This procedure is provided for compatibility only. AWAITIO will not complete any ARLIB2 nowait I/O.

```
CALL ARCOMPLETEIO ( return-code                              ! o
                  , count-transferred                        ! o
                  , tag );                                   ! i
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

count-transferred input

INT:value

is the value of the *count-transferred* parameter returned by AWAITIOX.

tag input

INT(32):value

is the value of the *tag* value returned by AWAITIOX.

ARFETCHAFTERIMAGE

This procedure copies the after-image field from the most recently read audit record into the application's buffer. This procedure works with records of type Update (10) and Insert (5). This procedure cannot be used with audit records for SQL/MX objects. The object type can be determined by checking the OBJECTTYPE field in the ARRECORD.

```
CALL ARFETCHAFTERIMAGE ( return-code                  ! o
                       , buffer                       ! o
                       , max-copy-length              ! i
                       ,[ maybe-comp-byte ] );        ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) Bit 14, when set, indicates that the first byte of the after-image is a 1 (possibly representing a compression count rather than the actual first byte of data if data compression is enabled). The *Enscribe Programmer's Guide* and the *HP NonStop SQL/MP Reference Manual* discuss the DCOMPRESS attribute.

Bit 15, when set, indicates that the after-image is longer than *max-copy-length*. At least one byte was truncated.

buffer output

INT .EXT:ref:*

is a buffer in the application process in which the information returned by ARFETCHAFTERIMAGE is stored. Note that *buffer* can be allocated from the 64K data stack or from an extended segment.

max-copy-length input

INT(32):value

is the maximum number of bytes to copy into the buffer. This routine returns an error for values of *max-copy-length* less than 1.

maybe-comp-byte output, optional

STRING .EXT:ref:1

returns the single byte which may be the actual first byte of the after-image in cases where warning <return-code>.<14> is returned. If the data file has the DCOMPRESS attribute set, then this is the actual first byte of the after-image, and

should be substituted for <buffer>[0].<0:7>. If the DCOMPRESS attribute is not set for the data file, or the warning <return-code>.<14> is not returned by ARFETCHAFTERIMAGE, then the value returned in this parameter is meaningless.

ARFETCHAUXPOINTER

This procedure retrieves information about the ranges of audit in auxiliary audit trails that are logically ordered at this point relative to audit records in the master audit trail. Together, these ranges create a logical ordering among all audit records in all audit trails on a given system.

```
CALL ARFETCHAUXPOINTER  ( return-code                  ! o
                        , audit-trail-index            ! i
                        , aux-trail-range );           ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

audit-trail-index input

INT:value

specifies the index of an auxiliary audit trail in the TMF configuration whose audit-trail position is to be returned from the current AUXPOINTER audit record. The specified value must be an integer between 1 and 15, but not greater than the value of NUMAUXTRAILS in the current AUXPOINTER audit record. A value of 1 specifies auxiliary audit trail AUX01, 2 specifies AUX02, and so forth.

aux-trail-range output

INT .EXT:ref:*

points to a caller-allocated buffer into which the AUXPOINTER information for the specified auxiliary audit trail will be returned. The format of this information is as follows:

```
record AUXPOINTERINFO
02  LOWPOS    type ATLOCSPEC.
02  HIGHPOS   type ATLOCSPEC.

record ATLOCSPEC
02  SEQ-NO    type binary 32, 0.
02  FILLER    type x(4).
02  RBA       type binary 64, 0.
```

ARFETCHBEFOREIMAGE

This procedure copies the before-image field from the most recently read audit record into the application's buffer. This procedure works with records of type Update (10) and Delete (3). This procedure cannot be used with audit records for SQL/MX objects. The object type can be determined by checking the OBJECTTYPE field in the ARRECORD.

```
CALL ARFETCHBEFOREIMAGE ( return-code                  ! o
                        , buffer                       ! o
                        , max-copy-length );           ! i
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	Bit 15, when set, indicates that the before image is longer than *max-copy-length.* At least one byte was truncated from the end.

buffer output

INT .EXT:ref:*

is a buffer in the application process in which the information returned by ARFETCHBEFOREIMAGE is stored. Note that *buffer* can be allocated from the 64K data stack or from an extended segment.

max-copy-length input

INT(32):value

is the maximum number of bytes to copy into the buffer.

ARFETCHBEFOREIMAGE returns an error for values of *max-copy-length* less than 1.

ARFETCHCHILDNODELIST

This procedure copies the child node list from the most recently read audit record into the application's buffer.

```
CALL ARFETCHCHILDNODELIST ( return-code                    ! o
                          , buffer                         ! o
                          , max-copy-length );             ! i
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	Bit 15, when set, indicates that the child node list is longer than *max-copy-length*. At least one of the elements was truncated from the end.

buffer output

INT .EXT:ref:*

is a buffer in the application process in which the information returned by ARFETCHCHILDNODELIST is stored.

max-copy-length input

INT:value

is the maximum number of bytes to copy into the buffer.

ARFETCHCHILDNODELIST returns an error for values of *max-copy-length* less than 4.

ARFETCHFIELDVALUE

This procedure retrieves both the before-image and after-image from a field-compressed update (UPDATE FIELDCOMP) record. These are SQL-only records. This procedure cannot be used with audit records for SQL/MX objects. The object type can be determined by checking the OBJECTTYPE field in the ARRECORD.

```
CALL ARFETCHFIELDVALUE  ( return-code                        ! o
                        , field-number                       ! i
                        , [ before-image ] )                 ! o
                        , [ max-before-copy-len ] )          ! i
                        , [ actual-before-len ] )            ! o
                        , [ after-image ] )                  ! o
                        , [ max-after-copy-len ] )           ! i
                        , [ actual-after-len ]               ! o
                        , [ next-field ]                     ! o
                        , [ sub-system ]                     ! o
                        , [ ar-error ] );                    ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	Bit 14, when set, indicates that the before image is longer than *max-before-copy-len.* At least one byte was truncated from the end. Bit 15, when set, indicates that the after image is longer than *max-after-copy-len.* At least one byte was truncated from the end.

field-number input

INT:value

is the zero-based ordinal field number within the data record of the field whose before-image and/or after-image is to be retrieved. This value can be found in the NonStop SQL/MP catalog (the COLNUMBER field of the COLUMNS table).

before-image output, optional

INT .EXT:ref:*

is a buffer in the application process in which the before-image of the field is returned.

max-before-copy-len input, optional

INT:value

is the maximum number of bytes to copy into *before-image.* This value must be supplied if *before-image* is present.

ARFETCHFIELDVALUE returns an error for values of *max-before-copy-len* less than 1.

actual-before-len output, optional

INT .EXT:ref:1

returns the actual length of the retrieved before-image of the field. For a variable-length character (VARCHAR) field, this value includes the 2-byte length word of the field. For null fields, this count includes the two bytes occupied by the null indicator.

after-image output, optional

INT .EXT:ref:*

is a buffer in the application process in which the after-image of the field is returned.

max-after-copy-len input, optional

INT:value

is the maximum number of bytes to copy into *after-image*. This value must be supplied if *after-image* is present.

ARFETCHFIELDVALUE returns an error for values of *max-after-copy-len* less than 1.

actual-after-len output, optional

INT .EXT:ref:1

returns the actual length of the retrieved after-image of the field. In the case of a variable-length character (VARCHAR) field, this value includes the 2-byte length word of the field. For fields that can be set to a null value, this count includes the 2 bytes occupied by the null indicator.

next-field output, optional

INT .EXT:ref:1

returns the next available field in the current audit record.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

- The modified data file must be accessible.
- If a file that has the same name as the modified file, but different characteristics, exists on disk and is accessible, the results of ARFETCHFIELDVALUE will most likely be incorrect. The TMF audit-reading procedures cannot always detect this situation.
- The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARFETCHFRAGMENT

This procedure returns information about updated data record fragments from audit-compressed update records. Use the ARFETCHFRAGMENT procedure call after reading a record of type UPDATE AUDITCOMP to get the offset, length, before-image, and after-image of each data record fragment. This procedure works with records of type UPDATE AUDITCOMP (11).

```
CALL ARFETCHFRAGMENT ( return-code                   ! o
                      ,frag-number                   ! i
                      ,[ frag-offset ]               ! o
                      ,[ before-image-buf ]          ! o
                      ,[ max-before-copy-len ]       ! i
                      ,[ actual-before-len ]         ! o
                      ,[ after-image-buf ]           ! o
                      ,[ max-after-copy-len ]        ! i
                      ,[ actual-after-len ] );       ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) Bit 14 indicates that the before-image of the fragment is longer than *max-before-copy-length*. At least one byte was truncated from the end.

Bit 15 indicates that the after-image of the fragment is longer than *max-after-copy-length*. At least one byte was truncated from the end.

frag-number input

INT:value

is the zero-based ordinal number of the updated data record fragment to retrieve.

frag-offset output, optional

INT .EXT:ref

is the zero-based byte offset within the data record where the specified fragment begins.

before-image-buf output, optional

INT .EXT:ref:*

is a buffer in the application process in which the before-image of the updated data record fragment is returned.

max-before-copy-len input, optional

INT:value

is the maximum number of bytes to copy into *before-image.* This must be supplied if *before-image-buf* is supplied.

ARFETCHFRAGMENT will return an error for values of *max-before-copy-len* less than 1.

actual-before-len output, optional

INT .EXT:ref

is the actual byte length of the before-image of the fragment.

after-image-buf output, optional

INT .EXT:ref:*

is a buffer in the application process in which the after-image of the updated data record fragment is returned.

max-after-copy-len input, optional

INT:value

is the maximum number of bytes to copy into *after-image.* This must be supplied if *after-image-buf* is supplied.

ARFETCHFRAGMENT will return an error for values of *max-after-copy-len* less than 1.

actual-after-len output, optional

INT .EXT:ref

is the actual byte length of the after-image of the fragment.

Considerations

- ARFETCHFRAGMENT may be called without supplying the before-image and after-image buffers. This usage may be useful in determining an appropriate buffer size to allocate for successive calls that have those buffers supplied.

ARFETCHMXAFTERDATA

This procedure retrieves column data from the after image of audit records for SQL/MX objects. ARFETCHMXAFTERDATA can only be used with data-fork audit records for SQL/MX objects. The object type can be determined by checking the OBJECTTYPE field in the ARRECORD. ARFETCHMXAFTERDATA copies the data for all columns from the after image of the current audit record into a buffer specified by the application. ARFETCHMXAFTERDATA works with audit records of type UPDATE (10), UPDATE FIELDCOMP (15), and INSERT (5).

```
CALL ARFETCHMXAFTERDATA ( return-code                       ! o
                        , request-bitmap                    ! i
                        , reply-bitmap                      ! o
                        , bitmap-length                     ! i
                        , image-buffer                      ! o
                        , image-buffer-length               ! i
                        , end-image-data-offset             ! o
                        , reply-hint                        ! o
                        , [sub-system ]                     ! o
                        , [ar-error ] );                    ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

request-bitmap input

INT(32) .EXT:ref:*

must be *bitmap-length* bytes long. The setting of the various bits, however, does not matter. *request-bitmap* and *reply-bitmap* must not overlap one another. For best performance, *request-bitmap* and *reply-bitmap* should be allocated as INT(32) variables.

reply-bitmap output

INT(32) .EXT:ref:*

is *bitmap-length* bytes long. Each bit represents a column in the output buffer. Upon return, a bit is zero if data is not present in the corresponding column or one if data is present. If *reply-bitmap* is not large enough to represent all of the columns in the returned output, then data about columns corresponding to the missing bits is not returned. *request-bitmap* and *reply-bitmap* must not overlap one another. For best performance, *request-bitmap* and *reply-bitmap* should be allocated as INT(32) variables.

bitmap-length input

INT(32):value

specifies the length of *request-bitmap* and *reply-bitmap* in bytes, and must be mod 4.

image-buffer output

FIXED .EXT:ref

points to a caller-allocated buffer where the image data is returned. The information is returned in a fixed format defined by SQL/MX with room for all the columns, regardless of how many are requested. VARCHAR columns have their maximum lengths allocated. The performance of this procedure is improved if this buffer starts on a mod 8 address, and is mod 8 in length.

