This manual includes details of the new features, enhancements, and fixes for the FairCom products since the V7.12 commercial release.
Table of Contents

Documentation Overview
  Purpose of This Manual ............................................................. iii
  Audience ................................................................................ iii
  Structure .................................................................................. iii
  Syntax Diagram Conventions ...................................................... xiv

1 Highlights of this Release
  1.1 Introduction ........................................................................ 1-1
  1.2 Compatibility Notes ............................................................ 1-2
    1.2.1 A New Look ................................................................ 1-2
    1.2.2 New m-tree Includes c-treeDB C and C++ ....................... 1-3
    1.2.3 c-treeDB, ISAM and Low-Level Integration ..................... 1-4
    1.2.4 Upgrading the c-tree Server .......................................... 1-4
    1.2.5 Restoring Old Dump Files ............................................ 1-4
    1.2.6 HUGE File Support is Now Default for all (mtree) Models .... 1-4
    1.2.7 JDBC Driver Compatibility ........................................... 1-5
    1.2.8 Server System Support Guide ....................................... 1-5
    1.2.9 The Most Up to Date Information ................................. 1-5

2 Performance/Scalability Enhancements
  2.1 Performance Enhancement Overview ..................................... 2-1
  2.2 Transaction Processing Performance Enhancements ............... 2-1
    2.2.1 V7.12 to V8.14 Transaction Performance Comparison ........ 2-1
      2.2.1.1 Performance Test Specifications .............................. 2-1
      2.2.1.1.1 Configuration Settings Used for c-tree Server V7.12 .... 2-2
      2.2.1.1.2 Configuration Settings Used for c-tree Server V8.14 .... 2-2
      2.2.1.2 Performance Test Results ...................................... 2-2
    2.2.2 V8.14 Transaction Performance Enhancement Details .......... 2-3
      2.2.2.1 Transaction Group Commit (Commit Delay) ................. 2-3
      2.2.2.2 Commit Delay Operational Details ............................ 2-4
        2.2.2.2.1 Commit Delay Enhancements ................................ 2-6
        2.2.2.2.2 Enabling Transaction Commit Delay ..................... 2-6
      2.2.2.3 Reduced Flushing of Updated Data and Index Cache Pages . 2-7
        2.2.2.3.1 Using the TRANSACTION_FLUSH Server Keyword .... 2-7
        2.2.2.3.2 Using the CHECKPOINT_FLUSH Server Keyword .... 2-7
      2.2.2.4 Improved Log Flushing Strategy ................................ 2-8
      2.2.2.5 More Efficient Transaction Checkpoints ..................... 2-8
        2.2.2.5.1 Increasing the Interval Between Checkpoints ......... 2-9
      2.2.2.6 More Efficient Creation of Transaction Logs ............... 2-9
        2.2.2.6.1 Enabling Log Templates .................................... 2-9
        2.2.2.6.2 Limitations .................................................. 2-10
      2.2.2.7 More Efficient Flushing of Files .............................. 2-10
      2.2.2.8 Efficient Single Savepoint for Large Transactions .......... 2-10
      2.2.2.9 Deferred Flush of Transaction Begin ........................ 2-10
2.3 c-treeSQL Performance Enhancements ............................................. 2-11
  2.3.1 Support for Backward Index Scan ........................................... 2-11
  2.3.2 Optimized Index Use Due to Index Range Support ...................... 2-11
  2.3.3 COUNT(*) Optimization ...................................................... 2-12
  2.3.4 TOP Select Clause Support ............................................... 2-12
  2.3.5 Cardinality ................................................................. 2-13
  2.3.6 New Architecture of JDBC Driver and Server ........................... 2-13
  2.3.7 SQL Optimizer Configuration Keywords ................................ 2-13
2.4 Other Performance Enhancements ............................................... 2-14
  2.4.1 c-tree Server Uses Windows Critical Section Objects for Mutex Support. ................... 2-14
    2.4.1.1 Tip: Consider This Change for Your Multi-Threaded Windows Application 2-14
  2.4.2 c-tree Server Supports Data and Index Cache Sizes Up to 64 GB ............................................. 2-15
    2.4.2.1 Configuring Large Cache Sizes .................................... 2-15
    2.4.2.2 Limitations .......................................................... 2-15
  2.4.3 Apple Open Transport Protocol Performance Enhancements .......... 2-15

3 Performance Monitoring
  3.1 Performance Monitoring Overview ............................................ 3-1
  3.2 Performance Monitoring Using the ctstat Utility .......................... 3-1
    3.2.1 ctstat Utility Usage .................................................. 3-2
    3.2.2 ctstat Utility Examples ................................................ 3-2
      3.2.2.1 Admin-System Report Example .................................. 3-2
      3.2.2.2 Tivoli-System Report Example .................................. 3-3
      3.2.2.3 Admin-File Report Example ...................................... 3-4
      3.2.2.4 Tivoli-File Report Example ...................................... 3-4
      3.2.2.5 Admin-User Report Example ...................................... 3-5
      3.2.2.6 Function Timing Report Example .................................. 3-5
      3.2.2.7 Text Report Example .............................................. 3-6
  3.3 Performance Monitoring Using Server Keywords ........................... 3-6
    3.3.1 Automatically Logging Performance Snapshots ........................ 3-6
    3.3.2 Automatic Logging to the Server System Event Log ................ 3-6
    3.3.3 Automatic Logging to SNAPSHOT.FCS ................................ 3-7
  3.4 Performance Monitoring Using the SnapShot API ........................... 3-7
    3.4.1 SnapShot API Function Usage ........................................ 3-7
      3.4.1.1 Automatic Snapshot Support ..................................... 3-8
      3.4.1.2 Function Timing Support ......................................... 3-8
      3.4.1.3 System Snapshot Support ........................................ 3-9
      3.4.1.4 User Snapshot Support .......................................... 3-9
      3.4.1.5 File Snapshot Support .......................................... 3-10
      3.4.1.6 Combined Snapshot Support ..................................... 3-11
    3.4.2 Snapshot Contents ................................................... 3-11
      3.4.2.1 System Snapshot Structure ...................................... 3-11
      3.4.2.2 User Snapshot Structure ....................................... 3-15
      3.4.2.3 File Snapshot Structure ....................................... 3-16
      3.4.2.4 Function Timing .................................................. 3-18
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 c-treeSQL Server Support for Additional Platforms</td>
<td></td>
</tr>
<tr>
<td>5 Memory Files</td>
<td></td>
</tr>
<tr>
<td>5.1 Memory Files Overview</td>
<td>5-1</td>
</tr>
<tr>
<td>5.2 Using Memory Files</td>
<td>5-1</td>
</tr>
<tr>
<td>5.2.1 Creating Memory Files Using Server Configuration Keyword</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2.2 Creating Memory Files Programmatically</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2.3 Sharing Memory Files Created Programmatically</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2.4 Collecting Memory File Statistics</td>
<td>5-3</td>
</tr>
<tr>
<td>5.2.5 Memory File Limitations</td>
<td>5-4</td>
</tr>
<tr>
<td>5.2.6 Tip: Faster Server Shutdown with Memory Files</td>
<td>5-4</td>
</tr>
<tr>
<td>6 c-tree Plus for .NET</td>
<td></td>
</tr>
<tr>
<td>6.1 .NET Overview</td>
<td>6-1</td>
</tr>
<tr>
<td>6.2 c-tree .NET Architecture</td>
<td>6-2</td>
</tr>
<tr>
<td>6.3 c-tree .NET Quick Guide to Functions</td>
<td></td>
</tr>
<tr>
<td>6.3.1 Working with Sessions</td>
<td>6-3</td>
</tr>
<tr>
<td>6.3.2 Creating a Session Object</td>
<td>6-4</td>
</tr>
<tr>
<td>6.3.3 Creating a New Session Dictionary</td>
<td>6-5</td>
</tr>
<tr>
<td>6.3.4 Working with Databases</td>
<td>6-6</td>
</tr>
<tr>
<td>6.3.4.1 Creating a Database Object</td>
<td>6-6</td>
</tr>
<tr>
<td>6.3.5 Database Object with Transaction Control</td>
<td>6-7</td>
</tr>
<tr>
<td>6.3.6 Working with Tables</td>
<td>6-7</td>
</tr>
<tr>
<td>6.3.7 Creating a Table Object</td>
<td>6-8</td>
</tr>
<tr>
<td>6.3.8 Altering a Table</td>
<td>6-9</td>
</tr>
<tr>
<td>6.3.9 Working with Records</td>
<td>6-10</td>
</tr>
<tr>
<td>6.3.10 Creating a Record Object</td>
<td>6-10</td>
</tr>
<tr>
<td>6.3.11 Adding a Record</td>
<td>6-10</td>
</tr>
<tr>
<td>7 Index Ranges</td>
<td></td>
</tr>
<tr>
<td>7.1 Index Ranges Overview</td>
<td>7-1</td>
</tr>
<tr>
<td>7.2 Using Index Ranges from the c-tree ISAM API</td>
<td>7-2</td>
</tr>
<tr>
<td>7.2.1 Allocating an Index Range</td>
<td>7-2</td>
</tr>
<tr>
<td>7.2.2 Reading Records Using an Index Range</td>
<td>7-3</td>
</tr>
<tr>
<td>7.2.3 Freeing an Index Range</td>
<td>7-3</td>
</tr>
<tr>
<td>7.3 Using Index Ranges from the c-treeDB C API</td>
<td>7-4</td>
</tr>
<tr>
<td>7.3.1 Initiating an Index Range Operation</td>
<td>7-4</td>
</tr>
<tr>
<td>7.3.2 Reading Records With an Index Range In Effect</td>
<td>7-4</td>
</tr>
<tr>
<td>7.3.3 Terminating an Index Range Operation</td>
<td>7-5</td>
</tr>
<tr>
<td>7.3.4 Determining if a Range Operation is in Effect</td>
<td>7-5</td>
</tr>
<tr>
<td>7.3.5 c-treeDB Example Using the C API</td>
<td>7-5</td>
</tr>
<tr>
<td>7.4 Using Index Ranges from the c-treeDB C++ API</td>
<td>7-6</td>
</tr>
<tr>
<td>8 Blocking Record Reads</td>
<td></td>
</tr>
<tr>
<td>8.1 Blocking Record Reads Overview</td>
<td>8-1</td>
</tr>
<tr>
<td>8.2 Using Blocking Record Reads</td>
<td>8-1</td>
</tr>
<tr>
<td>9 Server-Based Queues</td>
<td></td>
</tr>
<tr>
<td>9.1 Server-Based Queues Overview</td>
<td>9-1</td>
</tr>
</tbody>
</table>
9.2 Using Server-Based Queues .................................................. 9-1
  9.2.1 Creating or Opening a System Queue .......................... 9-1
  9.2.2 Writing to a System Queue ......................................... 9-3
  9.2.3 Reading from a System Queue .................................. 9-5
  9.2.4 Retrieving the Size of the Next Message in a System Queue .. 9-6
  9.2.5 Retrieving the Number of Messages in a System Queue ......... 9-7
  9.2.6 Closing a System Queue ........................................... 9-8

10 File Notification
  10.1 File Notification Overview ........................................... 10-1
  10.2 Using File Notification ................................................ 10-1
    10.2.1 Enabling Notification for Actions on a File ................. 10-1
      10.2.1.1 Open Files for Which Notification is Requested ...... 10-2
      10.2.1.2 Open a System Queue ................................... 10-2
      10.2.1.3 Establish Notification for the File ................... 10-2
    10.2.2 Receiving Notifications for Actions on a File .......... 10-3
      10.2.2.1 Notification Queue Message Format ..................... 10-3
        10.2.2.1.1 Fixed Portion of Notification Queue Message .... 10-4
        10.2.2.1.2 Optional Key Values ................................ 10-5
        10.2.2.1.3 Optional Actor Node Name .......................... 10-6
        10.2.2.1.4 Optional Full Record Image ....................... 10-6
      10.2.2.2 Reading Notification Messages .......................... 10-6
    10.2.3 Disabling Notification for Actions on a File ............ 10-9
    10.2.4 Notification Callbacks ......................................... 10-10

11 New c-treeDB, ISAM and Low-Level Integration
  11.1 Overview ............................................................... 11-1
  11.2 Switching c-tree instances ....................................... 11-1
  11.3 Switching ISAM contexts .......................................... 11-2
  11.4 Obtaining table data and file number .......................... 11-3
    11.4.1 Obtaining data file number ................................ 11-3
    11.4.2 Obtaining index file number ................................ 11-4

12 Record Batch Enhancements
  12.1 Record Batches Overview .......................................... 12-1
  12.2 Using Record Batch Enhancements ................................ 12-1
    12.2.1 Batch Read in Physical Order ................................ 12-1
    12.2.2 Batch Read in Key Decreasing Order ....................... 12-2
    12.2.3 Batch Read with Index Range Definition ................... 12-2
    12.2.4 Batch Record Insertion ....................................... 12-2
      12.2.4.1 Additional Useful Batch Read Options .................. 12-3