Note. The data within the image buffer returned by ARFETCHMXAFTERDATA is not aligned on any boundary. You must use a string move to move numeric values from the image buffer to an aligned numeric field before processing the value. The same is true for NULL indicator fields and VARCHAR character length fields, and their length fields.

image-buffer-length input

INT(32):value

specifies the length in bytes of the caller-allocated image buffer. The performance of this procedure is improved if this length is mod 8.

If this length is not large enough to contain the data for all the columns in the record (plus fillers, as indicated by the information returned by the ARGETMXCOLUMNINFO procedure), an ARE-BUFFER-TOO-SMALL error is returned and the needed length is returned in reply hint.

end-image-data-offset output

INT(32) .EXT:ref

specifies the offset in the image buffer of the byte after the end of the last column (highest column number) for which image data is being returned by this call.

reply-hint output

INT(32) .EXT:ref

is a hint returned by TMFARLB2.

If the return code is AR-OK, this parameter contains the highest column number for which image data is being returned.

If the return code is ARE-BUFFER-TOO-SMALL, this parameter contains the minimum length necessary for the image buffer in bytes (adjusted to mod 8).

For all other return-code values this parameter contains zero.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARFETCHMXAFTERDATA2

This procedure is the same as ARFETCHMXAFTERDATA except that it uses byte maps instead of bit maps. Byte maps are easier to use for languages that have difficulty checking and setting bits.

This procedure retrieves column data from the after image of audit records for SQL/MX objects. ARFETCHMXAFTERDATA2 can only be used with data-fork audit records for SQL/MX objects. The object type can be determined by checking the OBJECTTYPE field in the ARRECORD. ARFETCHMXAFTERDATA2 copies the data for all columns from the after image of the current audit record into a buffer specified by the application. ARFETCHMXAFTERDATA2 works with audit records of type UPDATE (10), UPDATE FIELDCOMP (15), and INSERT (5).

```
CALL ARFETCHMXAFTERDATA2 ( return-code                    ! o
                         , request-bytemap                ! i
                         , reply-bytemap                  ! o
                         , bytemap-length                 ! i
                         , image-buffer                   ! o
                         , image-buffer-length            ! i
                         , end-image-data-offset          ! o
                         , reply-hint                     ! o
                         , [sub-system ]                  ! o
                         , [ar-error ] );                 ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

request-bytemap input

STRING .EXT:ref:*

must be *bytemap-length* bytes long. The setting of the various bytes, however, does not matter. *request-bytemap* and *reply-bytemap* must not overlap one another.

reply-bytemap output

STRING .EXT:ref:*

is *bytemap-length* bytes long. Each byte represents a column in the output buffer. Upon return, a byte is 0x00 if data is not present in the corresponding column or 0xFF if data is present. If *reply-bytemap* is not large enough to represent all of the columns in the returned output, then data about columns

corresponding to the missing bytes is not returned. *request-bytemap* and *reply-bytemap* must not overlap one another.

bytemap-length input

INT(32):value

specifies the length of *request-bytemap* and *reply-bytemap* in bytes, and can be any length.

image-buffer output

FIXED .EXT:ref

points to a caller-allocated buffer where the image data is returned. The information is returned in a fixed format defined by SQL/MX with room for all the columns, regardless of how many are requested. VARCHAR columns have their maximum lengths allocated. The performance of this procedure is improved if this buffer starts on a mod 8 address, and is mod 8 in length.

Note. The data within the image buffer returned by ARFETCHMXAFTERDATA2 is not aligned on any boundary. You must use a string move to move numeric values from the image buffer to an aligned numeric field before processing the value. The same is true for NULL indicator fields and VARCHAR character length fields, and their length fields.

---

image-buffer-length input

INT(32):value

specifies the length in bytes of the caller-allocated image buffer. The performance of this procedure is improved if this length is mod 8.

If this length is not large enough to contain the data for all the columns in the record (plus fillers, as indicated by the information returned by the ARGETMXCOLUMNINFO procedure), an ARE-BUFFER-TOO-SMALL error is returned and the needed length is returned in reply hint.

end-image-data-offset output

INT(32) .EXT:ref

specifies the offset in the image buffer of the byte after the end of the last column (highest column number) for which image data is being returned by this call.

reply-hint output

INT(32) .EXT:ref

is a hint returned by TMFARLB2.

If the return code is AR-OK, this parameter contains the highest column number for which image data is being returned.

If the return code is ARE-BUFFER-TOO-SMALL, this parameter contains the minimum length necessary for the image-buffer in bytes (adjusted to mod 8).

For all other return-code values this parameter contains zero.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARFETCHMXBEFOREDATA

This procedure retrieves column data from the before image of audit records for SQL/MX objects. ARFETCHMXBEFOREDATA can only be used with data-fork audit records for SQL/MX objects. The object type can be determined by checking the OBJECTTYPE field in the ARRECORD. ARFETCHMXBEFOREDATA copies the data for all columns from the before image of the current audit record into a buffer specified by the application. ARFETCHMXBEFOREDATA works with audit records of type UPDATE (10), UPDATE FIELDCOMP (15), and DELETE (3).

```
CALL ARFETCHMXBEFOREDATA ( return-code                 ! o
                         , request-bitmap              ! i
                         , reply-bitmap                ! o
                         , bitmap-length               ! i
                         , image-buffer                ! o
                         , image-buffer-length         ! i
                         , end-image-dataoffset        ! o
                         , reply-hint                  ! o
                         , [sub-system ]               ! o
                         , [ar-error ] );              ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

request-bitmap input

INT(32) .EXT:ref:*

must be *bitmap-length* bytes long. The setting of the various bits, however, does not matter. *request-bitmap* and *reply-bitmap* must not overlap one another. For best performance, *request-bitmap* and *reply-bitmap* should be allocated as INT(32) variables.

reply-bitmap output

INT(32) .EXT:ref:*

is *bitmap-length* bytes long. Each bit represents a column in the output buffer. Upon return, a bit is zero if data is not present in the corresponding column or one if data is present. If *reply-bitmap* is not large enough to represent all of the columns in the returned output, then data about columns corresponding to the missing bits is not returned. *request-bitmap* and *reply-bitmap* must not overlap one another. For best performance, *request-bitmap* and *reply-bitmap* should be allocated as INT(32) variables.

bitmap-length input

INT(32):value

specifies the length of *request-bitmap* and *reply-bitmap* in bytes, and must be mod 4.

image-buffer output

FIXED .EXT:ref

points to a caller-allocated buffer where the image data is returned. The information is returned in a fixed format defined by SQL/MX with room for all the columns, regardless of how many are requested. VARCHAR columns have their maximum lengths allocated. The performance of this procedure is improved if this buffer starts on a mod 8 address, and is mod 8 in length.

Note. The data within the image buffer returned by ARFETCHMXBEFOREDATA is not aligned on any boundary. You must use a string move to move numeric values from the image buffer to an aligned numeric field before processing the value. The same is true for NULL indicator fields and VARCHAR character length fields, and their length fields.

---

image-buffer-length input

INT(32):value

specifies the length in bytes of the caller-allocated image buffer. The performance of this procedure is improved if this length is mod 8.

If this length is not large enough to contain the data for all the columns in the record (plus fillers, as indicated by the information returned by the ARGETMXCOLUMNINFO procedure), an ARE-BUFFER-TOO-SMALL error is returned and the needed length is returned in reply hint.

end-image-dataoffset output

INT(32) .EXT:ref

specifies the offset in the image buffer of the byte after the end of the last column (highest column number) for which image data is being returned by this call.

reply-hint output

INT(32) .EXT:ref

is a hint returned by TMFARLB2.

If the return code is AR-OK, this parameter contains the highest column number for which image data is being returned.

If the return code is ARE-BUFFER-TOO-SMALL, this parameter contains the minimum length necessary for the image-buffer in bytes (adjusted to mod 8).

For all other return-code values this parameter contains zero.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARFETCHMXBEFOREDATA2

This procedure is the same as ARFETCHMXBEFOREDATA except that it uses byte maps instead of bit maps. Byte maps are easier to use for languages that have difficulty checking and setting bits.

This procedure retrieves column data from the before image of audit records for SQL/MX objects. ARFETCHMXBEFOREDATA2 can only be used with data-fork audit records for SQL/MX objects. The object type can be determined by checking the OBJECTTYPE field in the ARRECORD. ARFETCHMXBEFOREDATA2 copies the data for all columns from the before image of the current audit record into a buffer specified by the application. ARFETCHMXBEFOREDATA2 works with audit records of type UPDATE (10), UPDATE FIELDCOMP (15), and DELETE (3).

```
CALL ARFETCHMXBEFOREDATA2 ( return-code              ! o
                          , request-bytemap          ! i
                          , reply-bytemap            ! o
                          , bytemap-length           ! i
                          , image-buffer             ! o
                          , image-buffer-length      ! i
                          , end-image-dataoffset     ! o
                          , reply-hint               ! o
                          , [sub-system ]            ! o
                          , [ar-error ] );           ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

request-bytemap input

STRING .EXT:ref:*

must be *bytemap-length* bytes long. The setting of the various bytes, however, does not matter. *request-bytemap* and *reply-bytemap* must not overlap one another.

reply-bytemap output

STRING .EXT:ref:*

is *bytemap-length* bytes long. Each byte represents a column in the output buffer. Upon return, a byte is 0x00 if data is not present in the corresponding column or 0xFF if data is present. If *reply-bytemap* is not large enough to represent all of the columns in the returned output, then data about columns

corresponding to the missing bytes is not returned. *request-bytemap* and *reply-bytemap* must not overlap one another.

bytemap-length input

INT(32):value

specifies the length of *request-bytemap* and *reply-bytemap* in bytes, and can be any length.

image-buffer output

FIXED .EXT:ref

points to a caller-allocated buffer where the image data is returned. The information is returned in a fixed format defined by SQL/MX with room for all the columns, regardless of how many are requested. VARCHAR columns have their maximum lengths allocated. The performance of this procedure is improved if this buffer starts on a mod 8 address, and is mod 8 in length.

Note. The data within the image buffer returned by ARFETCHMXBEFOREDATA2 is not aligned on any boundary. You must use a string move to move numeric values from the image buffer to an aligned numeric field before processing the value. The same is true for NULL indicator fields and VARCHAR character length fields, and their length fields.

---

image-buffer-length input

INT(32):value

specifies the length in bytes of the caller-allocated image buffer. The performance of this procedure is improved if this length is mod 8.

If this length is not large enough to contain the data for all the columns in the record (plus fillers, as indicated by the information returned by the ARGETMXCOLUMNINFO procedure), an ARE-BUFFER-TOO-SMALL error is returned and the needed length is returned in reply hint.

end-image-dataoffset output

INT(32) .EXT:ref

specifies the offset in the image buffer of the byte after the end of the last column (highest column number) for which image data is being returned by this call.

reply-hint output

INT(32) .EXT:ref

is a hint returned by TMFARLB2.

If the return code is AR-OK, this parameter contains the highest column number for which image data is being returned.

If the return code is ARE-BUFFER-TOO-SMALL, this parameter contains the minimum length necessary for the image-buffer in bytes (adjusted to mod 8).

For all other return-code values this parameter contains zero.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARFETCHRECORDKEY

This procedure copies the record key field from the most recently read audit record into the application's buffer. This procedure works with records of type UPDATE AUDITCOMP (11).

```
CALL ARFETCHRECORDKEY ( return-code                    ! o
                      , buffer                         ! o
                      , [ max-copy-length ] );         ! i
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the various error codes.
Warnings (>0)	Bit 15, when set, indicates that the record key is longer than *max-copy-length*. At least one byte was truncated from the end.

buffer output

INT .EXT:ref:*

is a buffer in the application process in which the information returned by ARFETCHRECORDKEY is stored. Note that the buffer can be allocated either from the 64K data stack or from an extended segment.

max-copy-length input, optional

INT:value

is the maximum number of bytes to copy into the buffer.

ARFETCHRECORDKEY returns an error for values of *max-copy-length* less than 1.

Considerations

- The record key field only occurs in UPDATE AUDITCOMP audit records.
- The value of a record key is only meaningful if the audit (data) file is key-sequenced. For other types of files (entry-sequenced, relative, or unstructured), the field contains the two-word internal form of the data record address.

  Because audit records do not specify the file type, the TMF audit-reading procedures cannot distinguish between an internal record address and a two-word key, either of which can be returned. If the audited file is accessible, you can find out its type by calling the FILEINFO system procedure.
- The ARGETRECADDR procedure returns the data record address for changes to entry-sequenced, relative, or unstructured files (the file must be accessible). You can use ARGETRECADDR with any data change audit record (insert, delete, and any form of update).

ARGETANSINAME

This procedure returns the ANSI name of an SQL/MX object. It also returns the ANSI partition name of the object (where pertinent) and the name space of the returned ANSI name.

Using the ANSI name, the calling program can perform SQL/MX SELECT statements against the SQL/MX metadata to obtain additional information about that object not available from the ARGETMXCOLUMNINFO procedure.

Each partition of an SQL/MX table has another file associated with it called a resource fork that contains additional label information. Actually, all SQL/MX partitions/files except resource forks have resource forks associated with them.

If this procedure is called using the Guardian name of a resource fork, the information pertaining to the SQL/MX partition/file that the resource fork is associated with is returned.

```
CALL ARGETANSINAME ( return-code                          ! o
                   , guardian-name                        ! i
                   , guardian-name-length                 ! i
                   , ansi-name                            ! o
                   , ansi-name-buffer-length              ! i
                   , ansi-name-length                     ! o
                   , namespace                            ! i
                   , partition-name                       ! o
                   , partition-name-buffer-length         ! i
                   , partition-name-length                ! o
                   , [sub-system ]                        ! o
                   , [ar-error ] );                       ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

If the format of the specified Guardian name is not valid, an ARE-INVALID-PARAM error is returned.

If the specified Guardian name does not map to the SQL/MX name space, an ARE-ONLY-USABLE-WITH-SQLMX error is returned. This determination is made by examining the Guardian name, not by accessing the file on disk that has that Guardian name.

If the file that has the specified Guardian name does not exist on disk, an ARE-DATA-FILE-NOT-FOUND error is returned.

If the ANSI name being returned is too long to fit in the caller-allocated ANSI-name buffer, or the partition name being returned is too long to fit in the caller-allocated partition-name buffer, or both, an ARE-BUFFER-TOO-SMALL error is returned. No name information is returned in either name buffer in such cases, but the minimum buffer lengths needed are returned in both the *ansi-name-length* and *partition-name-length* parameters.

guardian-name input

STRING .EXT:ref:*

specifies the Guardian name. This is a pointer to a buffer of type DSMSMFILESPEC that is allocated by the calling process.

guardian-name-length input

INT(32):value

is the length in bytes of *guardian-name*.

ansi-name output

STRING .EXT:ref:*

is the corresponding ANSI name (in external format). This is a pointer to a buffer of length *ansi-name-buffer-length* bytes that is allocated by the calling process.

The format of an ANSI name is <Catalog>.<Schema>.<Object>.

ansi-name-buffer-length input

INT(32):value

is the length in bytes of the caller-allocated buffer for the returned ANSI name. The maximum length necessary for this buffer is the maximum length in bytes of an external format ANSI name: ((((128 * 2) + 2) * 3) + 2).

ansi-name-length output

INT(32) .EXT:ref

is the length in bytes of the returned ANSI name string.

namespace input

STRING .EXT:ref:2

is the name space of the returned ANSI name. Because the ANSI names of an SQL/MX base table and an SQL/MX index can be the same, the name space is a necessary adjunct to an ANSI name to prevent name clashes.

The defined ANSI name spaces are as follows:

TA = Base Table or View

IX = Index

IL = Log Table for Materialized Views (not returned in this interface)

RL = Range Log Table for Materialized Views (not returned in this interface)

TT = Trigger Temporary Table

= Unknown Namespace

partition-name output

STRING .EXT:ref:1

is the corresponding partition name (in external format). This is a pointer to a buffer of length *partition-name-buffer-length* bytes that is allocated by the calling process. A partition name is returned only when the specified Guardian name is for an SQL/MX base-table partition or index partition. Otherwise, when there is no associated partition name, zero is returned in *partition-name-length*.

partition-name-buffer-length input

INT(32):value

is the length in bytes of the caller-allocated buffer for the returned partition name. The maximum length necessary for this buffer is ((128 * 2) + 2) bytes.

partition-name-length output

INT(32) .EXT:ref

is the length in bytes of the returned partition-name string.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARGETAUDRECHEADERINFO

This procedure returns certain information from the audit-record header.

```
CALL ARGETAUDRECHEADERINFO ( return-code                     ! o
                       , [ file-format ]                     ! o
                       , [ blocksize ]                       ! o
                       , [ rsn-rbn ]                         ! o
                       , [ recnum-offset ] ) EXTENSIBLE;     ! o
```

return-code output

INT .EXT:ref:1

indicates the outcome of the call.

file-format output

INT .EXT:ref

indicates the file format (0 = format-1 files, 1 = format-2 files).

blocksize output

INT .EXT:ref

indicates the block size of the file (0 = 512, 1 = 1024, 2 = 2048, 3 = 4096)

rsn-rbn output

INT(32) .EXT:ref

indicates the relative sector number (rsn) for format-1 files or the relative block number (rbn) for format-2 files.

recnum-offset output

INT(32) .EXT:ref

indicates the record number within this block for relative files and entry sequence files, or the offset within the block for unstructured files.

ARGETFIELDINFO

This procedure returns information about the fields contained in a field-compressed update audit record. This procedure works only with SQL records of type UPDATE FIELDCOMP (15). This procedure cannot be used with audit records for SQL/MX objects (see the ARGETMXCOLUMNINFO procedure). The object type can be determined by checking the OBJECTTYPE field in the ARRECORD.

```
CALL ARGETFIELDINFO ( return-code                          ! o
                    , field-number                         ! i
                    , [ field-type ]                       ! o
                    , [ field-length ]                     ! o
                    , [ is-key-length ]                    ! o
                    , [ next-field ]                       ! o
                    , [ null-allowed ]                     ! o
                    , [ flags ]                            ! o
                    , [ collation-def ]                    ! o
                    , [sub-system ]                        ! o
                    , [ar-error ] );                       ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

field-number input

INT:value

is the zero-based ordinal field number within the data record of the field about which information is to be returned. This value can be found in the NonStop SQL/MP catalog (the COLNUMBER field of the COLUMNS table).

field-type output, optional

INT .EXT:ref:1

is one of a set of constants that identify the SQL type of the field, as indicated below. Note that these correspond to the field type values found in the NonStop SQL/MP Catalog (in the FSDATATYPE field of the COLUMNS Table).

Value	Field Type	Literal Name
0	Fixed-length ASCII character	ARVALUE-ASCII-F
1	Fixed-length upshifted ASCII	ARVALUE-ASCII-F-U
64	Variable-length ASCII character	ARVALUE-ASCII-V
65	Variable-length upshifted ASCII	ARVALUE-ASCII-V-U

Value	Field Type	Literal Name
130	Signed 16-bit integer	ARVALUE-SMALLINT-S
131	Unsigned 16-bit integer	ARVALUE-SMALLINT-U
132	Signed 32-bit integer	ARVALUE-INT-S
133	Unsigned 32-bit integer	ARVALUE-INT-U
134	Signed 64-bit integer	ARVALUE-LARGEINT-S
140	32-bit floating point number	ARVALUE-FLOAT32
141	64-bit floating point number	ARVALUE-FLOAT64
150	Unsigned decimal	ARVALUE-DECIMAL-U
152	Decimal, leading sign embedded	ARVALUE-DECIMAL-LSE
192	Datetime, date, time, timestamp	ARVALUE-DATETIME
195	Interval years	ARVALUE-INTRV-Y
196	Interval months	ARVALUE-INTRV-MN
197	Interval years to months	ARVALUE-INTRV-Y-MN
198	Interval days	ARVALUE-INTRV-D
199	Interval hours	ARVALUE-INTRV-H
200	Interval days to hours	ARVALUE-INTRV-D-H
201	Interval minutes	ARVALUE-INTRV-M
202	Interval hours to minutes	ARVALUE-INTRV-H-M
203	Interval days to minutes	ARVALUE-INTRV-D-M
204	Interval seconds	ARVALUE-INTRV-S
205	Interval minutes to seconds	ARVALUE-INTRV-M-S
206	Interval hours to seconds	ARVALUE-INTRV-H-S
207	Interval days to seconds	ARVALUE-INTRV-D-S
208	Interval fraction	ARVALUE-INTRV-F
209	Interval seconds to fraction	ARVALUE-INTRV-S-F
210	Interval minutes to fraction	ARVALUE-INTRV-M-F
211	Interval hours to fraction	ARVALUE-INTRV-H-F
212	Interval days to fraction	ARVALUE-INTRV-D-F

field-length output, optional

INT .EXT:ref:1

is the 0-byte length of the particular field. In the case of a variable-length character (VARCHAR) field, field-length is the maximum length of the field (the declared length plus the 2 bytes that make up the length word). For fields that can be set to a null value, this count includes the 2 bytes occupied by the null indicator.

is-key-field output, optional

INT .EXT:ref:1

is a Boolean value indicating whether or not the field is an element of the primary key (either user-defined or system-defined) of the data file. The value returned is nonzero for a key field and 0 for all other types of fields.

next-field output, optional

INT .EXT:ref:1

is the ordinal field number of the next field (in ascending field number order) that is present in the audit record. If no field follows the field specified by *field-number* in the audit record, then *next-field* is set to -1.

null-allowed output, optional

INT .EXT:ref:1

is a Boolean value indicating whether or not the null value is allowed for the field. The value returned is nonzero if the field can be set to a null value and 0 if it cannot be set to a null value. Note that this flag does not indicate whether or not the field is actually null.

flags output, optional

INT .EXT:ref:1

returns the flag information for the field.

collation-def output, optional

INT .EXT:ref:1

returns internal-form filename of the collation definition for the field, if any. Returns all blanks if the field is not a character-type field or if no collation is defined.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