13 Other Features by Product
  13.1 c-tree Server ....................................................... 13-1
    13.1.1 Extended Transaction Number Support ....................... 13-1
      13.1.1.1 Configurable Extended Transaction Number Options ..... 13-2
    13.1.2 Dynamic Advanced Encryption ................................ 13-2
      13.1.2.1 Enabling Advanced Encryption Support .................. 13-3
13.1.2.2 Encrypting Files Using Advanced Encryption ........................................... 13-3
13.1.3 Automatic Segmented File Support ................................................................. 13-3
13.1.4 Enhanced Precision for International Languages ............................................. 13-4
   13.1.4.1 Client/Server Support ................................................................. 13-4
   13.1.4.2 Standalone Support ............................................................................. 13-4
13.1.5 Server Logs Stack Trace in Case of Critical Error ........................................... 13-4
13.1.6 Create a List of Arbitrary Names in Server Configuration File ......................... 13-5
   13.1.6.1 Application Name List Usage ............................................................. 13-5
   13.1.6.2 Application Name List Restrictions ..................................................... 13-5
   13.1.6.3 Application Name List Example .......................................................... 13-6
13.1.7 ctFeatCHANNELS Now Default for All Servers .............................................. 13-6
13.2 c-treeSQL Server ................................................................................................. 13-7
   13.2.1 Support SQL User-Defined Scalar Functions ........................................... 13-7
      13.2.1.1 Creating a User-Defined Scalar Function ........................................ 13-8
      13.2.1.2 Invoking a User-Defined Scalar Function ........................................ 13-8
      13.2.1.3 Deleting a User-Defined Scalar Function ......................................... 13-9
      13.2.1.4 User-Defined Scalar Function Security ........................................... 13-9
13.2.2 c-treeSQL Conformance with Java 2 Platform ............................................... 13-9
13.2.3 Run-Time Settings for Java Stored Procedure Support .................................... 13-9
13.2.4 Case-Sensitivity Issues .................................................................................... 13-10
13.2.5 Set Environment Variable for Server Process .............................................. 13-10
13.3 c-treeDB ................................................................................................... 13-11
   13.3.1 Create c-treeDB Databases Accessible to c-treeSQL ................................... 13-11
      13.3.1.1 Limitations ....................................................................................... 13-11
   13.3.2 Detect c-tree Operational Model Used By c-treeDB ..................................... 13-11
   13.3.3 Test for Presence of $DELFLD$ Field in a Table ........................................ 13-12
   13.3.4 c-treeDB Support for c-tree Features ......................................................... 13-12
      13.3.4.1 File Mirroring .................................................................................... 13-12
         13.3.4.1.1 Creating Mirrored Files using the c-treeDB API ............................ 13-12
         13.3.4.1.2 Opening Mirrored Files using the c-treeDB API .......................... 13-13
         13.3.4.1.3 Adding Mirrored Files to a c-treeDB Database ......................... 13-13
         13.3.4.1.4 Suspending Mirroring ................................................................. 13-13
         13.3.4.1.5 Mirroring Limitations ................................................................. 13-13
      13.3.4.2 Keep Locks on Transaction Termination ................................................ 13-14
      13.3.4.3 c-treeDB C++ Support for Setting Key Type ...................................... 13-14
      13.3.4.4 Support Creating Multiple Indexes in One Physical Index File ................ 13-15
         13.3.4.4.1 New c-treeDB C API Functions .................................................. 13-15
         13.3.4.4.2 New c-treeDB C++ API Methods .............................................. 13-16
         13.3.4.4.3 Example: Creating Multiple Indices in One Physical Index File .... 13-16
   13.3.5 c-treeDB Support for c-tree API Functions .................................................. 13-17
      13.3.5.1 EstimateKeySpan .............................................................................. 13-17
         13.3.5.1.1 c-treeDB C API Example ............................................................ 13-17
      13.3.5.2 KeyAtPercentile ............................................................................... 13-18
      13.3.5.3 SetNodeName ................................................................................... 13-18
         13.3.5.3.1 c-treeDB C API Example ............................................................ 13-19
13.3.6 Remove Table Function ........................................ 13-19
13.3.7 User Defined Path Prefixes .................................... 13-19
13.4 dbExpress Driver .................................................. 13-20
13.5 c-treeVCL/CLX ..................................................... 13-20
   13.5.1 Additional Event Handlers for TCtTable ...................... 13-21
      13.5.1.1 When Record Deletion Fails .......................... 13-21
      13.5.1.2 When Record Edit Fails ................................ 13-21
      13.5.1.3 When Record Posting Fails ............................ 13-21
      13.5.1.4 When Inserting or Appending a New Record .......... 13-21
13.6 Support Setting Data and Index File Extensions ............... 13-21
13.7 Catching Exceptions Specific to c-treeVCL .................... 13-22
13.8 Additional c-treeVCL Enhancements ............................ 13-22
13.9 c-treeSQL ODBC Driver ........................................... 13-22
   13.9.1 c-treeSQL ODBC Driver for Unix .......................... 13-22
   13.9.2 Direct Link c-treeSQL ODBC Driver for Unix and Windows. 13-23
   13.9.3 Support Configurable c-treeSQL ODBC Driver Login Timeout 13-23
13.10 c-treeSQL JDBC Driver .......................................... 13-24
   13.10.1 New Architecture .......................................... 13-24
   13.10.2 JDBC Driver Bulk Fetch Support .......................... 13-24
   13.10.3 Connection Pooling Support ............................... 13-24
   13.10.4 JDBC 3.0 Specification Conformance ...................... 13-24
      13.10.4.1 ParameterMetaData Interface ......................... 13-24
      13.10.4.2 Modified Methods .................................... 13-25
13.11 c-tree Plus ...................................................... 13-25
   13.11.1 Support Next/Previous Operation After Record Add. .... 13-25

14 c-tree Utilities
14.1 c-tree Server Status Monitoring Utility, ctsysm ............... 14-1
   14.1.1 Using the ctsysm Utility ................................... 14-2
   14.1.2 ctsysm Configuration File ................................. 14-3
   14.1.3 ctsysm Configuration File Sample ......................... 14-4
14.2 Superfile Data Export Utility, ctsfex .......................... 14-5
14.3 Enhanced Superfile Compact Utility, ctscmp and ctscmpi ........ 14-6
14.4 Inspecting Index Contents and Checking Index Integrity .......... 14-7
   14.4.1 Examples .................................................. 14-7
14.5 Updating File IDs ................................................. 14-8
14.6 Moving Files without Stopping the c-tree Server ................ 14-9
14.7 c-treeSQL Maintenance Utility, ctsqlutl ........................ 14-9
14.8 Rebuild With Duplicate Record Purge in Client/Server Mode ........ 14-10
14.9 c-treeSQL Import Utility Modifications ......................... 14-10
   14.9.1 Support for Precision and Scale Information ............ 14-10
   14.9.2 Check for Invalid Characters in Field Names ........... 14-10
   14.9.3 Support for Null Fields and Field Default Values ....... 14-11
   14.9.4 Support for Importing Table with an Alternative Name ... 14-11
15 Other c-tree Highlights

15.1 c-tree Server SDK .......................................................... 15-1
15.2 Run-Time Data Filters ...................................................... 15-1
15.3 Partitioned Files ............................................................. 15-2
15.4 Advanced File Encryption ................................................. 15-2
15.5 Unicode Support ............................................................. 15-3

16 Critical, Serious, and Other Fixes

16.1 Critical Issues ............................................................... 16-1
  16.1.1 [c-tree] Local Disk Read I/O Problem ........................... 16-1
  16.1.2 [c-treeDB] ctcbWriteRecord Bug ................................. 16-2
  16.1.3 [Server] PREIMG Transaction Undo ............................... 16-2
  16.1.4 [Server] KDEL_ERR During Automatic Recovery ............... 16-2
  16.1.5 [Server] Duplicate Transaction Number ......................... 16-2
  16.1.6 [Server] Checkpoint Deadlock ...................................... 16-2
  16.1.7 [Server] Long Transaction in Superfile Causes Failure ...... 16-2
  16.1.8 [Server] New Index Node Failure .................................. 16-3
  16.1.9 [Server] Unexpected Client Termination While Aborting a Transaction .... 16-3
  16.1.10 [Server] V8 Server Abnormal Termination ..................... 16-3
  16.1.11 [Server] Server Hangs after ISAM Read Operation .......... 16-3
  16.1.12 [Server] c-tree Server Crash when Dump Stream File Open Fails. 16-3
  16.1.13 [Server] Automatic Recovery Error ............................. 16-4
  16.1.14 [SQL] SQL Crash Fixed .............................................. 16-4
  16.1.15 [SQL] c-treeSQL Server Memory Leak Source of Server Termination 16-4
  16.1.16 [SQL] Prevent Server Crash When SQL Clients are Active at Shutdown 16-4

16.2 Serious Issues .............................................................. 16-5
  16.2.2 [c-tree] Use AVLFILNUM() to Get a Valid File Number in ctcntidx() 16-5
  16.2.3 [c-tree] Multiple c-tree Instances in Each Native Thread ........ 16-5
  16.2.4 [c-tree] Rebuild Not Setting Index Corrupt Flag ............... 16-5
  16.2.5 [c-tree] Stop Server Start-Up with Incompatible Log Files ....... 16-6
  16.2.6 [c-tree] Invalid Approach to Detect CT_STRING Field Type Length 16-6
  16.2.7 [c-tree] UPDCIDX Interaction with PRMIIDX and ctDROPIDX ........ 16-6
  16.2.8 [c-tree] NXTREC/PRVREC Issue When Called After ADDREC ....... 16-6
  16.2.9 [c-tree] SETFLTR Memory Leaks Fixed ........................... 16-7
  16.2.10 [c-tree] c-tree Plus Filter Problem Comparing Strings .......... 16-7
  16.2.11 [c-tree] DLOK.ERR on UNIX in Multi-Threaded FPUTFGET Mode 16-7
  16.2.12 [c-tree] Conditional Expression Involving \( \) Not Properly Evaluated 16-7
  16.2.13 [c-tree] Duplicate Record Purge During Alter Table Rebuild .... 16-7
  16.2.14 [c-treeDB] c-treeDB Memory Violation .......................... 16-8
  16.2.15 [Server] Key Compression with Keys > 255 Bytes Cause Server Crash 16-8
  16.2.16 [Server] Abnormal Server Shutdown Due to Sharing Conflict ...... 16-8
  16.2.17 [Server] Windows 32 bit versions Service Background Issue .... 16-8
  16.2.18 [Server] 64-bit Bound Server Retrieves Incorrect IFIL ........ 16-9
16.2.19 [Server] Service Control Manager (SCM) Reports Untimely Response ........... 16-9
16.2.20 [Server] Windows 32 bit Versions c-tree Server Frees Invalid Handle ........... 16-9
16.2.21 [Server] Server Windows 32 bit versions Service Utility Hanging Error ........... 16-9
16.2.22 [Server] Bound Server Shutdown Problem .................................................. 16-10
16.2.23 [Server] Killing a Shared Memory Client .................................................... 16-10
16.2.24 [Server] Error 36 During Automatic Recovery ............................................. 16-10
16.2.25 [Server] Netware NLM Server Shutdown Issues ............................................ 16-10
16.2.26 [SQL] c-treeSQL Server Shuts Down if Unexpected Error Occurs .................. 16-10
16.2.27 [SQL] c-treeSQL Server Shutdown Issue ..................................................... 16-10
16.2.28 [SQL] SQL Client Could Open Files With Conflicting File Modes .................. 16-11
16.2.29 [JDBC] JDBC Exception if 20 or More Parameters in a Prepared Statement ... 16-11
16.2.30 [ISQL] Syntax Errors due to Formatting issues Fixed ................................... 16-11
16.2.31 [r-tree] Add Sector Size Command Line Option in r-tree to Open Superfiles 16-11
16.2.32 [r-tree] r-tree Bad Sort Key Logic .............................................................. 16-11
16.2.33 [r-tree] r-tree External DODA Fixes ............................................................ 16-11
16.2.34 [Utility] c-treeSQL Import Utility Cannot Import Files with Extensions Other Than .dat ................................................................. 16-12
16.2.35 [Utility] c-treeSQL Import Utility Avoids Data Padding Prompt if the Table is Empty .............................................................. 16-12
16.2.37 [Utility] Superfile Compact Utility Improvements ....................................... 16-12
16.2.38 [Utility] Last Member Number Cause Superfile Compact Utility Error ........... 16-12
16.2.40 [Utility] ctdump Always Prompts for the Script Name .................................. 16-13
16.2.41 [VCL] c-treeVCL BLOB Field Stack Overflow Fixed ..................................... 16-13
16.2.42 [VCL] c-treeVCL Memory Leak ................................................................. 16-13
16.3 Other Issues ..................................................................................................... 16-13
16.3.1 [c-tree] Obscure Infinite Loop in fnidkey() ..................................................... 16-13
16.3.2 [c-tree] Win32 Service Install Utility Eliminated Runtime Dependency ........... 16-13
16.3.3 [c-tree] File Creation Resource Errors ........................................................... 16-14
16.3.4 [c-tree] 64-bit Integer Multiply and Divide Errors ......................................... 16-14
16.3.5 [c-tree] Incorrect Number of Active Data Records in a Superfile .................... 16-14
16.3.6 [c-tree] WRTDEMO Cleanup ........................................................................... 16-14
16.3.7 [c-tree] SETNODE Adjustment - Client-Side Label ....................................... 16-14
16.3.8 [c-tree] STPUSRA and ctSNAPSHOT Unresolved Functions at Link Time .. 16-14
16.3.9 [c-tree] Header File Write Without Header Lock ............................................. 16-15
16.3.10 [c-tree] Client-Side Huge File frmkey Error ................................................. 16-15
16.3.11 [c-tree] Error Check for SERIALNUM During Record Update ...................... 16-15
16.3.12 [c-tree] CT_CHAR Schema Segment Mapped to Unsigned Integer ................. 16-15
16.3.13 [c-tree] Change the Bound Server Interpretation of the ctThrdInit numthreads Parameter .................................................................................................................. 16-15
16.3.14 [c-tree] Automatic Recovery Error INOT_ERR (101) .................................... 16-16
16.3.15 [c-tree] No PUTFIL Error When Turning Off TRANLOG Support .................. 16-16
16.3.16 [c-tree] PUTHDR to Suspend and Restore Serial Segment Handling ............. 16-16
16.3.17 [c-tree] Allow ctTRANSMODE with no ctTRNLOG Under TRANPROC 16-17
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.3.18</td>
<td>Mac: mbOpen() Not Returning Proper System Error for File Open</td>
</tr>
<tr>
<td>16.3.19</td>
<td>Mac: Improper Use of PBHOpenSync() and PBHCreateSync() Return Values</td>
</tr>
<tr>
<td>16.3.20</td>
<td>[c-treeDB] Duplicate Record Purge During Alter Table Rebuild</td>
</tr>
<tr>
<td>16.3.21</td>
<td>[c-treeDB] CTDB Could Not Iterate a Table in &quot;natural&quot; Order</td>
</tr>
<tr>
<td>16.3.22</td>
<td>[c-treeDB] CTDB Could Not Specify Read-Only Mode in the Same Manner as ISAM ctREADFIL</td>
</tr>
<tr>
<td>16.3.23</td>
<td>[c-treeDB] Alter Table Does Not Preserve ROWID Values During Rebuild</td>
</tr>
<tr>
<td>16.3.24</td>
<td>[c-treeDB] CTFiel Class Member Function Error 4016</td>
</tr>
<tr>
<td>16.3.25</td>
<td>[c-treeDB] CTDB Not Calling ctThrdInit() in Multi-Threaded Models</td>
</tr>
<tr>
<td>16.3.26</td>
<td>[c-treeDB] Wrong Field Data After ctdbCopyRecordBuffer</td>
</tr>
<tr>
<td>16.3.27</td>
<td>[c-treeDB] ctdbCreateTable API Enhanced</td>
</tr>
<tr>
<td>16.3.28</td>
<td>[c-treeDB] Scan Function Problem</td>
</tr>
<tr>
<td>16.3.29</td>
<td>[c-treeDB] Alter Table Function Index Member Deletion Error</td>
</tr>
<tr>
<td>16.3.30</td>
<td>[c-treeDB] Variable Length Record Error</td>
</tr>
<tr>
<td>16.3.31</td>
<td>[c-treeDB] Alter Table Issues</td>
</tr>
<tr>
<td>16.3.32</td>
<td>[c-treeDB] Defer Begin Transaction Added</td>
</tr>
<tr>
<td>16.3.33</td>
<td>[c-treeDB] ctdbAddDatabase Does not Check if Database Dictionary Exists</td>
</tr>
<tr>
<td>16.3.34</td>
<td>[c-treeDB] Conditional Index not Working</td>
</tr>
<tr>
<td>16.3.35</td>
<td>[c-treeDB] ctdbAddSegment Fail when Adding INTSEG and CT_BOOL</td>
</tr>
<tr>
<td>16.3.36</td>
<td>[c-treeDB] Segment Mode Change Failed to Trigger Index Rebuild</td>
</tr>
<tr>
<td>16.3.37</td>
<td>[c-treeDB] ctdbDatePack Fails to Pack Date 03/01/1700 Correctly</td>
</tr>
<tr>
<td>16.3.38</td>
<td>[c-treeDB] ctdbAlterTable Fails if Index File Name is Not Specified</td>
</tr>
<tr>
<td>16.3.39</td>
<td>[c-treeDB] Usage of Invalid Index Key Length for Modified Index</td>
</tr>
<tr>
<td>16.3.40</td>
<td>[c-treeDB] Rebuild Index Failure</td>
</tr>
<tr>
<td>16.3.41</td>
<td>[c-tree ODBC] Non-Null-Terminated Fixed-Length Strings in Variable-Length Part of Data Record</td>
</tr>
<tr>
<td>16.3.42</td>
<td>[Server] Set FILE_HANDLES Defaults to Maximum Number of Open Files the Server Supports</td>
</tr>
<tr>
<td>16.3.43</td>
<td>[Server] Skip Flush of Updated Cache Pages at End of Dump Checkpoint</td>
</tr>
<tr>
<td>16.3.44</td>
<td>[Server] ctUSE_RLIMIT Support for Linux File Handles</td>
</tr>
<tr>
<td>16.3.45</td>
<td>[Server] Cache Page Removed From List to Flush</td>
</tr>
<tr>
<td>16.3.46</td>
<td>[Server] Windows RCHK_ERR After Apparent Clean Shutdown</td>
</tr>
<tr>
<td>16.3.47</td>
<td>[Server] Advanced Encryption Master Password Over 9 Characters</td>
</tr>
<tr>
<td>16.3.48</td>
<td>[Server] Default Unix Server Compile Switches Changed</td>
</tr>
<tr>
<td>16.3.49</td>
<td>[Server] Add User Termination Check to Rebuild Loop With Memory File</td>
</tr>
<tr>
<td>16.3.50</td>
<td>[Server] LOKDMP Displays Incorrect File Disk I/O Figures</td>
</tr>
<tr>
<td>16.3.51</td>
<td>[Server] Restoring the Top-Level Exception Handler</td>
</tr>
<tr>
<td>16.3.52</td>
<td>[Server] Auto Transaction Commits Improved</td>
</tr>
<tr>
<td>16.3.53</td>
<td>[Server] Permit Time-outs to be Interrupted</td>
</tr>
<tr>
<td>16.3.54</td>
<td>[Server] Superfile Member Create Causes Server Core Dump</td>
</tr>
<tr>
<td>16.3.55</td>
<td>[Server] Application Server Crash During Shutdown with Active Client</td>
</tr>
<tr>
<td>16.3.56</td>
<td>[Server] User Name and Password Not Maintained by Application Server</td>
</tr>
<tr>
<td>16.3.57</td>
<td>[Server] cstat Utility Crashes When Using Function Option</td>
</tr>
<tr>
<td>16.3.58</td>
<td>[Server] Automatic Recovery Failure with LOPN_ERR</td>
</tr>
</tbody>
</table>
16.3.59 [SQL] HAVING Clause is fixed. .......................... 16-26
16.3.60 [SQL] SQL/ODBC Error when CT_DATE Fields were zero 16-26
16.3.62 [SQL] Hide Columns Within a Table in the SQL Import Utility 16-27
16.3.63 [SQL] Error 21041 When Selecting Rows with CT_TIMES Columns 16-27
16.3.64 [SQL] SQL Wrong Results When Selecting Inexistent rowed 16-27
16.3.65 [SQL] Implemented Record Locking in CTSQLIMP 16-27
16.3.66 [SQL ODBC] SQL Server Connection Error in Remote Machine. 16-28
16.3.67 [SQL ODBC] Better performance for ctGetIndexByUID 16-28
16.3.68 [SQL ESQIL] Prevent eSQL From Closing Standard Output on Disconnect 16-28
16.3.69 [SQL ESQIL] ESQL Limitation of 20 Columns 16-28
16.3.70 [r-tree] External DODA with Alignment Restrictions Cause Violation Error 16-28
16.3.71 [Utility] ctBTRV - Btrieve Utilities Updated 16-28
16.3.72 [Utility] Compact IFIL Utility Multi-Threaded Client Error 16-29
16.3.73 [VCL] Freezing Issue with TCtDatabase and TCtTable Components 16-29
16.3.74 [VCL] Internal Post Method Trapping Exceptions 16-29
16.3.75 [VCL] Modified Record Buffer Not Indicated 16-29
16.3.76 [VCL] TCtTable Component Properties Not Set at Runtime 16-29
16.3.77 [VCL] GotoBookmark Method Failed to Retrieve Bookmarked Record 16-29

17 New Function Descriptions
17.1 c-tree Plus Low-Level API Functions .......................... 17-1
17.1.1 ReplaceSavePoint Function .................................. 17-2
17.2 c-tree Plus ISAM API Functions ................................. 17-2
17.2.1 AllocateRange .............................................. 17-4
17.2.2 BlockingISAMRead ........................................... 17-6
17.2.3 CloseCtFileByName ........................................... 17-8
17.2.4 ctCallback ................................................. 17-9
17.2.5 ctNotify .................................................. 17-11
17.2.6 ctSysQueueClose ............................................. 17-13
17.2.7 ctSysQueueCount ............................................ 17-14
17.2.8 ctSysQueueLIFOWrite ........................................ 17-15
17.2.9 ctSysQueueMlen ............................................. 17-17
17.2.10 ctSysQueueOpen ............................................ 17-18
17.2.11 ctSysQueueRead ............................................ 17-19
17.2.12 ctSysQueueWrite ............................................ 17-20
17.2.13 FirstInRange ................................................ 17-21
17.2.14 FreeRange ................................................ 17-22
17.2.15 LastInRange ............................................... 17-23
17.2.16 NextInRange ............................................... 17-24
17.2.17 PreviousInRange ............................................ 17-25
17.3 c-treeDB C API Functions ....................................... 17-26
17.3.1 ctdbEstimateSpan ........................................... 17-28
17.3.2 ctdbGetDatno .............................................. 17-30
17.3.3 ctdbGetIdxno .............................................. 17-31
17.3.4 ctdbGetIdxnoByName ...................................... 17-32
17.3.5 ctdbGetIdxnoByNumber. ............................................. 17-33
17.3.6 ctdbGetPathPrefix. ................................................. 17-34
17.3.7 ctdbIsRecordRangeOn ............................................. 17-35
17.3.8 ctdbRecordAtPercentile ........................................... 17-36
17.3.9 ctdbRecordRangeOff .............................................. 17-37
17.3.10 ctdbRecordRangeOn ............................................. 17-38
17.3.11 ctdbRemoveTable .................................................. 17-40
17.3.12 ctdbSetCurrentNodeName ....................................... 17-41
17.3.13 ctdbSetPathPrefix .............................................. 17-42
17.3.14 ctdbSwitchContext ............................................. 17-43
17.3.15 ctdbSwitchInstance ............................................. 17-44
17.3.16 CTBase::SwitchInstance ....................................... 17-45
17.3.17 CTIndex::GetIdxno ............................................. 17-46
17.3.18 CTRecord::SwitchContext .................................... 17-47
17.3.19 CTSession::GetPathPrefix ................................... 17-48
17.3.20 CTSession::SetPathPrefix ................................... 17-49
17.3.21 CTTable::GetDatno ............................................. 17-50
17.3.22 CTTable::GetIdxno ............................................. 17-51
17.3.23 CTTable::Remove .............................................. 17-52
Documentation Overview

PURPOSE OF THIS MANUAL

This manual provides detailed descriptions of the new features, enhancements, and other fixes for the entire FairCom product line. You will discover that the new features and functionality covered in this update guide deliver significant new benefits to your application development efforts. This update addresses the following products:

- c-tree Plus Version 8.14
- c-tree Server Version 8.14
- c-treeSQL Server Version 8.14 (including c-treeSQL ODBC and JDBC Drivers)
- c-tree VCL/CLX components Version 8.14
- c-tree Plus ODBC Drivers

With this release of our SQL technology for Linux, HP-UX, Solaris, AIX, and Mac OS X environments, we are able to provide developers with a high performance cross-platform SQL interface. Furthermore, with this release FairCom is introducing significant new interfaces for .NET and dbExpress as well as great features such as memory files and enhanced performance. If there are any specific recommendations for this technology, please contact the FairCom office nearest to you — we’re here to be of service.

AUDIENCE

This manual is directed towards experienced c-tree Plus developers who are interested in learning about the new features available in this release of the software.

STRUCTURE

The manual contains the following chapters:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>Introduces some of the major features in this release.</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Contains information about the enhancements made for performance and scalability.</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Contains information about the new performance monitoring features.</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Contains information about the c-treeSQL Server support for Unix and Mac OS X.</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Contains information about the new memory file feature.</td>
</tr>
</tbody>
</table>
### SYNTAX DIAGRAM CONVENTIONS

Syntax diagrams appear in Courier type and use the following conventions:

<table>
<thead>
<tr>
<th>UPPERCASE</th>
<th>Uppercase type denotes reserved words. You must include reserved words in statements, but they can be upper or lower case.</th>
</tr>
</thead>
<tbody>
<tr>
<td>lowercase</td>
<td>Lowercase type denotes either user-supplied elements or names of other syntax diagrams. User-supplied elements include names of tables, host-language variables, expressions, and literals. Syntax diagrams can refer to each other by name. If a diagram is named, the name appears in lowercase type above and to the left of the diagram, followed by a double-colon (for example, privilege::). The name of that diagram appears in lowercase in diagrams that refer to it.</td>
</tr>
<tr>
<td>{}</td>
<td>Braces denote a choice among mandatory elements. They enclose a set of options, separated by vertical bars (</td>
</tr>
<tr>
<td>[]</td>
<td>Brackets denote an optional element or a choice among optional elements.</td>
</tr>
<tr>
<td></td>
<td>Vertical bars separate a set of options.</td>
</tr>
<tr>
<td>...</td>
<td>A horizontal ellipsis denotes that the preceding element can optionally be repeated any number of times.</td>
</tr>
<tr>
<td>( ), ;</td>
<td>Parentheses and other punctuation marks are required elements. Enter them as shown in syntax diagrams.</td>
</tr>
</tbody>
</table>
Chapter 1

Highlights of this Release

1.1 INTRODUCTION

FairCom is pleased to deliver the results of our continuing development efforts for our array of technologies. The new features, fixes, and functionality offered here will result in significant benefits to your applications.

These notes reflect notable improvements in many areas. Major enhancements include:

- **Performance/Scalability Enhancements:** This release includes significant improvements to transaction processing performance, c-treeSQL performance, and c-tree Server for Windows performance.

- **Performance Monitoring:** Significant amounts of internal operation and performance information can now be monitored with a new c-tree utility called ctstat. Use our command-line tool or call this function internally to gain valuable insights to assist in tuning the c-tree Server or your application. Server SnapShots may be instigated, providing system-level, user-level, and file-level statistics with an array of information such as: locks, threads, c-tree calls, memory usage, transactions and much much more…

- **c-treeSQL Server Support for Unix and Mac OS X:** c-treeSQL now supports a number of key Unix systems including Solaris, Linux, AIX, and HP-UX, as well as Mac OS X through v10.3.3 (Panther).