- The file that was modified must be accessible.
- If a file that has the same name as the modified file, but different characteristics, exists on disk and is accessible, the results of ARGETFIELDINFO will probably be incorrect. The TMF audit-reading procedures cannot always detect this situation.
- If the specified field does not exist in the audit record, *return-code* is -17 (field not present). In this case, the value of *next-field* is valid despite the error indication.
- The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARGETMESSAGELINE

This procedure retrieves a line of text from the most recent error message for a particular cursor. The call indicates which line of the message to retrieve (the first line of text in each message is line number 1). If the indicated line does not exist, then *line-length* is set to 0. This convention allows you to retrieve the complete text of a message by making successive calls to ARGETMESSAGELINE starting with *line-to-retrieve* set to 1 and incrementing it by 1 each time through the loop until you detect a *line-length* of 0.

```
CALL ARGETMESSAGELINE ( return-code               ! o
                      , cursor-number             ! i
                      , line-to-retrieve          ! i
                      , buffer                    ! o
                      , line-length );            ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	Bit 13, when set, indicates that the cursor has no error information recorded.

cursor-number input

INT:value

is the number of the open cursor for which the message text is to be retrieved.

line-to-retrieve input

INT:value

is the number of the text line to be retrieved from the message.

buffer output

INT .EXT:ref:*

is a buffer in the application process in which the line of message text is stored by ARGETMESSAGELINE. The buffer must be at least 72 bytes long. Note that the buffer can be allocated from either the 64K data stack or an extended segment.

line-length output

INT .EXT:ref:1

is a returned value indicating the byte length of the text line. The range is 0 through 72.

ARGETMXCOLUMNINFO

This procedure returns column information for the SQL/MX object indicated by the current data-fork audit record (as long as the object exists on disk and has the same name). The information provided by this procedure is necessary to understand the format of the before-image and after-image data returned by the ARFETCHMXBEFOREDATA[2] and ARFETCHMXAFTERDATA[2] procedures.

This procedure can only be called when the OBJECTTYPE field in the ARRECORD indicates ARVALUE-SQLMX-DATADML.

You should call this procedure once for every SQL/MX object, and then cache the information and re-use it. Whenever a FILE ALTER record, FILE PURGE record, FILE CREATE record, FILE RENAME record, or an ARVALUE-SQLMX-DDL object-type record is returned, any cached column information for the corresponding data fork should be deleted so ARGETMXCOLUMNINFO will be called again when the next audit record for the data fork is read. DDL changes to the object may cause the column offsets in the image buffer to change.

This procedure works with audit records of the following types: DELETE (3), INSERT (5), UPDATE (10), FILE ALTER (12), FILE CREATE (13), UPDATE FIELDCOMP (15), and FILE RENAME (16)

```
CALL ARGETMXCOLUMNINFO ( return-code                      ! o
                       , info-buffer                      ! i
                       , info-buffer-length               ! i
                       , image-buffer-length-needed       ! o
                       , reply-hint                       ! o
                       , [sub-system ]                    ! o
                       , [ar-error ] );                   ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

info-buffer input

INT(32):value

is the address of the buffer into which the before-image and after-image description information is returned. The buffer must be as large as is specified by the *info-buffer-length* parameter. The returned information describes the format of the information returned by the ARFETCHMX* procedures in their *image-buffer* parameters. The performance of this procedure will be improved if this buffer starts on a mod 4 address.

The format of the information returned in this buffer is described by the IMAGEINFO and COLUMNINFO structures defined below. All offsets are zero-relative from the start of the image buffer.

info-buffer-length input

INT(32):value

is the length in bytes of the caller-allocated info buffer. The performance of this procedure will be improved if this length is mod 4.

If the specified length is not large enough to contain the information for all the columns in the record, an ARE-BUFFER-TOO-SMALL error is returned and the required length is returned in *reply-hint*.

image-buffer-length-needed output

INT(32) .EXT:ref

indicates how large in bytes the image buffer used with ARFETCHMXBEFOREDATA[2] or ARFETCHMXAFTERDATA[2] must be for the SQL/MX object associated with the current audit record.

If this value is not mod 8, it will be rounded up to the next higher value evenly divisible by 8 before returning to the caller.

reply-hint output

INT(32) .EXT:ref

is a hint returned by TMFARLB2.

If *return-code* is ARE-BUFFER-TOO-SMALL, *reply-hint* returns the minimum length in bytes necessary for *info-buffer*.

Otherwise, this parameter contains zero.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

The definitions of IMAGEINFO and COLUMNINFO are shown in Example 5-1. They are declared in the ARDDL2 file (and the related AR* files). The values returned in all the length and offset fields in these structures are in bytes. The order of the columns reflects the order they were declared in, not their order in the on-disk record, which can differ from the declared order for SQL/MX objects.

Example 5-1. IMAGEINFO and COLUMNINFO Definitions

```
efinition IMAGEINFO.
02  COLUMN-COUNT              type binary 32,0
02  ENCODEDKEYLENGTH          type binary 32,0
02  filler                    type pic x (120)
02  COLUMNINFO-ARRAY          type COLUMNINFO occurs 1 to
MAXMXCOLUMNS times depending on COLUMN-COUNT.
end  ! of definition IMAGEINFO

ddefinition  COLUMNINFO.
02  COLUMN-NUM                  type  binary 32,0
              Column number (same as in SQL/MX metadata tables).
              Column number zero is the system key.
02  COLUMN-ANSI-TYPE            type  binary 32,0
              ANSI type code for the data type of the column.
              See table 18 in the ANSI SQL92 standard.
02  COLUMN-FS-TYPE              type  binary 32,0
              FS data type of the column.  See file sqlcli.h for the _SQLDT_*
              defines that describe the codes returned here
02  KEYFLAG                     type  binary 32,0
              Set to -1 if this column is not part of the clustering key, set to
              the key column position otherwise (first column is column #0).
02  DATAOFFSET                  type  binary 32,0     ! Offset into the Image Record.
              Offset in bytes of the data portion of this column. Note that the data
              portion does not include the null indicator or the varchar length.
02  DATALENGTH                  type  binary 32,0
              Length in bytes of the data portion of this column. The data portion
              will occupy DATALENGTH bytes, starting at offset DATAOFFSET.
02  NULLINDOFFSET               type  binary 32,0
              Offset of the NULL indicator of this column in bytes. This value
              is undefined if the column is not nullable.
02  NULLINDLENGTH               type  binary 32,0
              Length in bytes of this column's NULL indicator.  Value is 0 if the
              column is not nullable. Otherwise, the NULL indicator will occupy
              NULLINDLENGTH bytes, starting at offset NULLINDOFFSET.
              Note that the NULL indicator may or may not be adjacent
              to the data portion of the column. This value could be 0, 2, 4, or 8.
02  VARCHARLENOFFSET            type  binary 32,0
              Similar to the NULL indicator, this describes the offset of the
              length indicator for varchars (and maybe other variable-length
              columns in the future). This value is undefined if the column
              does not have a variable length indicator.
02  VARCHARLENLENGTH            type  binary 32,0     ! Zero if column not varchar.
              Size (length) in bytes of the variable length indicator. This value
              could be 0, 2, 4, or 8.  A value of 0 indicates that the column does
              not have a variable length indicator.
02  DATETIME-INTCODE            type  binary 32,0
              Detailed information on datetime or interval ANSI datatypes.
              See Tables 19 and 20 in the ANSI SQL92 standard or see the
              SQLDTCODE_ and SQLINTCODE_ constants defined in file sqlcli.h.
02  DATETIME-FSCODE             type  binary 32,0
              A more detailed description if COLUMN-FS-TYPE has the value 192
              (_SQLDT_DATETIME).  This value is sometimes also returned as
              "precision" of a datetime value. See the SQLDTCODE_* constants
              defined in file sqlcli.h for the values returned.  This value is
              undefined if COLUMN-FS-TYPE is not _SQLDT_DATETIME.
02  LEADING-PRECISION           type  binary 32,0
              This value is only set for interval data types and indicates the
              precision of the leading part of the interval (years, months, etc.).
```

Example 5-1. IMAGEINFO and COLUMNINFO Definitions

```
02  PRECISION                 type  binary 32,0
            The precision (number of significant decimal digits)
            for numeric values.
02  SCALE                     type  binary 32,0
            The scale (decimal digits after the decimal point) for numeric values.
            For interval, time and timestamp columns, this is the "fraction
            precision", the precision of fractional seconds (a value between 0
            for whole seconds and 6 for microsecond resolution).
02  CHARACTERSET              type  binary 32,0
            Character set for a character column. See
            SQLCHARSETCODE_ literals in file sqlcli.h.
02  FUTURECOLLATION           type  binary 32,0
            A literal identifying the collation of a character column,
            should collations be supported in Release 2.
02  filler                    type  pic x (20)
end    ! of definition COLUMNINFO
```

To ensure upward-compatibility, do not make assumptions about how information gets returned. For example, do not assume that:

- the offsets in the columns are in ascending order.
- the null indicator offset, varlen offset, and data offset are within a contiguous range without intervening fillers or data from other columns.
- the data in the image info buffer is “dense,” meaning that there are no unused spaces in it.
- the first field in the record starts at a fixed offset.
- the values of nullIndLength_ and varLenLength_ are fixed at 2 and 4, respectively, if non-zero.
- fields in the image buffer that are not described by an MXARLibColumnInfo structure are set to any given value.

ARGETNETWORKRECS

This procedure enables the returning of network-related audit records (NETWORK-COMMIT and NETWORK-ABORT) when the ARREAD procedure is called.

```
CALL ARGETNETWORKRECS    ( return-code )                ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

ARGETNONDATACHNGRECS

This procedure reverses the effect of the ARSTOPNONDATACHNGRECS procedure (it causes certain audit records that do not reflect changes to customer data to be returned instead of being discarded by the ARREAD procedure).

Audit reading must be started before you call this procedure.

```
CALL ARGETNONDATACHNGRECS  ( return-code );             ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

ARGETRECADDR

For format 1 and non-oversized format 2 entry-sequenced, relative, or unstructured files, this procedure computes and returns the 32-bit external (user-understood) record address of the data record whose modification is reflected in the current audit record.

Format 1 is the default file format and supports file sizes up to 2 GB - 1MB. Format 2 is a disk format that supports large format files (big files). An oversized format 2 file is one that has a maximum file size greater than 4 gigabytes - 4 kilobytes. If you use this procedure with an oversized format 2 file, the call fails with an error -21.

```
CALL ARGETRECADDR ( return-code                              ! o
                  , recaddr                                  ! o
                  , [ root-part ]                            ! i
                  , [sub-system ]                            ! o
                  , [ar-error ] );                           ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

recaddr output

INT(32) .EXT:ref:1

returns the record address of the modified data record. This is a record address, record number, or relative byte address, depending upon the particular file type (entry-sequenced, relative, or unstructured).

If the procedure call fails with an error -21, the record address returned is -3D.

root-part input, optional

INT .EXT:ref:4

is the name of the volume containing the primary (root) partition of the data file. If the modified file is a secondary partition of a partitioned Enscribe file, this information is necessary to determine the record address; in all other cases, this parameter is simply ignored.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

- You can use ARGETRECADDR after reading any type of data change audit record (insert, delete, or any form of update).
- The file that was modified must be accessible. If the file is a secondary partition of an Enscribe partitioned file, *root-part* must be present and the root partition of the file must also be accessible.
- If a file that has the same name as the modified file, but different characteristics, exists on disk and is accessible, the results of ARGETRECADDR will most likely be incorrect. The TMF audit-reading procedures cannot always detect this situation.
- The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARGETRECADDR64

For format 1, format 2, and oversized format 2 entry-sequenced, relative, or unstructured files, this procedure computes and returns the 64-bit external (user-understood) record address of the data record whose modification is reflected in the current audit record.

Format 1 is the default file format and supports file sizes up to 2 GB - 1MB. Format 2 is a disk format that supports large format files (big files). An oversized format 2 file is one that has a maximum file size greater than 4 gigabytes - 4 kilobytes.