- **Memory Files:** Speed, Speed, SPEED! Need we say more? c-tree now allows you to create memory-resident data and index files that support the complete breadth of c-tree's data and index sophistication, while residing totally in memory. This much-requested feature is now a reality. You will love it!

- **Native .NET Interface:** c-tree Plus introduces support for yet another environment! Our new Microsoft® .NET interface gives application developers a simple interface into the proven core of c-tree Plus. c-tree Plus for the Microsoft .NET Framework is a component that integrates with VB .NET, C#, J#, Delphi and others. This powerful new component exposes the methods and properties of the c-tree Plus for .NET Class Level API and Function Level API.
- **Index Ranges**: c-tree gives you the ability to specify a range of values for one or more key segments. Each record retrieved must satisfy all the specified ranges. Support is provided for: equality; strictly greater than; greater than or equal; less than or equal; strictly less than; not equal; between inclusive; between inclusive lower range; between inclusive upper range; between exclusive; and not between, no overlap.

- **Blocking Record Reads**: c-tree provides the ability to block (with time-out) on ISAM record reads. As an example, assume you have read to end-of-file and you want to wait for the next inserted record. You can now stay blocked, waiting for the new record, and let c-tree "wake you up" when the new record arrives. This is one small example of the many possibilities available with this great new feature.

- **Server-Based Queues**: A new server enhancement allows client applications to allocate server-side queues, which provides a powerful centralized resource for inter-client communication. Clients can allocate shared queues; block for each other's messages; or communicate anything! There are many uses for this global resource.

- **File Notification**: This provides an ability to be alerted (notified) when a specific type of activity is performed on a file (i.e., tell me when there is an ADDREC on this file, etc.). This is an extraordinary new feature with unlimited potential for your applications!

- **c-treeDB, ISAM and Low-Level Integration**: New capabilities were added to the c-treeDB layer to give the developer immediate access to the ISAM and Low-level layers when needed. This gives c-tree developers more power than ever to use every available option for performance, while remaining in the c-treeDB layer. This will ease migrating older ISAM application for those who prefer the easy to use c-treeDB API. Power, flexibility and performance are what c-tree is all about!

- **Record Batches**: Enhancements to c-tree's ability to read blocks of records (batches) in a single function call result in improved performance. And now, the ability to ADD or INSERT multiple records (blocks) at one time allows you to write a massive number of records at once to disk… think speed!

### 1.2 COMPATIBILITY NOTES

Please review the following changes before using this version of c-tree Plus. A complete review of this entire document is also encouraged.

#### 1.2.1 A New Look

This release includes a major change to the c-tree directory structure. We have standardized our directory structure to include new tutorials, tools and installation programs. But relax, your trusty work zone where you will find the familiar mtree directory to build your projects, the c-tree source code directory, and all of the lib, obj, and bin directories you've been accustomed to. It is now located inside a new `ctreeSDK\ctreeAPI` directory.
Here are the directories you will find when you look inside your new FairCom installation:

- **ctreeDocs** - A repository for documentation.
- **ctreeSDK** - The c-tree core libraries and mtmake.
- **ctreeServers** - Server binaries.
- **ctreeTools** - Client and administrative utility binaries.
- **ctreeTutorials** - Example code demonstrating the many diverse development environments c-tree supports, right out of the box.

The screenshots below represent a before-and-after directory structure:

### c-tree Plus V7.12 Directory Structure

### c-tree Plus V8.14 Directory Structure

1.2.2 **New m-tree Includes c-treeDB C and C++**

FairCom’s m-tree make system has been modified so that the libraries created now include the c-treeDB C and C++ interfaces by default. Because of this modification, the default code size of the libraries has increased. You can exclude these API’s from your libraries by starting *mtmake* with the “-ctpp” option to exclude the c-treeDB C++ API or the “-ctdb” option to exclude both the c-treeDB C and C++ API’s.
1.2.3 c-treeDB, ISAM and Low-Level Integration

If you are creating a new application and have selected c-treeDB as the API of choice to access your data, there may be situations where you need to place calls directly into the ISAM or even low-level layers while remaining in your c-treeDB code. This may be to obtain certain specific services that are not be directly supported by c-treeDB, or you may want to rewrite certain c-treeDB functionality to better suit your specific requirements.

It may also be common to find situations were you have an existing application written using the ISAM or low-level API, however, you develop new modules using the c-treeDB API and will migrate the existing modules over time to c-treeDB. In either case you will need c-treeDB to support the mix of ISAM or low-level function calls with your c-treeDB code, specifically when you must work with a table's data and index files and record data.

New functionality has been added to more easily support using c-treeDB with multiple APIs. Complete details can be found in Chapter 11 "New c-treeDB, ISAM and Low-Level Integration".

1.2.4 Upgrading the c-tree Server

Because of changes to the transaction log format in the c-tree Server V8, we recommend that you perform the following steps prior to installing the new V8 server. Shut down the c-tree Server cleanly, restart the server so it can do automatic recovery with no clients attached, and then shut it down cleanly a second time. Performing two shutdowns in a row ensures the application files are up to date and there are no pending recovery items so all *FCS files except the FAIRCOM.FCS file can safely be removed. Be sure not to overlook the CTSYSCAT.FCS file (used for ODBC) and the SYSLOGDT.FCS and SYSLOGIX.FCS files. If you are using the SERVER_DIRECTORY, LOCAL_DIRECTORY, LOG_EVEN, LOG_ODD, START_EVEN, START_ODD or similar keywords that take a directory path, be sure to check that path for any *FCS files that should be removed or data and index files that should be cleaned.

1.2.5 Restoring Old Dump Files

If you have customers using the dynamic dump features of the c-tree Server, take note. The format of the dynamic dump files changed with version 8 of the c-tree Server. If you upgrade customers to use the c-tree Server V8, you may want to consider maintaining a copy of the older Dynamic Dump Recovery utility, ct rdmp, in case the user needs to restore from an older dump file at a later date.

1.2.6 HUGE File Support is Now Default for all (mtree) Models

FairCom now includes HUGE file support for all models. c-tree Huge Files are HUGE! 8 byte file addresses combined with segmented file support will allow files on the order of $10^{19}$ bytes. That's 18,000,000 terabytes!

You will no longer be prompted for this option when building your projects with mtree. Should you need to remove this support for backward compatibility, pass "--huge" as an additional
Highlights of this Release

1.2.7 JDBC Driver Compatibility

Our JDBC Driver now connects with both our c-treeSQL Server and our c-treeSQL Server Java Edition! Communicate with your existing c-treeSQL Servers, or use the flexible stored procedures and triggers available with our c-treeSQL Server Java Edition. One driver now connects to both!

1.2.8 Server System Support Guide

c-tree Servers are designed for superior performance and utmost stability. We have put together a comprehensive document to assist c-tree Server administrators to maintain their systems with maximum efficiency, in addition to always-on availability. You can download a copy of this great new manual from our Web site:


1.2.9 The Most Up to Date Information

Please visit the support section of our Web site for the most complete and up to date information regarding c-tree. You can find all of our complete manuals and errata at:

http://www.faircom.com/support/mans.shtml
Chapter 2

Performance/Scalability Enhancements

2.1 PERFORMANCE ENHANCEMENT OVERVIEW

The c-tree Server and c-treeSQL Server V8.14 are the result of the ongoing technical focus by FairCom developers on database performance. Using real-world performance tests and the new performance monitoring abilities of the c-tree Server, FairCom profiled the c-tree Server's performance at both the ISAM and SQL level and applied significant enhancements which benefit transaction and SQL query throughput and server scalability. This chapter details performance enhancements that are available in the c-tree Server V8.14.

2.2 TRANSACTION PROCESSING PERFORMANCE ENHANCEMENTS

The V8.14 release of the c-tree Server contains major enhancements to the c-tree Server's transaction processing subsystem, which can increase transaction throughput, decrease latency, and enhance system scalability.

This section presents a performance comparison between the c-tree Server V7.12 and the c-tree Server V8.14 for two identical transaction loads run on the same Windows 2000 dual-processor system. Following the performance comparison is a detailed explanation of the new transaction processing performance enhancements and how to enable them.

2.2.1 V7.12 to V8.14 Transaction Performance Comparison

This section presents an example showing performance comparisons between the c-tree Server V7.12 and V8.14 for an application with multiple clients adding, updating, and deleting records in a set of transaction-controlled files.

2.2.1.1 Performance Test Specifications

The performance test simulates an OLTP system in which transactions are presented to the system at a fixed rate. The clients operate on a set of twenty-three identical variable-length files. The load is distributed such that 60% of the time operations are performed on five of the files, and 40% of the time operations are performed on 18 of the files.
The unit of work for the test consists of:

1. Adding a 1024-byte variable-length record to a file.
2. Reading the record by key value.
3. Placing a write lock on the record.
4. Deleting the record.

The add and delete operations are implemented as two separate transactions. The key is a 90-byte unique key consisting of a four-byte iteration count, a 78-byte string, a four-byte thread ID, and a four-byte process ID.

The test involves four multi-threaded client processes, each having 40 threads. The system used for the test is a dual processor Xeon system running Windows 2000.

### 2.2.1.1.1 Configuration Settings Used for c-tree Server V7.12

The server configuration file used by the c-tree Server V7.12 for this performance comparison contains the following options:

```plaintext
; Cache settings
PAGE_SIZE 32768
DAT_MEMORY 50000000
IDX_MEMORY 50000000

; Transaction processing settings
LOG_SPACE 480
CHECKPOINT_INTERVAL 40000000
CHECKPOINT_FLUSH 19
TRANSACTION_FLUSH 10000
COMMIT_DELAY 10
COMPATIBILITY LOG_WRITETHRU
SUPPRESS_LOG_SYNC YES

; Windows-specific settings
PROCESS_PRIORITY NORMAL
```

### 2.2.1.1.2 Configuration Settings Used for c-tree Server V8.14

The server configuration file used by the c-tree Server V8.14 for this performance comparison contains the same options as the V7.12 server plus the following newly supported option:

```plaintext
LOG_TEMPLATE 2
```

### 2.2.1.2 Performance Test Results

The following table summarizes the test results:

<table>
<thead>
<tr>
<th></th>
<th>V7.12 Result</th>
<th>V8.14 Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. Transaction Time</td>
<td>528 milliseconds</td>
<td>106 milliseconds</td>
</tr>
</tbody>
</table>
These test results show the following significant performance improvements when using the c-tree Server V8.14:

1. **Transaction latency (average transaction time), decreased by 80%**, from 528 milliseconds to 106 milliseconds, when using the V8.14 c-tree Server. This is due to the transaction enhancements such as the commit delay feature, which allows grouping of log flushes by multiple clients during transaction commits into a single operation.

2. **Transaction throughput (average transaction rate), increased by 367%**, from 300 transactions/sec to 1400 transactions/sec. This is a result of the decreased transaction latency and better use of the two CPUs available on the system, as shown by the CPU usage figures.

3. **Total bytes written to disk during the test decreased by 37%**, from 273 MB to 171 MB. This is due to the more effective commit delay algorithm used by the c-tree Server V8.14, which reduces transaction log write activity.

### 2.2.2 V8.14 Transaction Performance Enhancement Details

This section describes the significant enhancements to transaction processing performance, which are available in the c-tree Server V8.14.

#### 2.2.2.1 Transaction Group Commit (Commit Delay)

The c-tree Server supports grouping transaction commit operations for multiple clients into a single write to the transaction logs. This feature is referred to as a group commit or commit delay. Transaction commit delay is a good choice for optimizing performance in an environment with large numbers of clients with a high transaction rate. This technique takes advantage of the overhead involved in flushing a transaction log. The performance improvement per individual thread may result in only milliseconds or even microseconds, but multiplied times hundreds of threads and thousands of transactions per second, this can become significant.

FairCom has supported the commit delay feature in previous versions, but during recent performance testing, FairCom identified a number of ways to enhance the effectiveness of the commit delay logic. Now, the performance advantages are even greater, and additionally, this...
opportunity expands the applications that were previously considered marginal candidates for performance improvement.

The remainder of this section covers the following topics:

- How transaction commit delay works
- The performance enhancements FairCom applied to transaction commit delay in V8.14
- How to configure transaction commit delay

### 2.2.2.2 Commit Delay Operational Details

Without commit delay, each thread performs its own transaction log flush during a transaction commit. When commit delay is enabled, rather than each thread directly flushing the transaction log during a commit, threads enter the commit delay logic which behaves as follows.

In previous versions, any thread executing in the commit delay logic is known as either the **blocker** or a **cohort**. The blocker is the thread that eventually performs the transaction log flush on behalf of all threads waiting in the commit delay logic. A thread becomes the blocker on entry to the commit delay logic if there is not already a thread designated as the blocker. The blocker acquires a synchronization object (**blocker**), which is used to coordinate the threads (**cohorts**) that subsequently enter the commit delay logic. The blocker sleeps for the commit delay period specified in the server configuration file, wakes up, flushes the transaction log, and clears the block.

While the blocker is sleeping, other threads may enter the commit delay logic during their own transaction commit operations. These threads are known as the **cohorts**. The cohorts wait for the blocker to clear the block. When the blocker clears the block, each cohort acquires and releases the block, exits the commit delay logic without flushing the transaction log (because the blocker has already done this), and completes its commit operation.
The above figure shows the effect of commit delay on the commit times for individual threads. The left side of the figure shows the situation when commit delay is disabled. The right side shows the situation when commit delay is enabled. This example shows six threads (labeled T1 through T6) with random arrival times in the transaction commit log flush logic. In this example, the thread T1 arrives first, followed by thread T2 and so on through thread T6.

When commit delay is disabled, each thread flushes the transaction log in turn. The shaded part of the rectangles represents the time spent by each thread flushing the transaction log. Thread T1 flushes first. T2 waits until T1's flush completes and then performs its flush, and so on.

When commit delay is enabled, the first thread entering the commit delay logic becomes the blocker (thread T1 in this example). Threads entering the commit delay logic after this point in time (threads T2 through T6) become cohorts. The blocker sleeps for the commit delay period and then flushes the transaction log. The cohorts sleep until the blocker has finished flushing the transaction log and has released the block, at which point they acquire and release the block and complete their commit without flushing the transaction log.
An exception to the "blocker - cohort" concept arises when the log buffer becomes full prior to the delay period. In this instance, the cohort will flush thus releasing the blocker. Statistics are captured to measure this occurrence and to assess how the transactions flow through the commit delay logic.

### 2.2.2.2.1 Commit Delay Enhancements

Two major enhancements were identified to increase the performance of the commit delay logic. The first modification eliminates the need to sleep the thread that is about to set the commit delay block (the blocker). The purpose of the short sleep was to ensure that all the commit delay cohorts from the previous block have had a chance to flow past the block. They were permitted to flow past the previous block because their log entries have been flushed to disk. They could not get past the block (hence the sleep) because the synchronization primitives on virtually all platforms do not provide a truly efficient way to release all blocking threads. Instead, each thread flowing past the block had to acquire (and then immediately release) a mutex type object.

By multiplexing the synchronization object that is used to create the block, the server can release the prior block on one channel, and then set the next block on a different channel. This keeps the blocker from re-acquiring the block before all the threads flow past from the previous block. The number of log flush wait channels to use is set at compile time, and the c-tree Server defaults to three (3) channels.

The second modification has added additional logic to the commit delay. The time spent by the blocking thread in a commit delay has been made more adaptive to the number of thread cohorts actually coordinating their log flushes. The initial blocker sleeps for the commit delay time specified in the server configuration file and checks the number of cohorts accumulated during the sleep period. If the blocker determines that relatively few cohorts accumulated during the sleep period, it reduces the commit delay sleep time to be used by the next blocker. If the blocker determines that many cohorts accumulated during the sleep period, it increases the commit delay sleep time to be used by the next blocker.

### 2.2.2.2 Enabling Transaction Commit Delay

Commit delay can be enabled using either of these server configuration keywords:

- `COMMIT_DELAY <milliseconds>`

  where `<milliseconds>` is the commit delay interval specified in milliseconds, or:

- `COMMIT_DELAY_USEC <microseconds>`

  where `<microseconds>` is the commit delay interval specified in microseconds (one millisecond is 1000 microseconds).

If both forms of the commit delay keyword are used, then the last entry in the configuration file prevails.
Note that not all systems support arbitrarily short sleep times. For example, FairCom has found that on Solaris, unless using real-time capabilities of the operating system, the minimum achievable sleep time is 10 milliseconds even if a shorter sleep time is requested.

### 2.2.2.3 Reduced Flushing of Updated Data and Index Cache Pages

The c-tree Server follows a buffer aging strategy, which ensures updated cache pages for transaction-controlled files are eventually flushed to disk. The factors that affect when a buffer is flushed include the number of times a buffer has been updated and the number of checkpoints that have occurred since the buffer was last flushed. This section describes ways to tune the c-tree Server's buffer aging strategy to avoid unnecessary flushing of updated buffers for transaction-controlled files.

#### 2.2.2.3.1 Using the TRANSACTION_FLUSH Server Keyword

The TRANSACTION_FLUSH server configuration keyword controls the aging of updated buffers based on the number of times a buffer has been updated since it was last flushed.

```
TRANSACTION_FLUSH <num_updates>
```

sets the maximum number of updates made to a data or index cache page before it is flushed. The default value is 100. Increasing this value reduces repeated flushing of updated cache pages that may occur in a system that maintains a high transaction rate with a pattern involving frequently updating the same buffers.

#### 2.2.2.3.2 Using the CHECKPOINT_FLUSH Server Keyword

The CHECKPOINT_FLUSH server configuration keyword controls the aging of updated buffers based on the number of checkpoints that have occurred since the buffer was last flushed.

```
CHECKPOINT_FLUSH <num_chkpts>
```

sets the maximum number of checkpoints to be written before a data or index cache page holding an image for a transaction controlled file is flushed. The default value is 2. Increasing this value avoids repeated flushing of updated cache pages that may occur in a system that maintains high transaction rates. When CHECKPOINT_FLUSH is increased, the c-tree Server automatically detects the reliance on previous transaction logs and increases the active log count as needed provided that the FIXED_LOG_SIZE server configuration keyword is not enabled.

The following formula estimates the number of logs required to support unwritten updated cache pages:

Let:

- \( CPF \) = CHECKPOINT_FLUSH value (defaults to 2)
- \( CPL \) = # of checkpoints per log (typically 3 and no less than 3)
- \( MNL \) = minimum # of logs to support old pages

Then:

\[
MNL = \left( \frac{(CPF + CPL - 1)}{CPL} \right) + 2, \text{ where integer division is used}
\]

**Note:** The c-tree Server does not use a formula. It detects the actual "recovery vulnerability" to determine what logs are required.
Example:
CPF=2, CPL=3 => MNL = 3 (but the server enforces a minimum of 4)
CPF=19, CPL=3 => MNL = 9 active transaction logs

2.2.2.4 Improved Log Flushing Strategy

The primary performance impact of transaction control is the result of flushing critical data to the transaction logs. Prior to V8.14, the c-tree Server flushed data to the transaction logs by issuing a write to the file system cache then calling a system function to flush the file system cache buffers to disk. FairCom found that the most efficient way to flush data to the transaction logs is to open the logs in a synchronous write mode, in which writes bypass the file system cache and go straight to disk.

The c-tree Server V8.14 supports the following server configuration keywords for opening the transaction logs in synchronous write mode:

- COMpatibility SYNC_Log is used on Unix systems to instruct the c-tree Server to open its transaction logs in synchronous write (direct I/O on Solaris) mode. In this mode, writes to the transaction logs go directly to disk (or disk cache), avoiding the file system cache, so the server is able to avoid the overhead of first writing to the file system cache and then flushing the file system cache buffers to disk. This keyword also causes flushed writes for data and index files to use direct I/O. Using this keyword enhances performance of transaction log writes.

- COMpatibility LOG_WRITETHRU is used on Windows systems to instruct the c-tree Server to open its transaction logs in synchronous write mode. In this mode, writes to the transaction logs go directly to disk (or disk cache), avoiding the file system cache, so the server is able to avoid the overhead of first writing to the file system cache and then flushing the file system.

2.2.2.5 More Efficient Transaction Checkpoints

A checkpoint is a snapshot of the c-tree Server's transaction state at an instance in time. The server writes checkpoints to the current transaction log at predetermined intervals. The most recent checkpoint entry placed in the c-tree Server's transaction logs identifies the starting point during the server's automatic recovery.

The interval at which checkpoints are written to the transaction log is determined by the server's CHECKPOINT_INTERVAL configuration keyword. This keyword specifies the number of bytes of data written to the transaction logs after which the server issues a checkpoint. It is ordinarily about one-third (1/3) the size of one of the active log files ( Lnmmnnnn.FCS). For high transaction rate systems, FairCom recommends increasing the checkpoint interval so that checkpoints occur less frequently.
2.2.2.5.1 Increasing the Interval Between Checkpoints

Follow these steps to increase the interval between checkpoints:

1. Because the server enforces a minimum of three checkpoints per transaction log, increase the size of the transaction logs using the LOG_SPACE keyword. Set LOG_SPACE to 12 times the desired checkpoint interval to accommodate three checkpoints per log for four active transaction logs.

2. Set the CHECKPOINT_INTERVAL setting to the desired checkpoint interval.

For example, to set the checkpoint interval to 20,000,000 bytes, use these server configuration settings:

```plaintext
LOG_SPACE 240
CHECKPOINT_INTERVAL 20000000
```

2.2.2.6 More Efficient Creation of Transaction Logs

Information concerning ongoing transactions is saved on a continual basis in a transaction log file. A chronological series of transaction log files is maintained during the operation of the c-tree Server. Transaction log files containing the actual transaction information are saved as standard files. They are given names in sequential order, starting with L0000001.FCS (which can be thought of as "active c-tree Server log, number 0000001") and incrementing sequentially (i.e., the next log file is L0000002.FCS, etc.). By default, the c-tree Server saves up to four active logs at a given time.

A transaction log must be 0xff filled to ensure known contents. Also, the logs are extended and flushed to ensure log space is available. Without the use of the log template feature, the 0xff filling of the log file (and forcing its directory entries to disk) occurs during log write operations. This means that the log file is tied up during this fill/extension processing, which can lead to increased latency for transactions that are in progress when log extension occurs.

With the transaction log template feature enabled, a new empty log template named L0000000.FCT is created at server startup to serve as a log template. The first actual full size log, L0000001.FCS, is copied from the template and a blank log, L0000002.FCT, is copied from the template. Whenever a new log is required, the corresponding blank log file is renamed from L000000X.FCT to L000000X.FCS and, asynchronously, the next blank log, L000000Y.FCT, is copied from the template.

This feature is designed only for use in very high speed, high throughput systems where it is desirable to be more than one log ahead in case the template copy operation runs slowly.

2.2.2.6.1 Enabling Log Templates

Enable the log template feature by specifying the server keyword LOG TEMPLATE in the server configuration file. The usage is:

```plaintext
LOG_TEMPLATE <n>
```
where \( n \) is the number of log templates you want the server to maintain. The default is 0, which means no use of log templates. For instance, a value of two (2) means that two blank logs (\( L0000002.FCT \) and \( L0000003.FCT \)) would be created at first server startup in addition to the template (\( L0000000.FCT \)).

Prior to using the log template feature, existing transaction logs must be deleted in order to cause the server to create log templates. To do this, shut down the server cleanly and delete \( Lnnnnnnn.FCS, S0000000.FCS, \) and \( S0000001.FCS \). When the server is restarted after adding this keyword, startup may take longer due to creation of template log files (*FCT).

### 2.2.2.6.2 Limitations

Log templates are not supported when mirrored logs or log encryption is in use.

### 2.2.2.7 More Efficient Flushing of Files

Similar to the strategy used in transaction log flushing, the c-tree Server can avoid excessive flushing of data and index files under transaction control. Two additional keywords affect this behavior.

COMPATIBILITY TDATA_WRITETHRU and COMPATIBILITY TINDEX_WRITETHRU force transaction controlled data files and index files, respectively, to be written directly to disk (whenever c-tree determines that they must be flushed from the c-tree buffers), and the calls to flush their OS buffers are skipped.

### 2.2.2.8 Efficient Single Savepoint for Large Transactions

The c-tree Server uses the `ReplaceSavePoint` function internally to provide a fast, efficient means to carry along a save point in large transactions, so that an error can be undone by calling `RestoreSavePoint` and then continuing the transaction. Compared to `SetSavePoint`, which inserts a separate save point for each call, `ReplaceSavePoint` simply updates some pre-image space links to effectively move the save point.

The `ReplaceSavePoint` is now included in the c-tree client API. If your c-tree client application needs the ability to undo the last change in a transaction in case of an error, consider using the `ReplaceSavePoint` function.

### 2.2.2.9 Deferred Flush of Transaction Begin

It is not uncommon for a higher-level application API to start transactions without knowledge of whether or not any updates will occur. To reduce the overhead of unnecessary log flushes, FairCom added a new mode, \texttt{ctDEFERBEG}, to the c-tree API function \texttt{Begin}, which is used to begin a transaction. \texttt{ctDEFERBEG} causes the actual transaction begin entry in the log to be delayed until an attempt is made to update a transaction-controlled file, and if a transaction commit or abort is called without any updates, then the transaction begin and end log entries are not flushed to disk.
FairCom applied this change after finding that c-treeSQL SELECT statements performed in autocommit mode involved transaction log activity due to the transaction begin and abort calls. The c-treeSQL engine now includes the ctDEFERBEG mode in transaction begin calls, eliminating transaction log I/O for transactions that do not involve updates. If your application begins transactions that might not involve updates, consider adding the ctDEFERBEG mode to your transaction begin calls.

### 2.3 C-TREESQL PERFORMANCE ENHANCEMENTS

In addition to all of the performance enhancements discussed in the previous sections, the c-treeSQL Server V8.14 includes numerous additional performance enhancements. This section discusses the specific enhancements and gives examples in which they benefit performance.

#### 2.3.1 Support for Backward Index Scan

Indices are useful not only to quickly retrieve information but also to sort records. The c-treeSQL Server V7.12 used the index to sort records only if the sorting direction requested for a SQL query (ascending or descending) matched the sorting direction of the index. For example, given an ascending index on a column, requesting rows in descending order required the c-treeSQL Server to scan the table in forward order, then sort the rows in order to return them in descending order. The c-treeSQL Server V8.14 supports scanning indexes backwards, so the server is able to efficiently perform ascending and descending order sort operations regardless of the ordering of the underlying index.

#### 2.3.2 Optimized Index Use Due to Index Range Support

The c-treeSQL Server V8.14 takes advantage of the new ISAM index range support to greatly improve the performance of SQL queries whose criteria specify ranges of key values or multiple columns from a single index. Prior to the introduction of the index range support, these types of queries required the c-treeSQL Server to retrieve records using only part of the criteria and then to filter the records using the remaining criteria.