```
CALL ARGETRECADDR64 ( return-code                          ! o
                    , recaddr64                            ! o
                    , [ root-part ]                        ! i
                    , [sub-system ]                        ! o
                    , [ar-error ] );                       ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

recaddr64 output

FIXED .EXT:ref:1

returns the 64-bit record address of the modified data record. This is a record address, record number, or relative byte address, depending upon the particular file type (entry-sequenced, relative, or unstructured). Note that you cannot use this value with the POSITION or KEYPOSITION Enscribe procedures; instead you must use it with the FILE_SETPOSITION_ or FILE_SETKEY_ procedure.

root-part input, optional

INT .EXT:ref:4

is the name of the volume containing the primary (root) partition of the data file. If the modified file is a secondary partition of a partitioned Enscribe file, this information is necessary to determine the record address; in all other cases, this parameter is simply ignored.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

- You can use ARGETRECADDR64 after reading any type of data change audit record (insert, delete, or any form of update).
- The file that was modified must be accessible. If the file is a secondary partition of an Enscribe partitioned file, *root-part* must be present and the root partition of the file must also be accessible.
- If a file that has the same name as the modified file, but different characteristics, exists on disk and is accessible, the results of ARGETRECADDR64 will most likely be incorrect. The TMF audit-reading procedures cannot always detect this situation.
- The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

AROPEN

This procedure opens a cursor. You can specify bounds on the sequence number of the audit files to which the cursor refers. The number of the cursor assigned to the audit trail is returned for use in subsequent calls to other audit-reading procedures.

```
CALL AROPEN ( return-code                                    ! o
            , generic-name                                   ! i
            , cursor-number                                  ! o
            , [ min-seqno ]                                  ! i
            , [ max-seqno ]                                  ! i
            , [ open-flags ]                                 ! i
            , [ trail-index-count ]                          ! i
            , [ trail-index-list ]                           ! i
            , [sub-system ]                                  ! o
            , [ar-error ] );                                 ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

generic-name input

INT .EXT:ref:9

is an array containing the name ($volume.subvol.two-char-prefix) of the audit-trail file (must be in local internal format) or the audit-trail identifier (MAT, MERGE, AUX01…AUX15) which identifies the audit trail to be read. If the audit-trail ID is MERGE, a merge cursor is opened and the master audit trail and auxiliary audit trails are merged into a single cursor (note that AuxPtr records are not returned to the application).

If the subvol name is “ZTMFAT” or *generic-name* is an audit-trail ID, TMF must be running and the TMP will locate and restore the necessary audit-trail files. If audit restore is allowed and is required, the TMP will start a TMFDR process to restore the audit file to a configured restore volume. When an EOF is returned, there is no need to close and reopen the cursor or call ARPOSITION. Simply delay a short time and call ARREAD again so that any additional audit written to the audit trail will be returned, even if a rollover to a new audit file has occurred. If TMF is not running, an error 82 will be returned when trying to obtain the current audit-file name or EOF.

If the subvol name is not “ZTMFAT” or an audit-trail ID, you must pre-restore the audit files to a chosen location and specify that location in this parameter.

cursor-number output

INT .EXT:ref:1

returns a number used to identify the cursor in subsequent calls.

min-seqno input, optional

INT(32):value

is a double-word value, within the range 1 through 999999, specifying the minimum audit file sequence number that this cursor will accept. By including this value, you can prevent the cursor from reading or being positioned into any audit file whose sequence number is less than this value. The default value is one.

max-seqno input, optional

INT(32):value

is a double-word value, within the range 1 through 999999, specifying the maximum audit file sequence number that this cursor will accept. By including this value, you can prevent the cursor from reading or being positioned into any audit file whose sequence number is greater than this value. The default value is 999999.

open-flags input, optional

INT:value

specifies certain attributes of the cursor, as described below. If *open-flags* is omitted, all fields default to 0.

Bit	Meaning When Set
0	Audit restore from audit dumps should be attempted for files that are not present on disk when an attempt is made to read from them. The requested audit dumps must be in the local TMF catalog and the TMP process must be running on the local system.
1	Audit restore from audit dumps should be attempted for the next audit file in the cursor direction if the file is not on disk when the current file is first read. The use of this option might improve performance for sequential reading through multiple audit files. This flag bit is ignored unless bit 0 is also set.
all others	Undefined.

trail-index-count input, optional

INT:value

is the number of entries in the *trail-index-list* that follows. If the cursor is not a MERGE cursor, the value must be zero or the parameter must be omitted. If specified for a MERGE cursor, the value must be in the range of 1 through 15.

trail-index-list input, optional

INT .EXT ref:*trail-index-count*

is an integer array of *trail-index-count* size. Valid only if *trail-index-count* is not zero. Each entry is the audit-trail index of an auxiliary audit trail to be read by this MERGE cursor. Each value must be 1 through 15, and may not be repeated. Each value must be that of a configured auxiliary audit trail.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARPOSITION

This procedure positions a cursor within the audit trail to which it refers and specifies the direction in which subsequent calls to ARREAD traverse the audit. Cursors are positioned by the sequence number of the desired audit-trail file and a relative byte address (offset) within the file.

```
CALL ARPOSITION ( return-code                          ! o
                , cursor-number                        ! i
                , audit-file-seqno                     ! i
                , audit-file-rba                       ! i
                , aux-index                            ! i
                , aux-seqno                            ! i
                , aux-rba                              ! i
                , [ cursor-direction ]                 ! i
                , [sub-system ]                        ! o
                , [ar-error ] );                       ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

cursor-number input

INT:value

is the number of the open cursor for which the message text is to be retrieved.

Note. The following five fields are used to position the specified cursor. These fields are the same as the values returned in the ARRECORD structure. By saving these fields from the ARRECORD, you can position the cursor and re-read the record. When reading forward, the record that begins at or after the position is returned. When reading in reverse, the record that ends at or before the position is returned.

---

audit-file-seqno input

INT(32):value

specifies the sequence number of an audit file in the audit trail referred to by the cursor. If this is a MERGE cursor, this number is the MAT sequence number.

`audit-file-rba` input

FIXED:value

specifies the relative byte address (offset) within the audit file to which the cursor is being positioned. If the value specified is zero, the cursor is positioned at the beginning of the audit file. If the value specified is -1, the cursor is positioned at the end of the audit file. If this is a MERGE cursor, then this is the MAT rba.

`aux-index` input

INT:value

This input parameter is used only for MERGE cursors. It specifies the auxiliary audit trail to position into. A zero value specifies the MAT audit trail.

`aux-seqno` input

INT(32):value

This input parameter is used only for MERGE cursors. It specifies the sequence number to position to within the specified *aux-index* audit trail.

`aux-rba` input

FIXED:value

This input parameter is used only for MERGE cursors. It specifies the rba to position to within the *aux-index* audit trail.

`cursor-direction` input, optional

INT:value

is a value indicating which direction the cursor will travel through the audit. The values are:

0 forward

> 0 reverse

If `cursor-direction` is omitted, forward is assumed.

When reading forward, ARREAD returns the next audit record that begins at or after the specified rba. If the rba is the beginning of an audit record, that record is returned; if the rba is not at the beginning of an audit record, the next audit record is returned.

AuditRecord Start rba :

100: Rec 1

200: Rec 2

300: Rec 3

ARPOSITION to 200, Read Forward returns Rec 2; Read Reverse returns Rec 1

ARPOSITION to 201, Read Forward returns Rec 3; Read Reverse returns Rec 2

ARPOSITION to 199, Read Forward returns Rec 2; Read Reverse return Rec 1

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARPOSITION2

This procedure uses the aux-trail-range returned by an ARFETCHAUXPOINTER call to position the cursor for the specified auxiliary audit trail. An AUXPOINTERINFO Def structure may also be specified for the MAT. When reading forward, ARREAD returns audit records starting with the LOWPOS and returns ARE-END-OF-AUDIT when the HIGHPOS is reached. When reading in reverse, ARREAD returns audit records starting with the HIGHPOS and returns ARE-END-OF-AUDIT when the LOWPOS is reached. The AUXPOINTERINFO Def structure simplifies merging the MAT and auxiliary audit trails.

```
CALL ARPOSITION2 ( return-code                              ! o
                 , cursor-number                            ! i
                 , position-info                            ! i
                 , [ cursor-direction ]                     ! i
                 , [sub-system ]                            ! o
                 , [ar-error ] );                           ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

cursor-number input

INT:value

is the number of the previously opened cursor that is to be positioned.

position-info input

INT .EXT:ref:1

is an AUXPOINTERINFO Def structure containing the LOWPOS and HIGHPOS audit-trail position information.

cursor-direction input, optional

INT:value

is a value indicating which direction the cursor will travel through the audit. The values are:

0	forward
> 0	reverse

If *cursor-direction* is omitted, forward is assumed.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARPRINTMESSAGE

This procedure writes, to a specified file, the message text associated with the most recent error condition for a particular cursor.

```
CALL ARPRINTMESSAGE ( return-code                        ! o
                    , cursor-number                      ! i
                    , file-number                        ! i
                    ,[ max-line-length ] );              ! i
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) Bit 13, when set, indicates that the cursor has no error information recorded.

cursor-number input

INT:value

is the number of the open cursor for which the message text is to be written.

file-number input

INT:value

is the filenumber of the file to which the message text is to be written.

max-line-length input, optional

INT:value

is the maximum number of characters to be printed as a single line of text. The default value is 72.

Considerations

- You can retrieve lines of the message text in a buffer by calling the ARGETMESSAGELINE procedure.

ARREAD

This procedure advances the cursor to the next audit record and copies the fixed-length fields and certain other attributes into the application's buffer. The next audit record is the subsequent record (in the set of audit records externalized by this interface) in the cursor direction specified by the most recent call to ARPOSITION for the specified cursor.

```
CALL ARREAD ( return-code                                    ! o
            , cursor-number                                  ! i
            , buffer                                         ! o
            , max-copy-length                                ! i
            , [sub-system ]                                  ! o
            , [ar-error ] );                                 ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) Bit 15, when set, indicates that the record returned in the buffer is longer than *max-copy-length*. At least one byte was truncated from the end.

cursor-number input

INT:value

is the number of the open cursor identifying the audit trail that is to be read.

buffer output

INT .EXT:ref:*

is a buffer in the application process in which the record returned by ARREAD is stored. Note that the buffer can be allocated either from the 64K data stack or from an extended segment.

max-copy-length input

INT:value

is the maximum number of bytes to copy into the buffer. ARREAD returns an error for values of *max-copy-length* less than 2.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARSETOPTIONS

This procedure sets several options that control audit-record processing. Rather than having multiple procedures to turn on and off the various options, this single procedure controls all of them. If a parameter is not specified, that option is not changed from its current setting.