For example, suppose that the Customer table has one index on the Name and City fields. Consider the following query:

```sql
SELECT * FROM Customer WHERE Name > 'M' AND Name < 'P'
```

Before the implementation of the index range support, executing this query required the index search logic to retrieve all the records matching either Name < 'M' or Name < 'P' and then to filter these two sets of records into a single set of records matching both criteria. Now the c-treeSQL Server searches the index using both restrictions, fetching only the records that satisfy both criteria.

Using the same table and index, consider the following query:

```sql
SELECT * FROM Customer WHERE Name < 'M' AND City = 'Dallas'
```
Previously, the c-treeSQL Server performed an index search on Name < 'M' followed by filtering the returned records on City = 'Dallas'. Now the index search logic uses both criteria to retrieve in one step the records that satisfy the criteria.

2.3.3 **COUNT(\*) Optimization**

When a SQL query specifies a COUNT(\*) field in order to count the number of records in a table, the c-treeSQL Server can now efficiently count the records by returning the record count stored in the header of the data file or the key count stored in an index associated with the data file. By comparison, in order to count all the records in a file, the c-treeSQL Server V7.12 read all records in the file. This change significantly improves performance of COUNT(\*) queries on large tables.

**Note:** This improvement does not affect COUNT(\*) with criteria specified (to count a subset of the records in the table). In this case, the c-treeSQL Server must retrieve the records that satisfy the specified criteria in order to count the records.

2.3.4 **TOP Select Clause Support**

FairCom added a new SELECT clause, TOP, to the c-treeSQL Server. With the addition of the new clause, the syntax of the select statement becomes:

```
SELECT [ALL | DISTINCT] [TOP n] <other select clauses>
```

The TOP n clause specifies that only the first n rows are to be output from the query result set. If the query includes an ORDER BY clause, the first n rows ordered by the ORDER BY clause are output. If the query has no ORDER BY clause, the order of the rows is arbitrary.

**Example:**

For example, if table USERS contains the following data:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>John</td>
</tr>
<tr>
<td>2</td>
<td>Mark</td>
</tr>
<tr>
<td>3</td>
<td>George</td>
</tr>
</tbody>
</table>

Executing the statement SELECT TOP 2* FROM USERS generates the following output:

```
ID   NAME
--   ----
1    John
2    Mark
```
When the ORDER BY clause is used, the statement SELECT TOP 2* FROM USERS ORDER BY generates the following output:

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>George</td>
</tr>
<tr>
<td>1</td>
<td>John</td>
</tr>
</tbody>
</table>

2.3.5 **Cardinality**

The c-treeSQL Server now has the ability to know in advance (without retrieving any records) the number of records in a table and the number of key values in an index. This information aids the optimizer in determining whether or not it is appropriate to use an index. There are cases when a table is small where it is more efficient to read the entire table and sort it in memory rather than using the index.

2.3.6 **New Architecture of JDBC Driver and Server**

With this release, the c-treeSQL architecture has been changed to improve the JDBC client and JDBC server performance. The new architecture bypasses the JNI layer on the server-side, thus resulting in a significant improvement in the performance of JDBC applications.

Because the new JDBC driver architecture no longer has a JNI layer in the server, there is no need to initialize the JVM in the server. This has been done without any additional overhead in the JDBC Driver. The new JDBC driver can interact with a c-treeSQL Server built in the Non-Java configuration, whereas previously the c-treeSQL Server Java Edition was required for JDBC support.

2.3.7 **SQL Optimizer Configuration Keywords**

The SQL engine optimizer plays an important role in reducing the execution time of SQL queries provided to the SQL engine. The role of the optimizer is to minimize the number of read requests that are made on the data and index files.

This minimization process is decided by the SQL optimizer based on the cardinality (the number of records selected), selectivity (the fraction of the records selected based on an operator), and a set of index key values. For many operations on records or indices, a cost, in terms of time units, is used to decide between the use of a record or index-oriented scan.

Users may wish to disable the cardinality, selectivity and/or costing features to force the operation of c-treeSQL to be similar to the first release of the server. Having these features enabled or disabled will induce the SQL optimizer to perform queries using different strategies to select the resulting dataset.

Three new c-treeSQL Server configuration file keywords were added to control the SQL engine optimizer's usage of cardinality, selectivity and costing of record and index operations: NO_CARDINALITY, NO_SELECTIVITY and NO_COSTS. These keywords must be used with the SQL_OPTION configuration command.
• **NO_CARDINALITY**: disables the cardinality logic that informs the SQL optimizer about the number of rows in a data or index file.

• **NO_SELECTIVITY**: disables the selectivity logic that informs the SQL optimizer about the percentage of rows returned when applying a given search criteria.

• **NO_COSTS**: disables the logic that returns to the SQL optimizer the cost in units of time for given record and index operations. The SQL optimizer will use this information to decide whether to use a record or an index scan.

**Note**: Costs are not implemented in this current version of c-tree. Behavior may change in future releases.

**Example:**

To disable cardinality, selectivity, and costing logic, add the following keywords to the c-tree Server configuration file:

```
SQL_OPTION NO_CARDINALITY
SQL_OPTION NO_SELECTIVITY
SQL_OPTION NO_COSTS
```

### 2.4 OTHER PERFORMANCE ENHANCEMENTS

This release also includes the following significant performance enhancements.

#### 2.4.1 c-tree Server Uses Windows Critical Section Objects for Mutex Support

The c-tree Server for Windows now uses critical section objects rather than mutex objects to implement internal server mutex support. FairCom introduced this change after profiling the c-tree Server showed significant overhead associated with the mutex calls used to control access to common server resources by c-tree Server threads.

FairCom's developers found that critical section objects can be used in place of mutex objects for synchronizing threads within a process. Critical section objects provide better performance than mutexes when there is no contention for the critical section and perform as well as mutexes when there is thread contention. This is because Windows critical section API function calls take advantage of the Interlocked API functions implemented entirely in user mode, unlike mutex API calls, which must go between user mode and kernel mode.

#### 2.4.1.1 Tip: Consider This Change for Your Multi-Threaded Windows Application

If your application runs on Windows and uses mutex objects to synchronize threads within a single process, consider using critical section objects rather than mutex objects for improved performance.
Details on the performance advantages of Windows critical section objects as compared to mutex objects can be found on the Web. As an example, the following Web page includes a discussion on this subject:


2.4.2 c-tree Server Supports Data and Index Cache Sizes Up to 64 GB

The c-tree Server now supports data and index cache sizes over 2 GB each. The ability to allocate large data and index caches improves performance for large data sets by allowing a larger portion of the data set to reside in server memory.

2.4.2.1 Configuring Large Cache Sizes

To configure large data and index cache sizes, add COMPATIBILITY LARGE_CACHE to the c-tree Server configuration file. This keyword permits the DAT_MEMORY and IDX_MEMORY values to be reinterpreted as megabytes instead of bytes.

If the byte value in the configuration file is less than or equal to 64000, then the value is reinterpreted as megabytes. This permits up to 64GB of index or data cache to be requested. If the value is greater than 64000, it is interpreted as bytes (just as without the LARGE_CACHE option). If the LARGE_CACHE option is not used, the values for DAT_MEMORY and IDX_MEMORY are interpreted as bytes, regardless of their values.

Example:

```
COMPATIBILITY LARGE_CACHE
IDX_MEMORY     100000000
DAT_MEMORY     4096
```

Requests 100 million bytes of index cache, and 4 GB of data cache.

2.4.2.2 Limitations

- The c-tree Server does not check that the specified amount of memory actually exists as available physical memory on the system. To avoid c-tree Server startup errors or performance degradation due to memory swapping, ensure that enough physical memory is available to accommodate the specified data and index cache sizes.

- The c-tree Server does not support this option on systems that do not support LARGE_CACHE. These systems are identified as 32-bit systems that do not have SYS_LONG8 defined in their ctcmpl.h include file.

2.4.3 Apple Open Transport Protocol Performance Enhancements

Open Transport is the Mac OS 8 and 9 API for accessing TCP/IP networks, such as the Internet, at the transport level. For Mac OS X, Apple provides Open Transport as a compatibility library to ease migration of legacy applications. For more information on the Open Transport protocol, see:
FairCom supports the Open Transport protocol for Carbon c-tree client applications on Mac OS 8, 9, and X. Native c-tree client applications on Mac OS 7-9 use MacTCP and native c-tree client applications on Mac OS X use the standard TCP/IP protocol.

FairCom identified performance bottlenecks in the use of Open Transport functions by the c-tree V7.12 client library and applied changes to resolve these. The result is a major boost in performance for c-tree clients that use the Open Transport protocol.

An example run on a FairCom test system highlights the dramatic performance improvement:

FairCom's ctvlqa example program linked with c-tree Plus V7.12 adds and reads 500 1024Kb records with full transaction control in 607 seconds. Linked with c-tree Plus V8.14, ctvlqa completes the same operation in 30 seconds. This new speed is comparable to the performance of the c-tree client using standard TCP/IP.
3.1 PERFORMANCE MONITORING OVERVIEW

The c-tree Server V8.14 tracks and reports a wealth of performance-oriented statistics. The server's Performance Snapshot capability enables the capture of performance monitoring data using a combination of configuration file options and the SnapShot c-tree Plus API function. The performance data can be captured automatically at specified intervals or on demand in the following ways:

- The *ctstat* utility provides a command-line interface to the SnapShot API function, supporting output of c-tree Server statistics at the c-tree Server system, user, file, and function levels.

- Configuration file options support automatic performance snapshots by specifying the interval between automatic snapshots and the contents of the snapshots. The performance data captured by automatic snapshots are written to the c-tree Server system event log (SYSLOG) files.

- Use DIAGNOSTICS options to capture the automatic system snapshot data to the human-readable *SNAPSHOT.FCS* file.

- The SnapShot API function can control automatic snapshots, overriding configuration options (if any), and can capture on-demand performance data:
  - to either the SYSLOG files or to the human-readable SNAPSHOT.FCS file, or
  - as a return to the calling program.

The following sections discuss how to use the performance monitoring abilities of the c-tree Server.

3.2 PERFORMANCE MONITORING USING THE CTSTAT UTILITY

The ctstat utility is a client utility used to display statistics collected by the c-tree Server. It is found in the client folder of the Server installation and demonstrates the use of the SnapShot function. This section describes the reports that this utility produces and how to use the utility to generate these reports.
3.2.1 ctstat Utility Usage

The command-line usage for the ctstat utility is as follows:

```shell
ctstat <report_type> [-u <userid>] [-p <password>] [-s <servername>]
    [-i <int> [<cnt>]] [-h <frq>] [-d]
```

where:

- `<report_type>` is one of:
  - `-vas` Admin-System Report
  - `-vts` Tivoli-System Report
  - `-vaf <file>...` Admin-File Report
  - `-vtf <file>...` Tivoli-File Report
  - `-vau <user>...` Admin-User Report
  - `-func` Function Timing Report
  - `-text` Write system snapshot to SNAPSHOT.FCS

- `-u <userid>` Connect to c-tree Server using user ID <userid>
- `-p <password>` Connect to c-tree Server using password <password>
- `-s <servername>` Connect to c-tree Server using server name <servername>
- `-i <int> [<cnt>]` Pause <int> seconds for optional <cnt> times
- `-h <frq>` Print a description header every <frq> outputs
- `-d` Show cache stats as delta

3.2.2 ctstat Utility Examples

This section provides examples of the types of reports that the ctstat utility can generate.

3.2.2.1 Admin-System Report Example

The admin-system report displays c-tree Server system-wide statistics in the areas of cache usage, disk I/O, open files, established client connections, file locks, and transactions.

Below is a sample admin-system report produced by executing the command:

```shell
ctstat -vas -u ADMIN -p ADMIN -s FAIRCOMS -h 10 -i 2
```

<table>
<thead>
<tr>
<th>cache</th>
<th>disk i/o</th>
<th>files</th>
<th>connect</th>
<th>locks</th>
<th>transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>d%h</td>
<td>%m</td>
<td>i%h</td>
<td>%m</td>
<td>cur</td>
<td>cur</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>99</td>
<td>1</td>
<td>1</td>
<td>281/10000</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>99</td>
<td>1</td>
<td>2</td>
<td>281/10000</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>99</td>
<td>1</td>
<td>2</td>
<td>281/10000</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>99</td>
<td>1</td>
<td>0</td>
<td>281/10000</td>
</tr>
</tbody>
</table>

The columns shown in this report are described as follows:

- `d%h` Data cache hit rate
- `%m` Data cache miss rate [100 - Data cache hit rate]
- `i%h` Index cache hit rate
- `%m` Index cache miss rate [100 - Index cache hit rate]
- `r/s` Disk reads per second
- `w/s` Disk writes per second
- `cur` Current number of open files
- `max` Server limit on number of open files
Performance Monitoring

cur   Current number of client connections
max   Server limit on number of client connections
cur   Number of locks currently held
l%h  Lock hit rate [(lock attempts - locks blocked - locks denied) / lock
      attempts]
%m   Lock miss rate [100 - Lock hit rate]
dead Number of dead locks detected
act   Current number of active transactions
t/s   Number of transactions per second
r/t   Number of read operations per transaction
w/t   Number of write operations per transaction

cur   Current number of client connections
max   Server limit on number of client connections
cur   Number of locks currently held
l%h  Lock hit rate [(lock attempts - locks blocked - locks denied) / lock
      attempts]
%m   Lock miss rate [100 - Lock hit rate]
dead Number of dead locks detected
act   Current number of active transactions
t/s   Number of transactions per second
r/t   Number of read operations per transaction
w/t   Number of write operations per transaction

3.2.2.2 Tivoli-System Report Example

The Tivoli-system report displays c-tree Server system-wide statistics in the areas of cache
usage, disk I/O, open files, established client connections, file locks, and transactions. The
Tivoli-system report displays much of the same statistics that the admin-system report dis-
plays, but in a format appropriate for input to tools such as the Tivoli monitoring application.