```
CALL ARSETOPTIONS ( return-code                          ! o
                  , [ return-networkrecs ]               ! i
                  , [ return-nondatachangerecs ]         ! i
                  , [ max-ignorereccount ]               ! i
                  , [ set-undoauditflag ]                ! i
                  , [ return-TypeFlags ] );              ! i
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

return-networkrecs input

INT:value

controls the returning of Network Abort and Network Commit records. If TRUE, the records are returned. The default is FALSE; the records are not returned.

return-nondatachangerecs input

INT:value

controls the returning of audit records that do not change the data in the database, including audit generated by AudServ during certain operations WITH SHARED ACCESS. Some types of SQL/MX audit may also be ignored. If FALSE, the records are not returned. If TRUE, the records are returned. The default is TRUE; the records are returned.

max-ignorereccount input

INT(32):value

sets the maximum number of continuous audit records that can be ignored before returning an ARTYPE-CURRENTPOS record (no data is returned in the body area of the ARRECORD). Must be zero or a positive number. If zero, there is no limit on the number of audit records that may be ignored. The default is zero.

set-undoauditflag input

INT:value

controls the setting of UndoFlag in undo audit records. If this parameter is TRUE then the UndoFlag will differentiate between transaction-level and file-level undo audit. If it is transaction-level undo, the value of UndoFlag will be 1 and if it is file-level undo then the value will be 2. If the parameter is FALSE then UndoFlag will indicate that it's a transaction-level undo audit. If it is transaction-level undo, the value of UndoFlag will be 1 else it will be 0. The parameter default is FALSE.

return-TypeFlags input

INT:value

controls the return of transaction type flags from transaction state records. If the parameter is FALSE, type flags are not returned. If it is set to TRUE, ARREAD of transaction state record will return the type flags associated with the transaction. The parameter default is FALSE; type flags are not returned.

ARSTART

This procedure initializes the audit-reading programmatic interface. You must call ARSTART before you begin issuing calls to any other audit-reading procedure.

```
CALL ARSTART ( return-code                                  ! o
             , [ number-of-cursors ]                        ! i
             , [ operator-term ]                            ! i
             , [ extended-seg-size ]                        ! i
             , [ sqlmx-cache-size ]                         ! i
             ,   ar-version    );                           ! i
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

number-of-cursors input, optional

INT:value

is an integer in the range 1 through 30 indicating the maximum number of cursors that you can have open simultaneously. The amount of extended virtual memory allocated for each cursor is approximately 100 KB. The default value is one.

operator-term input, optional

INT .EXT:ref:12

is an array containing the name (in internal form) of a terminal to which messages to the operator are to be printed.

If *operator-term* is omitted, then the home terminal is used.

extended-seg-size input, optional

INT:value

specifies the maximum size (in megabytes) of the main extended segment. This memory space will be shared by all memory allocated except that related to audit for SQL/MX objects, including the memory reserved for use by cursors (approximately 100 KB each), as well as caching of labels for SQL/MP objects (approximately 32 KB each) and caching of label information for Enscribe objects (approximately 50 bytes each). The minimum is 2, the maximum is 128, and the default is 5.

sqlmx-cache-size input, optional

INT:value

specifies the maximum size (in megabytes) of the cache used by SQL/MX for caching labels of objects. The minimum is 2, the maximum is 128, and the default is 5. The cache is maintained in a SQL/MX specific extended segment that is different from the main extended segment. Note that this value does not represent the limit on the total memory used by SQL/MX.

ar-version input

INT:value

specifies which version of the ARRECORD structure your program understands. The set of literals begins as follows:

literal ARVALUE^CURRENT^VERSION = 1;

literal ARVALUE^SQLMX^VERSION = 1;

The next time ARRECORD is changed, a new structure will describe it, another fixed-version literal will be added to represent it with an assigned value of two, and ARVALUE^CURRENT^VERSION will be changed to two. The ARRECORD structure associated with ARVALUE^SQLMX^VERSION remains unchanged.

Typically you should not specify ARVALUE^CURRENT^VERSION for this parameter because the value of ARVALUE^CURRENT^VERSION changes automatically each time the ARRECORD structure is changed. Instead you should use the highest fixed-version literal (ARVALUE^SQLMX^VERSION initially). If you change your program to use a later version of ARRECORD, you then change this parameter to the fixed-version literal that applies to the new ARRECORD version. Using ARVALUE^CURRENT^ VERSION implies that you will change your program to use the new ARRECORD structure every time the structure is changed.

This parameter is required.

ARSTOP

This procedure does the appropriate cleanup and closes the programmatic interface.

```
CALL ARSTOP ( return-code                              ! o
            , [sub-system ]                            ! o
            , [ar-error ] );                           ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

sub-system output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case this value reports the subsystem that returned the error. Table 5-4 describes the subsystem codes.

ar-error output, optional

INT .EXT:ref:1

is a returned value valid only when `return-code` equals `ARE-INTERNAL-ERROR` (-1000), in which case the value indicates the specific error. Table 5-5 describes the error codes.

Considerations

The optional parameters *sub-system* and *ar-error* must be passed in pairs. You must pass either both the parameters or none.

ARSTOPNETWORKRECS

This procedure disables the returning of network-related audit records (NETWORK-COMMIT and NETWORK-ABORT) when the ARREAD procedure is called.

```
CALL ARSTOPNETWORKRECS    ( return-code )                  ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0) Table 5-3 describes the error codes.

Warnings (>0) None.

ARSTOPNONDATACHNGRECS

This procedure causes certain audit records that do not reflect changes to customer data to be discarded instead of being returned by the ARREAD procedure.

Audit records filtered out after this procedure is called include the following:

- Audit records generated during SQL online DDL modification operations (SQL NonStop Availability operations). SQL NSA operations are performed by the SQL AUDSERV program when the “WITH SHARED ACCESS” option is specified in an SQL ALTER TABLE or ALTER INDEX command.
- Audit records generated by the activity of the RDF updater on the RDF backup system.

Note that FLABMOD audit records are an exception; they are always returned, even when after this procedure is called.

Audit reading must be started before you call this procedure.

```
CALL ARSTOPNONDATACHNGRECS ( return-code );                ! o
```

return-code output

INT .EXT:ref:1

is a returned value indicating the outcome of this procedure.

Errors (<0)	Table 5-3 describes the error codes.
Warnings (>0)	None.

Error Codes

Table 5-3 lists all of the error codes returned by the TMF audit-reading procedures. Additional error codes will be defined in the future. You should design your error handling code so that it can be easily expanded to recognize and respond to error code values not documented in this table.

Table 5-3. Error Codes by Class (page 1 of 4)

Code	DDL Name	Meaning
Successful		
0	AR-OK	Successful completion.
User Errors		
-1	ARE-ALREADY-STARTED	You called ARSTART, but audit reading is already started.
-2	ARE-BAD-PARAM-COMBINATION	There is a conflict between two or more parameters that are valid when taken individually.
-3	ARE-CURSOR-NOT-ALLOCATED	The specified cursor was not allocated in the call to ARSTART.
-4	ARE-CURSOR-NOT-OPEN	The specified cursor is not open.
-5	ARE-CURSOR-NOT-POSITIONED	The specified cursor has not been positioned.
-6	ARE-CURSOR-SEQNO-RANGE	The sequence number passed to ARPOSITION is not within the range specified when the cursor was opened.
-7	ARE-INVALID-PARAM	The value of a parameter is outside the valid range.
-8	ARE-MISSING-PARAM	A required parameter is missing from the call.
-9	ARE-NO-CURRENT-RECORD	There is no current audit record from which to satisfy the request.
-10	ARE-NO-DISK-SPACE	ARSTART received an error 43 (no disk space) when attempting to allocate an extended segment.
-11	ARE-NO-FREE-CURSORS	All allocated cursors are in use.
-12	ARE-NOT-AVAILABLE	The object of the procedure call, such as a before-image or after-image, does not exist in the current audit record (you are using the wrong procedure call for the current type of audit record).

Table 5-3. Error Codes by Class (page 2 of 4)

Code	DDL Name	Meaning
Successful		
-13	ARE-NOT-STARTED	You tried to do something before starting the audit-reading interface (you must call ARSTART before calling any of the other audit-reading procedures).
-14	ARE-DATA-FILE-NOT-FOUND	The required data file is not available on disk.
-15	ARE-DATA-FILE-TYPE	The procedure you called cannot operate on the current type of data file.
-16	ARE-DATA-FILE-VERSION	The data file on disk is not the same type or does not have the same attributes as the one whose modification generated the current audit record.
-17	ARE-FIELD-NOT-PRESENT	The specified field does not exist in the current field-compressed update audit record (the field was not updated and is not part of the record key).
-18	ARE-NO-SQL	The attempted operation requires NonStop SQL/MP, but the system on which the operation was attempted does not include the NonStop SQL/MP product.
-19	ARE-FRAGMENT-NOT-PRESENT	The requested data record fragment does not exist in this audit record
-20	ARE-SWAP-VOL-INACCESSIBLE	When ARSTART attempted to allocate an extended segment, it received a file system error that indicated the swap volume for the segment was not accessible.
-21	ARE-OVERSIZE-FILE	You called ARGetRecAddr, but the data file whose modification generated the current audit record is an oversized FORMAT2 file. Use ARGetRecAddr64 instead.
-22	ARE-WRONG-AUDIT-VERSION	The Audit Trail record is in an older format.
-23	ARE-INVALID-OPERATION	The operation specified is not allowed on this file type.
SQL/MX Support Errors		
-100	ARE-NO-SQLMX-SUPPORT	SQL/MX is not installed.
-101	ARE-ONLY-USEABLE-WITH-SQLMX	The called procedure requires that the current audit record be an SQL/MX audit record.
-102	ARE-NOT-USEABLE-WITH-SQLMX	The called procedure requires a non-SQL/MX audit record.

Table 5-3. Error Codes by Class (page 3 of 4)

Code	DDL Name	Meaning
Successful		
-103	ARE-BUFFER-TOO-SMALL	The size of the output buffer is too small. The *reply-hint* variable contains the required size.
-104	ARE-MUST-USE-SEPARATE-BUFFERS	The request and reply maps must not overlap.
-105	ARE-ONLY-USEABLE-WITH-DATAFORK	The called procedure requires an SQL/MX data-fork record but the current record is not an SQL/MX data-fork record.
-106	ARE-ONLY-USABLE-WITH-DML	The called procedure requires a DML audit record but the current record is not a DML audit record.
-107	ARE-NEED-NEWER-SQLMX	The current version of SQL/MX is not new enough to understand the current audit record.
-108	ARE-NEED-OLDER-SQLMX	The current version of SQL/MX is too new to understand the current audit record.
File Management Error		
-700	ARE-WRITE-ERROR	An error occurred while attempting to write to the specified file. To obtain more information, call the FILEINFO system procedure.
Cursor Management Error		
-800	ARE-CURSOR-ERROR	An error occurred while attempting to open, close, or position a cursor or while switching between cursors. A more detailed error message is printed on the operator terminal and is available programmatically by calling the ARGETMESSAGELINE audit-reading procedure.
Auto I/O Errors		
-900	ARE-AUDIT-READ-ERROR	An error occurred while attempting to read an audit record. No further reads will be allowed for the current cursor until it has been repositioned. A more detailed error message is printed on the operator terminal and is available programmatically by calling the ARGETMESSAGELINE audit-reading procedure.

Table 5-3. Error Codes by Class (page 4 of 4)

Code	DDL Name	Meaning
Successful		
-901	ARE-END-OF-AUDIT	The cursor has reached the end of the audit file range specified when the cursor was opened. No further reads will be allowed for that cursor until it has been repositioned. A more detailed error message is printed on the operator terminal and is available programmatically by calling the ARGETMESSAGELINE audit-reading procedure.
-902	ARE-FILE-NOT-FOUND	The cursor is positioned to a sequence number that does not correspond to an audit file on disk, and audit restore failed or is not enabled. This error condition can result either from explicit cursor positioning (ARPOSITION) or successive reads. No further reads will be allowed for the current cursor until it has been repositioned.
-903	ARE-TMP-NOT-RUNNING	TMF is not running on the system. No TMF operation, including the ARLIB2 file, can be used. Start TMF.
-904	ARE-TMP-BAD-VERSION	The installed $TMP is downrev. The ARLIB2 file cannot be used with a downrev TMF. Install the correct version of TMF.
-905	ARE-NOT-LICENSED	The application file is not licensed using the ARLIB2 file or the TMFARUL2 file or both. The ARLIB2 file does not work. License the application file or the TMFARUL2 file or both.
Internal Errors		
-1000	ARE-INTERNAL-ERROR	An exceptional condition occurred within the system code. If the API that returned this error was called with two additional parameters, namely, sub-system and ar-error, see Table 5-4 for subsystem codes information and Table 5-5 for errors returned by the subsystem. The sub-system and ar-error information may be required for the Global Customer Support Center (GCSC) or your service provider for additional assistance.
-2000	ARE-MEMORY-ALLOCATION	An internal memory allocation occurred. Contact your service provider.

Table 5-4. Subsystem Codes

Code	DDL Name	Meaning
-3000	ARE-TMF	The subsystem is TMF. Table 5-5 describes the ar-error codes.
-3001	ARE-SQLMX	The subsystem is SQL/MX. For further information, see the *HP NonStop SQL/MX Messages Manual*.
-3002	ARE-FILESYSTEM	The subsystem is File Systems. For further information, see the *Guardian Procedure Errors and Messages Manual*.
-3003	ARE-DP2	The subsystem is DP2. For further information, see the *Guardian Procedure Errors and Messages Manual*.

Table 5-5. Errors returned in case return-code is -1000

Code	DDL Name	Meaning
-24	ARE-OVERSIZED-FORMAT1-FILE	The ARGetRecAddr procedure is called, but the data file whose modification generated the current audit record is an oversized FORMAT1 file. Use a Format1 file with a maximum capacity of 4GB or use a Format2 file.
-25	ARE-AUDIT-ERROR	The current audit record is not for a SQL/MX DataFork object or it does not have an before/after image to fetch or both. Check if the audit record being pointed at is correct.
-26	ARE-NIL-ADDRESS	The current audit record key address is nil which means it is a non key-sequenced file. The key value in this case is not stored in the audit record and must be derived using other information.

Table 5-5. Errors returned in case return-code is -1000

Code	DDL Name	Meaning
-27	ARE-PAGE-LIMIT-EXCEEDED	The ARGetRecAddr procedure is called. The total number of pages or blocks for all the partitions in the FORMAT1 file exceeds or equals 2^33. Use a Format 1 file which does not exceed the page/block limit or use a Format2 file.
-28	ARE-INVALID-FILETYPE	The ARGetRecAddr procedure is called. The data file whose modification generated the current audit record is not key-sequenced, entry-sequenced, unstructured, or relative. Check the file-type of the data file.
-29	ARE-CACHE-ERROR	Allocation and initialization of a new record address information cache entry could not be made.
-30	ARE-ILLEGAL-FLABMOD-RECTYPE	Illegal file label modification record type was encountered which does not match with the standard known file label modification operations. Ensure the file label modification operation you intend is among the standard known operations. It might be the case that the current audit record is not pointing to the intended location.
-31	ARE-ILLEGAL-RECTYPE	Illegal Audit Record type encountered. Ensure the current audit record is pointing to the intended record type.
-32	ARE-ILLEGAL-OTHER-RECCLASS	Illegal Record Class encountered which is not one of the following: Character field, Variable Length Character field, Numeric field. It is an assortment of other field types which is not expected.
-33	ARE-ILLEGAL-RECCLASS	Illegal Record Class encountered which cannot exist and thus is not expected.
-34	ARE-FIELDNUMBER-MISMATCH	Requested field number is not equal to previously saved field number.

Table 5-5. Errors returned in case return-code is -1000

Code	DDL Name	Meaning
-35	ARE-FIELD-NOT-FOUND	The requested field is not found in the audit record.
-36	ARE-INVALID-COLLATION-BASE	The information of the character field you requested from the field-compressed audit record has a non-standard collation.
-37	ARE-AUDITREAD-EXCEPTION	A non-standard TMF audit reading exception has occurred.
-38	ARE-GENERAL-AR-EXCEPTION	A general audit reading exception occurred which neither involves TMF nor Heap overflow.

Important. `These error codes are for debugging purpose and are to be used by support personnel only.`

How to Include Audit Reading in an Application

The TMF audit-reading procedures are contained in a set of files that are delivered with the TMF product. Table 5-6 summarizes the audit-reading procedure files.

Using FUP LICENSE and FUP PROGID to a Super group user will allow users other than SUPER.SUPER users to run the program based on Guardian file security.

There are two ways to associate the TMFARUL2 library file with the application run object file: the `ELD -libname` option, or the TACL RUN command `LIB` option. Once the association is made, the `LIB` option does not need to specified to execute the program. The RUN command `LIB` option can be used to change the location of the TMFARUL2 file.

Examples:

```
ELD linkfile .. TMFARLB2 .. -o runfile -l ZCLIPDLL .. other
required DLL's ..

   -libname $vol.subvol.TMFARUL2

FUP LICENSE (runfile, TMFARUL2)

TACL RUN /LIB TMFARUL2/ runfile
```

If `-libname` is part of the ELD step, the TACL LIB option is not required unless the TMFARUL2 file location changes.

A third party can use the ELD step to build the application file that is shipped to its customer, the `-libname` option is not required. Then the customer can do the FUP LICENSE and TACL RUN command with the LIB option to prepare the files for execution.

The FUP LICENSE and TACL RUN commands must be executed each time either file is replaced or moved.

Both the application runfile and TMFARUL2 must be LICENSEd or either a Process_Create_ error or ARLIB2 error -905 will occur. This will occur even if Super.Super runs the program.

Usage of the ZCLIDLL and ZCLIPDLL DLLs with the TMFARLB2 product

For the NonStop SQL/MX version released in the H06.10 RVU, the programs that call the TMFARLB2 procedures must use ZCLIPDLL.

Before this version of SQL/MX, some of the procedures required by TMFARLB2 were not exported by SQL/MX, thereby resulting in compile or link or run-time error when a user application tried to compile or link with TMFARLB2.

A workaround for this problem, on releases before the H06.10 RVU, is to use ZCLIDLL in place of ZCLIPDLL. The ZCLIDLL requires the CRE. This means that the EPTAL `main` must be renamed and a C `main` must be created to call the EPTAL `main`.

Example of linking with TMFARLB2 and ZCLIDLL:

```
ELD linkfile .. TMFARLB2 .. -o runfile -l ZCLIDLL .. other
required DLL's.. -libname $vol.subvol.TMFARUL2"
```

ZCLIPDLL released in the H06.10 RVU and in subsequent releases, includes the required exported procedures, and can be used instead of ZCLIDLL. This avoids the use of C main.

Example of linking with TMFARLB2 and ZCLIPDLL:

```
ELD linkfile .. TMFARLB2 .. -o runfile -l ZCLIPDLL .. other
required DLL's.. -libname $vol.subvol.TMFARUL2"
```

Table 5-6. Files Supplied With TMFARLB2 (T2781)

File Name	Content
ARDECS2	External declarations (in TAL) for the audit-reading procedures.
ARDECSC2	External declarations (in C) for the audit-reading procedures.
ARDDL2, ARC2, ARCOBOL2, ARTACL2, ARTAL2	DDL source and corresponding data-definition files for C, COBOL85, TACL, and TAL programs, respectively, that are going to call the audit-reading procedures. These files each contain the following sections: ● FILESPEC and ARRECORD define the record formats. ● ARTYPE-LITERALS contains the literals for record types. ● ARVALUE-LITERALS contains the literals for special values of certain fields and returned parameters. ● AR-RETURN-CODES contains the literals for the return codes.
TMFARLB2, TMFARUL2	Object files containing the ARLIB2 audit-reading procedures.

Use of AWAITIOX

AWAITIOX will not complete nowaited I/O calls started by TMFARLB2.

Index

A

ABORT audit record 5-21
Aborting transactions 1-4, 2-13/2-14
ABORTTRANSACTION code 3-11
ABORTTRANSACTION TMF procedure
    description of 2-3
    errors 2-4
    statements used for invoking 1-4
    syntax description 4-3
    unlocking records 1-15
    use and implications of 2-13
Access control block (ACB) 2-6
ACTIVATERECEIVETRANSID TMF procedure
    multithreaded heterogeneous processes 3-14
    restoring current transaction 2-13
    syntax description 4-5
After-image
    audit compression 5-7
    field description 5-28
Applications, including audit-reading procedures in 5-110
ARCLOSE audit-reading procedure 5-35, 5-65
ARCOMPLETEIO audit-reading procedure 5-36
ARFETCHAFTERIMAGE audit-reading procedure 5-37
ARFETCHAUXPOINTER audit-reading procedure 5-39
ARFETCHBEFOREIMAGE audit-reading procedure 5-40
ARFETCHCHILDNODELIST audit-reading procedure 5-41
ARFETCHFIELDVALUE audit-reading procedure 5-42
ARFETCHFRAGMENT audit-reading procedure 5-45
ARFETCHMXAFTERDATA audit-reading procedure 5-47
ARFETCHMXAFTERDATA2 audit-reading procedure 5-50
ARFETCHMXBEFOREDATA audit-reading procedure 5-53
ARFETCHMXBEFOREDATA2 audit-reading procedure 5-56
ARFETCHRECORDKEY audit-reading procedure 5-59
ARGETANSINAME audit-reading procedure 5-61
ARGETAUDRECHEADERINFO audit-reading procedure 5-65
ARGETFIELDINFO audit-reading procedure 5-66
ARGETMESSAGELINE audit-reading procedure 5-70
ARGETMXCOLUMNINFO audit-reading procedure 5-72
ARGETNETWORKRECS audit-reading procedure 5-77
ARGETNONDATACHANGERECS audit-reading procedure 5-78
ARGETRECADDR audit-reading procedure 5-79
ARGETRECADDR64 audit-reading procedure 5-82
AROPEN audit-reading procedure
    declaring a cursor 5-3
    syntax description 5-85
ARPOSITION audit-reading procedure
    positioning a cursor 5-4
    syntax description 5-88
ARPOSITION2 audit-reading procedure 5-91
ARPRINTMESSAGE audit-reading procedure 5-93
ARREAD audit-reading procedure
    retrieving information from audit records 5-5
    syntax description 5-94

ARSETOPTIONS audit-reading procedure 5-96
ARSTART audit-reading procedure 5-98
ARSTOP audit-reading procedure 5-100
ARSTOPNETWORKRECS audit-reading procedure 5-101
ARSTOPNONDATACHNGRECS audit-reading procedure 5-102
Audit compression 5-7
Audit dumps, restoring audit trail files from 5-4
Audit records
- description of fields 5-28/5-32
- formats 5-20
- retrieving information from 5-5
- types 5-19
- See also individual audit record names 5-21
- See also Subset audit record 5-15

Audit trail files
- See also Auxiliary audit trails 5-15
- cursors 5-3
- reading active 5-8
- restoring audit from audit dumps 5-4

Audit-reading procedures
- See also individual procedure names 1-1
- error codes 5-103
- including in an application 5-110

Audit-restore volume 5-4
AUX POINTER audit record 5-21
Aux pointer field description 5-29
Auxiliary audit trail files 5-15
AWAITIO system procedure 3-3

B

Backout anomalies 2-14
Before-image
- audit compression 5-7
- field description 5-29

BEGINTRANSACTION code 3-10
BEGINTRANSACTION TMF procedure
- statements used for invoking 1-3
- syntax description 4-7

BEGINTRANSACTION_EXT_ 4-10
BROWSE ACCESS, SQL/MP access options 1-14

C

CHECKMONITOR system procedure 2-8
CHECKPOINT system procedure
- description of 2-8
- placing CHECKPOINT calls 3-4/3-5

CHECKPOINTMANY system procedure 2-8
CHECKPOINTMANYX system procedure 2-8
Checkpoints
- placement of 2-7/2-10
- strategy in multithreaded requesters 3-3
- strategy in single-threaded requesters 2-6

CHECKPOINTTFILEENTRY code 3-10
CHECKPOINTX system procedure 2-8
Child node list field description 5-29
Child nodes
- distributed transactions 5-12
- relationship to parent node 5-13/5-14
- retrieving a list of 5-13

CLOSE system procedure
- closing a server 1-6
- closing the TFILE 3-2
- ENSCRIBE 1-15

Code
- ABORTTRANSACTION 3-11
- BEGINTRANSACTION 3-10
- CHECKPOINTTFILEENTRY 3-10
- ENDTRANSACTION 3-11
- ISSUEWRITEREADTOSERVER 3-10
- MAINLOOP 3-9

COMMIT audit record 5-22

Committing transactions 1-4
COMPUTETRANSID TMF procedure 4-12
Concurrency
  See Database concurrency 1-11
Condition codes (CCL, CCE, CCG) 4-5
Consistency
  See Database consistency 1-11
Context-sensitive servers 2-15
Current transaction
  description of 1-7
  manipulating 3-2
  obtaining the transid 3-13
  setting to nil 2-13
  setting to nil state 1-10, 3-13
Cursor
  closing 5-35
  declaration 5-3
  positioning 5-4
Cursor, closing 5-65

D

Database concurrency 1-11/1-19
Database consistency
  achieving maximum 1-11
  description of 1-11/1-19
  Enscribe 1-14/1-15
  NonStop SQL/MP 1-12/1-14
  tasks for maintaining 1-11
Datafile field description 5-30
DATETIME fields 5-18
DELETE audit record 5-22
Distributed transactions
  basic parent-child relationship 5-13/5-14
  description of 5-12
  layered offspring relationship 5-14
DSMSFILESPEC definition 5-21

E

ENDTRANSACTION code 3-11
ENDTRANSACTION TMF procedure
  Enscribe 1-15
  errors 2-3
  nowait calls 3-3
  statements used for invoking 1-4
  syntax description 4-15
  terminating transactions 2-3
Enscribe
  audit compression 5-7
  deleted record problem 1-17
  file locks
    ensuring level-1 database consistency 1-15
    exclusive 1-15
    locking modes 1-15
    performance issues 1-12
    using to ensure database consistency 1-14/1-19
  inserted record problem 1-16
  levels of consistency 1-18
Error codes
  description of 5-6
  returned by audit-reading procedures 5-103/5-106
Error messages, obtaining recent text 5-7
Errors
  See also error codes 5-103
  10 (file/record already exists) 1-15
  12 (file in use) 2-7, 3-2, 3-12
  73 (file/record is locked) 1-15
  75 (no transaction identifier) 1-8, 2-14
  76 (transaction ended) 2-10, 3-9
  79 (no file/record lock) 1-15
  81 (nowait I/O pending) 2-13
  ABORTTRANSACTION errors 2-4
  ENDTRANSACTION errors 2-3
  WRITEREAD errors 2-2

F

Field alignment, SQL/MP 5-16
Field descriptions in audit records 5-28/5-32
Field formats, internal 5-16
FILE ALTER audit record 5-23
FILE CREATE audit record 5-23
FILE PURGE audit record 5-24
FILE RENAME audit record 5-24
FILESPEC definition 5-20
FILE_OPEN_ system procedure
- opening the TFILE 2-7, 3-1
- opening $RECEIVE 2-12, 3-12

Fragments field description 5-30

G

Gateway processes 1-5
GETTMPNAME TMF procedure 4-17
GETTRANSACTIONDETAILS 4-19
GETTRANSID TMF procedure 4-24
GETTRANSINFO 4-26

H

Heterogeneous transactions 1-5
Home nodes 5-12
Homenode field description 5-30

I

Initiating transactions 1-3, 4-7
INSERT audit record 5-25
INTERPRETTRANSID TMF procedure 4-28
INTERVAL fields 5-18
ISSUEWRITEREADTOSERVER code 3-10

L

LASTRECEIVE system procedure 3-13
LOCKFILE system procedure 1-14
LOCKREC system procedure 1-14

M

MAINLOOP code 3-9
Message system 1-6
Multithreaded heterogeneous processes 3-14
Multithreaded operation 1-6
Multithreaded requesters
- checkpointing strategy 3-3
- description of 3-1
- differences from single-threaded requesters 3-1

Multithreaded servers
- description of 3-12
- differences from single-threaded servers 3-12

N

NETWORK ABORT audit record 5-25
NETWORK COMMIT audit record 5-26
NETWORK PREPARED audit record 5-26
Nil state 1-8
NOAUDITCOMPRESS file attribute 5-7
Nodes 5-12
NonStop servers 2-14, 3-13
NonStop SQL/MP
- audit compression 5-7
- DATETIME fields 5-18
- internal field alignment 5-16
- internal field formats 5-16
- INTERVAL fields 5-18
- levels of database consistency
  - BROWSE ACCESS 1-14
  - REPEATABLE ACCESS 1-12
  - STABLE ACCESS 1-13
- null fields 5-18
- variable-length character (VARCHAR) fields 5-17

Nowait ENDTRANSACTION calls 3-3
Nowait operations 4-15
Null fields 5-18

O

OPEN system procedure, opening the server process 1-6
Operations, nowait 4-15
Operations, waited 4-15

P

Parent nodes
- description of 5-12
- relationship to child nodes 5-13/5-14

Parentnode field description 5-30
Performance
- decreasing by using Enscribe file locks 1-12
- enhancing within a network environment 1-9

Procedure-call syntax
- See individual procedure names

Processes, heterogeneous multithreaded 3-14
Programming languages
- for coding requesters 1-6
- for coding servers 1-6

R

RBA field description 5-30
READLOCK system procedure 1-15
READUPDATE system procedure
- accepting incoming messages 2-12
- accepting work requests from requesters 1-6

READUPDATELOCK system procedure 1-15
Record key field description 5-30
Recoverable resource manager file 1-5
Rectype field description 5-31
Registering resource managers 1-5
Relative byte address (RBA) 5-3
Remote nodes 5-12
REPEATABLE ACCESS, SQL/MP access options 1-12
REPLY system procedure
- completing a READUPDATE 2-12
- implications of 2-14
- replying to requesters 3-13
- responding to the requester 1-6

Requesters
- programming languages for coding 1-6
- role in the requester/server model 1-7
- See also Multithreaded requesters
- See also Single-threaded requesters

Requester/server
- application design model 1-6/1-7
- communication 1-6
- multithreaded operation 1-6

Resource manager files 1-5
Resource manager registration 1-5
Resource managers 1-5
RESUMETRANSACTION TMF procedure
- backup process 3-8
- checkpoints 2-10
- multithreaded processes 3-14
- syntax description 4-31

Return codes 5-6

S

SCREEN COBOL, SEND verb 1-6
SEQNO field description 5-31
Servers
- context-sensitive 2-15
- excluding from a TMF transaction 1-8
- guarantees to 2-15
- minimizing processes 1-9
- NonStop 2-14, 3-13
- obtaining transaction identifier 1-10
- programming languages for coding 1-6
- role in the requester/server model 1-7
- subordinate 2-13
- See also Multithreaded servers 3-12

See also Single-threaded servers 2-11
SETMODE = 117 1-10
SETMODE = 4
description of 1-15
normal mode and reject mode options 1-17
Single-threaded requesters
checkpointing strategy 2-6
delegating work to servers 2-2
description of 2-1
system and TMF procedures for 2-1
Single-threaded servers
description of 2-11
file system procedures 2-11
SNOOP Dump/Restore (SNOOPDR) 5-4
SQL/MP
See NonStop SQL/MP
STABLE ACCESS, SQL/MP access options 1-13
Status descriptor 2-6
STATUSTRANSACTION TMF procedure 4-34
Subset audit records 5-15
System procedures
See individual procedure names
single-threaded servers 2-11

T

TEXTTOTRANSID TMF procedure 4-37
TFILE
access control block 2-6
clearing entry after unilateral abort 2-5
description of 2-6
multithreaded heterogeneous processes 3-14
opening 2-7, 3-1
Timestamp field description 5-31
TMF procedures
See individual procedure names
TMF SHUTDOWN audit record 5-26
TMF_BEGINTAG_FROM_TXHANDLE_ TMF procedure 4-40
TMF_GETTXHANDLE_ TMF procedure 4-43
TMF_GET_EXTTRANSID_ 4-44
TMF_GET_TX_ID_ TMF procedure 4-46
TMF_JOIN_EXT_ 4-50
TMF_JOIN_EXT_ TMF procedure 4-50
TMF_RESUME_ TMF procedure 4-53
TMF_SETTXHANDLE_ TMF procedure 4-55
TMF_SUSPEND_ TMF procedure 4-57, 4-60
TMF_SUSPEND_EXT_ 4-60
TMF_TXBEGIN_ TMF procedure 4-62
TMF_TXHANDLE_FROM_BEGINTAG_ TMF procedure 4-65
Transaction identifiers
BEGINTRANSACTION procedure 1-3
excluding servers from sharing 1-10
form 4-7
obtaining the current 3-13
Transaction Management Process (TMP) 4-17
Transaction pseudofile
See TFILE
Transactions
aborting 1-4, 2-3, 2-13
basic parent-child relationship 5-13, 5-14
committing 1-4, 2-3
considerations for defining 1-2/1-3
excluding servers 1-8
initiating 1-3, 4-7
invalid handle 4-40
layered offspring relationship 5-14
restarting aborted 3-8
status descriptor 2-6
terminating 2-3
unilateral aborts
clearing TFILE entry 2-5
description of 2-4

Transactions, distributed 5-12
Transactions, heterogeneous 1-5
Transactions, TMF 1-2
Transaction, current 1-7, 3-2
Transid field description 5-31
TRANSIDTOTEXT TMF procedure 4-68

U

Undoflag field description 5-32
Unilateral transaction aborts
  clearing TFILE entry 2-5
  description of 2-4
UNLOCKFILE system procedure 1-15
UNLOCKREC system procedure 1-15
UPDATE audit record 5-27
UPDATE AUDITCOMP audit record 5-27
UPDATE FIELDCOMP audit record 5-28

V

Variable-length character fields 5-17
Variable-length fields, reading 5-5
Volatile resource manager files 1-5
Volume field description 5-32
Volume Recovery, anomalies 2-14

W

Waited operations 4-15
WRITE system procedure 1-15
WRITEREAD system procedure
  errors 2-2
  sending a request to the server and receiving a reply 1-6
  subcontracting work to other servers 2-13

Special Characters

$RECEIVE
  multithreaded heterogeneous processes 3-14
  opening 3-12
  opening and closing 2-12
  requester/server model 1-6