Below is a sample Tivoli-system report produced by executing the command:

cstat -vts -u ADMIN -p ADMIN -s FAIRCOMS -h 10 -i 2

#%cachehit %cachemiss r/s w/s maxfiles openfiles totalconnections activetrans-
actions numdeadlock trans-r/s trans-w/s %hashhit %hashmiss transactions/s
92 8 0 0 10000 18 1 0 0 0 0 100 0 0
92 8 0 9 10000 18 1 0 0 0 17 100 0 1
92 8 0 0 10000 18 1 0 0 0 100 0 1
92 8 0 0 10000 18 1 0 0 0 100 0 1
92 8 0 1 10000 18 1 0 0 0 1 100 0 1
92 8 0 0 10000 18 1 0 0 0 100 0 1

Note: The header line shown in this example is written as a single output line although it is
shown on two lines here.

The columns shown in this report are described as follows:

%cachehit Data and index cache hit rate
%cachemiss Data and index cache miss rate
r/s Disk reads per second
w/s Disk writes per second
maxfiles Server limit on number of open files
openfiles Current number of open files
totalconnections Current number of client connections
activetransactions Current number of active transactions
numdeadlock Number of dead locks detected
trans-r/s Number of read operations per transaction
trans-w/s Number of write operations per transaction
%hashhit Lock hit rate
%hashmiss Lock miss rate
transactions/s Number of transactions per second
3.2.2.3 Admin-File Report Example

The admin-file report displays c-tree Server statistics for the specified file. Note that multiple data or index files can be specified on the command line. Below is a sample admin-file report produced by executing the command:

```
ctstat -vaf mark.dat mark.idx -u ADMIN -p ADMIN -s FAIRCOMS -h 10
```

<table>
<thead>
<tr>
<th>r/s</th>
<th>w/s</th>
<th>entries</th>
<th>locks</th>
<th>l%h</th>
<th>%m</th>
<th>dlock</th>
<th>recrd</th>
<th>node</th>
<th>t</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>11863</td>
<td>4</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>n/a</td>
<td>F</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>11863</td>
<td>2</td>
<td>96</td>
<td>4</td>
<td>0</td>
<td>n/a</td>
<td>32768</td>
<td>I</td>
<td>mark.idx</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>12192</td>
<td>4</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>n/a</td>
<td>32768</td>
<td>F</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>9</td>
<td>12192</td>
<td>3</td>
<td>97</td>
<td>3</td>
<td>0</td>
<td>n/a</td>
<td>32768</td>
<td>I</td>
<td>mark.idx</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>12730</td>
<td>5</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>n/a</td>
<td>F</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>12730</td>
<td>1</td>
<td>97</td>
<td>3</td>
<td>0</td>
<td>n/a</td>
<td>32768</td>
<td>I</td>
<td>mark.idx</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>13236</td>
<td>5</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>128</td>
<td>n/a</td>
<td>F</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>13236</td>
<td>0</td>
<td>97</td>
<td>3</td>
<td>0</td>
<td>n/a</td>
<td>32768</td>
<td>I</td>
<td>mark.idx</td>
</tr>
</tbody>
</table>

The columns shown in this report are described as follows:

- **r/s**: Disk reads per second for the file
- **w/s**: Disk writes per second for the file
- **entries**: Number of data records or key values in file
- **locks**: Number of locks currently held on file
- **l%h**: Lock hit rate for the file
- **%m**: Lock miss rate for the file
- **dlock**: Number of dead locks detected for the file
- **recrd**: Record length if data file, otherwise n/a
- **node**: Node size if index, otherwise n/a
- **t**: File type (F=fixed-length data, V=variable-length data, I=index)
- **filename**: Name of the file

3.2.2.4 Tivoli-File Report Example

The Tivoli-file report displays c-tree Server statistics for the specified file in a format appropriate for input to tools such as the Tivoli monitoring application.

Below is a sample Tivoli-file report produced by executing the command:

```
ctstat -vtf mark.dat mark.idx -u ADMIN -p ADMIN -s FAIRCOMS -h 10
```

<table>
<thead>
<tr>
<th>#r/s</th>
<th>w/s</th>
<th>currentlocks</th>
<th>waitinglocks</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1 1034</td>
<td>mark.idx</td>
<td>mark.idx</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4 0</td>
<td>mark.dat</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>0 1120</td>
<td>mark.idx</td>
<td>mark.idx</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5 0</td>
<td>mark.dat</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 1208</td>
<td>mark.idx</td>
<td>mark.idx</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4 0</td>
<td>mark.dat</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2 1324</td>
<td>mark.idx</td>
<td>mark.idx</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5 0</td>
<td>mark.dat</td>
<td>mark.dat</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2 1402</td>
<td>mark.idx</td>
<td>mark.idx</td>
</tr>
</tbody>
</table>

The columns shown in this report are described as follows:
Performance Monitoring

r/s             Disk reads per second for the file
w/s             Disk writes per second for the file
currentlocks    Number of locks currently held on file
waitinglocks    Cumulative lock wait count
filename        Name of the file

3.2.2.5 Admin-User Report Example

The admin-user report displays c-tree Server statistics for the specified users.

Below is a sample admin-user report produced by executing the command:

cstat -vau GUEST -u ADMIN -p ADMIN -s FAIRCOMS -h 10

<table>
<thead>
<tr>
<th>log</th>
<th>function</th>
<th>sec</th>
<th>fil</th>
<th>lok</th>
<th>l%h</th>
<th>%m</th>
<th>dlock</th>
<th>tid/uid/nodename</th>
</tr>
</thead>
<tbody>
<tr>
<td>7s</td>
<td>TRANEND</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>98</td>
<td>2</td>
<td>0</td>
<td>GUEST/10/</td>
</tr>
<tr>
<td>7s</td>
<td>ADDREC</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>98</td>
<td>2</td>
<td>0</td>
<td>GUEST/12/</td>
</tr>
<tr>
<td>7s</td>
<td>ADDREC</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>98</td>
<td>2</td>
<td>0</td>
<td>GUEST/13/</td>
</tr>
<tr>
<td>7s</td>
<td>ADDREC</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>98</td>
<td>2</td>
<td>0</td>
<td>GUEST/14/</td>
</tr>
<tr>
<td>0s</td>
<td>ctSNAPSHOT</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>ADMIN/15/ctstat</td>
</tr>
<tr>
<td>7s</td>
<td>ADDREC</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>98</td>
<td>2</td>
<td>0</td>
<td>GUEST/17/</td>
</tr>
</tbody>
</table>

The columns shown in this report are described as follows:

log               Total logon time for client
function          Function client is currently executing
sec               Current function request time
fil               Current number of files open by this client
lok               Current number of locks held by this client
l%h               Lock hit rate for this client
%m                Lock miss rate for this client
dlock             Number of deadlocks detected for this client
tid/uid/nodename  Server thread ID/User ID/Node name for this client

3.2.2.6 Function Timing Report Example

The function timing report displays c-tree Server statistics for each c-tree function that a client has called at least once since the time the server started.

Below is a sample function timing report produced by executing the command:

cstat -func -u ADMIN -p ADMIN -s FAIRCOMS -h 10

<table>
<thead>
<tr>
<th>function</th>
<th>counter</th>
<th>secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRSVSET</td>
<td>10</td>
<td>0.002</td>
</tr>
<tr>
<td>NXTVSET</td>
<td>20</td>
<td>0.001</td>
</tr>
<tr>
<td>GETDODAX</td>
<td>2</td>
<td>0.000</td>
</tr>
<tr>
<td>COMMBUF</td>
<td>1</td>
<td>0.000</td>
</tr>
<tr>
<td>ctSNAPSHOT</td>
<td>10</td>
<td>0.002</td>
</tr>
</tbody>
</table>

The columns shown in this report are described as follows:

function     Function name
counter      Cumulative number of times this function has been called
secs         Cumulative elapsed time for this function
3.2.2.7 Text Report Example

This command writes a c-tree Server system snapshot to the file `SNAPSHOT.FCS`. See the file `SNAPSHOT.FCS` for the detailed server statistics.

Below is an example showing how to use `ctstat` to generate a text report:

```
ctstat -text -u ADMIN -p ADMIN -s FAIRCOMS -h 10
```

3.3 PERFORMANCE MONITORING USING SERVER KEYWORDS

This section describes automatic performance snapshot logging using c-tree Server configuration keywords.

3.3.1 Automatically Logging Performance Snapshots

The c-tree Server supports automatically logging performance snapshots to the SYSLOG files and to the human-readable file `SNAPSHOT.FCS`. The following sections describe how to enable automatic snapshots.

3.3.2 Automatic Logging to the Server System Event Log

The `SNAPSHOT_INTERVAL` keyword enables automatic snapshots at specified intervals:

```
SNAPSHOT_INTERVAL   <minutes>
```

By default, only the system snapshot is captured. To add user or file-specific snapshots to the data captured, use one or more of the following configuration entries:

```
SNAPSHOT_USERID     <user ID>
SNAPSHOT_FILENAME   <file name>
```

Files and users added to the snapshots are said to be activated. Users and files may be activated whether or not the automatic snapshots are turned on in the configuration file. However, the activation has no effect until snapshots are written to the SYSLOG files.

The `<user ID>` and `<file name>` arguments may include wildcard matching characters: "*" matches an arbitrary number of any characters, and "?" matches exactly one of any character. A pattern of simply "*" matches any user or file name. For example, the following keywords activate all users, any file ending in ".dat", and the file `journal.idx`:

```
SNAPSHOT_USERID     *
SNAPSHOT_FILENAME   *.dat
SNAPSHOT_FILENAME   journal.idx
```

User IDs are not case sensitive. File name case sensitivity depends on the platform. For example, Windows is case insensitive and Unix is case sensitive. The file names activated must match the file name used to first open the file. In particular, paths used in the activation list and during the call to open the file must match.
### 3.3.3 Automatic Logging to SNAPSHOT.FCS

Write system snapshots to the human-readable SNAPSHOT.FCS text file with the following DIAGNOSTICS options:

```plaintext
DIAGNOSTICS   SNAPSHOT_SHUTDOWN
DIAGNOSTICS   SNAPSHOT_AUTOMATIC
```

DIAGNOSTICS SNAPSHOT_SHUTDOWN writes a system snapshot to `SNAPSHOT.FCS` at the start of the server shutdown process. DIAGNOSTICS SNAPSHOT_AUTOMATIC writes any automatic snapshots to `SNAPSHOT.FCS` instead of to the SYSLOG files. However, only the system snapshot is written. Snapshots for activated users and/or files are ignored.

### 3.4 PERFORMANCE MONITORING USING THE SNAPSHOT API

c-tree Server configuration options provide an easy way to enable automatic performance snapshots. Additional flexibility and access to the complete set of statistics maintained by the c-tree Server is available programmatically through the SnapShot c-tree API function. This API function:

- Controls automatic snapshots, overriding configuration options (if any).
- Writes snapshots to the SYSLOG files on demand, whether or not automatic snapshots are active.
- Writes system snapshots in human-readable form to SNAPSHOT.FCS.
- Returns snapshot data on-demand to the calling application program.

#### 3.4.1 SnapShot API Function Usage

The snapshot function is prototyped as follows:

```plaintext
ctCONV NINT ctDECL SnapShot(NINT opcode, pTEXT rqstdesc, pVOID snapbufr,
VRLEN buflen);
```

SnapShot returns NO_ERROR(0) on success, or an error code on failure. `opcode` specifies the action to be taken by the snapshot function. `rqstdesc` points to an ASCII string containing the input information. `snapbufr` points to an output buffer of length buflen. Different opcode's require different combinations of input and output parameters ranging from using neither of them to both of them.

The remainder of this section describes how to use the SnapShot API function to collect system statistics by executing automatic snapshots, function timing, and system, user, and file snapshots.
3.4.1.1 Automatic Snapshot Support

The SnapShot API function supports starting, stopping, and resuming automatic snapshots. The following table shows the parameters to pass to SnapShot to perform these operations.

<table>
<thead>
<tr>
<th>Snapshot action</th>
<th>opcode</th>
<th>rqstdesc</th>
<th>snapbufr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start automatic snapshots, or change time interval if automatic snapshots are already enabled.</td>
<td>ctPSSstartAuto</td>
<td>Interval in minutes (e.g., &quot;60&quot;)</td>
<td>NULL</td>
</tr>
<tr>
<td>Stop automatic snapshots.</td>
<td>ctPSSstopAuto</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Resume automatic snapshots with the same interval last used.</td>
<td>ctPSSresumeAuto</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

3.4.1.2 Function Timing Support

The SnapShot API function supports profiling of c-tree API function calls (function timing) and timing of user-defined operations. The supported operations include:

- Starting and stopping the collection of c-tree API function timing statistics.
- Resetting the function timing statistics.
- Accumulating timings in user-defined timing baskets. Users may define up to 8 different timing baskets. A begin mark call establishes a high resolution starting time for the specified basket. An end mark call causes the specified basket counter to be incremented, and the elapsed time is added to the timing basket. The baskets are numbered from 0 to 7.

**Note:** Function timings require a significant number of calls to the high-resolution timer, and are more consistent with diagnostic or testing modes of operation.

The following table shows the parameters to pass to SnapShot to perform these operations.

<table>
<thead>
<tr>
<th>Snapshot action</th>
<th>opcode</th>
<th>rqstdesc</th>
<th>snapbufr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start collecting function-timing statistics. This can be called whether or not automatic snapshots are currently active. It affects the contents of snapshots written to SYSLOG, not when or if they occur.</td>
<td>ctPSStimeWorkOn</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Turn off function timings.</td>
<td>ctPSStimeWorkOff</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
3.4.1.3 System Snapshot Support

The SnapShot API function supports logging c-tree Server system statistics. For a listing of the system statistics the c-tree Server collects, refer to the ctGSMS structure in the Snapshot Contents section below. The supported operations include:

<table>
<thead>
<tr>
<th>Snapshot action</th>
<th>opcode</th>
<th>rqstdesc</th>
<th>snapbufr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark beginning time for one of 8 user specified timing baskets.</td>
<td>ctPSSbegMark</td>
<td>Small integer between 0 and 7 (e.g., &quot;3&quot;)</td>
<td>NULL</td>
</tr>
<tr>
<td>Mark ending time for user specified timing basket.</td>
<td>ctPSSendMark</td>
<td>Small integer between 0 and 7 (e.g., &quot;3&quot;)</td>
<td>NULL</td>
</tr>
<tr>
<td>Clear all function-timing statistics.</td>
<td>ctPSStimeWorkClear</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

3.4.1.4 User Snapshot Support

The SnapShot API function supports logging c-tree Server user statistics. For a listing of the user statistics the c-tree Server collects, refer to the ctGUMS structure in the Snapshot Contents section below. The supported operations include:

<table>
<thead>
<tr>
<th>Snapshot action</th>
<th>opcode</th>
<th>rqstdesc</th>
<th>snapbufr</th>
</tr>
</thead>
<tbody>
<tr>
<td>On demand system snapshot returned directly in the output buffer. No entry is made in SYS-LOG.</td>
<td>ctPSSsystem</td>
<td>NULL</td>
<td>Address of output buffer large enough to hold at least a ctGSMS structure</td>
</tr>
<tr>
<td>On demand system snapshot written to the SNAPSHOT.FCS text file.</td>
<td>ctPSStext</td>
<td>An optional text description, or NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>Activate the user with the specified User ID. Use a pattern of &quot;***&quot; to activate all users.</td>
<td>ctPSSaddUserID</td>
<td>User ID (e.g., &quot;admin&quot;)</td>
<td>NULL</td>
</tr>
<tr>
<td>Activate the user with the specified thread handle.</td>
<td>ctPSSaddUserHandle</td>
<td>Small integer thread handle (e.g., &quot;12&quot;)</td>
<td>NULL</td>
</tr>
</tbody>
</table>
3.4.1.5 File Snapshot Support

The SnapShot API function supports logging c-tree Server file statistics. For a listing of the file statistics the c-tree Server collects, refer to the ctGFMS structure in the Snapshot Contents section below. The supported operations include:

<table>
<thead>
<tr>
<th>Snapshot action</th>
<th>opcode</th>
<th>rqstdesc</th>
<th>snapbufr</th>
</tr>
</thead>
<tbody>
<tr>
<td>On demand user snapshot returned directly in the output buffer. No entry is made</td>
<td>ctPSSuser</td>
<td>NULL</td>
<td>Address of output buffer large enough to hold at least a ctGUMS structure</td>
</tr>
<tr>
<td>in SYSLOG. The snapshot is for the user calling ctSNAPSHOT().</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undo all user activations.</td>
<td>ctPSSclearUsers</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Snapshot action</th>
<th>opcode</th>
<th>rqstdesc</th>
<th>snapbufr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate the file with the specified file name. Use a pattern of &quot;*&quot; to activate</td>
<td>ctPSSaddFile-</td>
<td>File name (e.g., &quot;cust*.&quot;)</td>
<td>NULL</td>
</tr>
<tr>
<td>all files.</td>
<td>Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Activate the file with the specified file number.</td>
<td>ctPSSaddFileNo</td>
<td>Small integer file number (e.g., &quot;0&quot;)</td>
<td>NULL</td>
</tr>
<tr>
<td>On demand file snapshot returned directly in the output buffer for the file with</td>
<td>ctPSSfile</td>
<td>Small integer file number (e.g., &quot;3&quot;)</td>
<td>Address of output buffer large enough to hold at least a ctGFMS structure</td>
</tr>
<tr>
<td>file number specified by rqstdesc. No entry is made in SYSLOG.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Undo all file activations.</td>
<td>ctPSSclearFiles</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>
3.4.1.6 Combined Snapshot Support

The SnapShot API function supports logging combined snapshots containing c-tree Server system statistics and user and file statistics for the specified activated users and files. This ability is supported by the following operation:

<table>
<thead>
<tr>
<th>Snapshot action</th>
<th>opcode</th>
<th>rqstdesc</th>
<th>snapbufr</th>
</tr>
</thead>
<tbody>
<tr>
<td>On demand snapshot of system and activated users and files written to SYSLOG. All entries to SYSLOG share the instance ID, a 4-byte integer, returned in the output buffer.</td>
<td>ctPSSnow</td>
<td>An optional text description up to 128 bytes, or NULL</td>
<td>Address of a 4-byte integer to hold instance ID</td>
</tr>
</tbody>
</table>

3.4.2 Snapshot Contents

Each snapshot is comprised of a structure (ctGSMS for system info, ctGUMS for user info and ctGFMS for file info) followed optionally by a variable region. Each of the three snapshot structures contains information about the size of the structure and the amount of variable information, if any, following the structure. Further, if the output buffer is too small to hold all the variable information, the contents member of the main structure will have the ctSNAPSHOT-truncated bit turned on if information had to be truncated.

Note: These structure definitions may vary depending on the version of the c-tree Server and c-tree Plus SDK you are using. Please consult the c-tree headers for the actual structure definition that is in use. Differences in definitions between a given ctstat client and c-tree Server may make ctstat incompatible with that server, in which case the ctstat utility displays an error message indicating structure incompatibility.

The only variable information consists of an array of function timing results for the overall system or for a user. The array is composed of ctWRKSTT structures. Function timings are described below, but these timings are only for direct calls from c-tree clients.

Most of the elapsed times accumulated in the snapshots are based on high-resolution timers that report in numbers of ticks and are stored in eight-byte integers. The description following each structure member contains a "^" to indicate results based on high-resolution timers. To convert such times to seconds, divide by the ticks per second, a system dependent value stored in the scthrbastim member of each snapshot structure.

3.4.2.1 System Snapshot Structure

System snapshots may include an optional function-timing array of ctWRKSTT structures as described in the Function Timing section below.

typedef struct ctgsms {
    ULONG client_ver;      /* client version of structure */
    ULONG server_ver;      /* server version of structure */
} ctgsms;
ULONG fixlen; /* length of fixed portion of snapshot */
ULONG varlen; /* length of variable region (if any) */
ULONG contents; /* bit map of var len contents */
ULONG unused; /* available for use */
LONG8 snapshottm; /* snapshot time stamp: seconds since 70*/

LONG8 sctdnd_red; /* delete node thread queue reads */
LONG8 sctdnd_wrt; /* delete node thread queue writes */
LONG8 sctdnd_rwt; /* delete node thread queue rewrites */
LONG8 sctdnd_abn; /* delete node thread queue abandons */

LONG8 sctdnd_rmv; /* delete node thread queue removals */
LONG8 sctdnd_non; /* delete node thread queue no action */
LONG8 sctloktry; /* count of lock attempts */
LONG8 sctlokhlk; /* subcount of hdr lock attempts */

LONG8 sctlokdn; /* count of locks denied */
LONG8 sctlokbl; /* count of locks blocked */
LONG8 sctlokhhb; /* subcount of header blocks */
LONG8 sctlokdl; /* count of dead locks */

LONG8 sctlokfr; /* count of locks freed */
LONG8 sctlokrel; /* count of blocks released */
LONG8 sctblckur; /* current count of locks held */
LONG8 sctblckur; /* current count of blocked requests */

LONG8 scttrntim; /* cumulative transaction time^ */
LONG8 scttrncnt; /* cumulative transaction count */
LONG8 scttrnmx; /* maximum elapsed tran time^ */
LONG8 scttrnav; /* available for use */

LONG8 sctttot_call; /* system-wide c-tree calls */
LONG8 sctttot_recv; /* system-wide time waiting for request^*/
LONG8 sctttot_work; /* system-wide time performing requests^*/
LONG8 sctttot_send; /* system-wide time to send request^ */

LONG8 sctmemhgh; /* system memory highwater mark */
LONG8 sctmemsum; /* current aggregate sum */
LONG8 sct_dbrqs; /* data buffer requests */
LONG8 sct_dbhit; /* data buffer hits */

LONG8 sct_iqbrqs; /* index buffer requests */
LONG8 sct_iqhit; /* index buffer hits */
LONG8 sct_rqops; /* number of read operations */
LONG8 sct_rqbyt; /* bytes read */

LONG8 sct_wqops; /* number of write operations */
LONG8 sct_wrbyt; /* bytes written */
LONG8 sct_rccops; /* number of comm read operations */
LONG8 sct_rccbyt; /* comm bytes read */

LONG8 sct_wcops; /* number of comm write operations */
LONG8 sct_wcbyt; /* comm bytes written */
LONG8 sctwlgops; /* number of log write operations */
LONG8  sctwlgbyt;  /* bytes written to log file */

LONG8  sctr1gops; /* number of log read operations */
LONG8  sctr1gbyt; /* bytes read from log file */
LONG8  sctxlgops; /* number of log extension operations */
LONG8  sctxlgbyt; /* log file extension bytes */

LONG8  sct_trbeg; /* # transaction begins */
LONG8  sct_trend; /* # transaction ends */
LONG8  sct_trabt; /* # transaction aborts */
LONG8  sct_trsav; /* # transaction savepoints */

LONG8  sct_trrst; /* # transaction restores */
LONG8  sct_trfls; /* # transaction log flush writes */
LONG8  sctsync_dosfls; /* DOSFLUSH sync calls */
LONG8  sctsync_logfil; /* transaction log sync calls */

LONG8  sctsync_regfil; /* c-tree file sync calls */
LONG8  sctchrtimbas; /* high res timer ticks per sec^ */
LONG8  sctchkpnttim; /* cumulative checkpoint time^ */
LONG8  sctchkpntsiz; /* cumulative checkpoint size */

LONG8  sctcmtdlytry; /* cmtdly eligible count */
LONG8  sctcmtdlycnt; /* cmtdly concurrent trans exist */
LONG8  sctcmtdlycoh; /* cmtdly cohorts exist (success) */
LONG8  sctcmtdcohmax; /* cmtdly max cohort count */

LONG8  sctcmtdlyclr; /* cmtdly cohort appeared (success) */
LONG8  sctcmtdlopclr; /* cmtdly appeared loop count */
LONG8  sctcmtdlyfls; /* cmtdly log flushed anyway (success) */
LONG8  sctcmtdlopfls; /* cmtdly log flush loop count */

LONG8  sctcmtdlydfr; /* cmtdly loop defer count */
LONG8  sctcmtdlyfls2; /* cmtdly log flushed anyway2 (success) */
LONG8  sctcmtdlopfls2; /* cmtdly log flush2 loop count */
LONG8  sctcmtdlynot; /* cmtdly direct flush count */

LONG8  stimchk[8]; /* diagnostic time ct_udefer check */

LONG8  selapwrktim; /* elapsed time func timings turned on */
LONG8  selapsrvtim; /* elapsed server operation time */
LONG8  sctcmtblktim; /* accum time for cmtdly blocks */
LONG8  sctcmtdclrtim; /* accum time for cmtdly clears */

LONG8  stimchktarget; /* ct_udefer expected result */
LONG8  stimchkconsec; /* time between consecutive hrt in usec */
LONG8  stimchkavg; /* avg time ofr hrtimer call in usec */
LONG8  available8[17]; /* available for future use */

LONG  sctactfil; /* # open physical files */
LONG  scttotfil; /* # open logical files */
LONG  scttotblk; /* # c-tree FCBs in use */
LONG  sctactfilx; /* max physical files opened */
LONG  scttotfilx; /* max logical files opened */
LONG scttotblkx; /* max c-tree FCBs in use */
LONG sctnusers; /* number of users */
LONG sctnusersx; /* max number of users */
LONG sctloknm; /* net locks over unlocks */
LONG sctloknmx; /* max net locks over unlocks */
ULONG sct_mxbuf; /* number of index buffer pages */
ULONG sctbufcnt; /* index buffer pages in use */
ULONG sctbufgh; /* max index buffers in use */
ULONG sct_dxbuf; /* available data cache pages */
ULONG scmdatcnt; /* data cache pages in use */
ULONG scmdatgh; /* max data cache pages in use */
ULONG scmtblkseq; /* not flow through cmtdly block */
ULONG sct_compflg; /* compatibility flag */
ULONG sct_compflg2;
ULONG sct_diagflg; /* diagnostic flag */
ULONG sct_diagflg2;
LONG sct_cmtdly; /* commit delay */
LONG sct_chkdly; /* checkpoint tranac_cnt delay */
ULONG sct_cmtdly; /* cmtdlytim block to clear ratio */
ULONG sct_cmtdly; /* cmtdlytim cohort size measure */
ULONG stimchkusec; /* ct_udefer test interval in usec's */
ULONG stimuli; /* ct_udefer test iterations */
ULONG sct_udefer_thld; /* ct_udefer usec threshold */
ULONG sct_udefer_64yd; /* sleep duration for 64 yields (usec) */
ULONG available4u[11]; /* available for use */
ULONG sct_dxspillmt; /* avail data file special cache pages */
ULONG sct_dxsplcnt; /* actual data file special cache pages */
ULONG sct_dxsplhgh; /* maximum data file special cache pages */
ULONG sctcpcnt; /* check point count */
LONG sct_numvfil; /* number of virtual files open */
LONG sct_avlfil; /* # available file control blocks */
LONG sctactusr; /* number of threads active in foreground */
LONG sct_nutcnt; /* # of index buffers on upd list (tran) */
LONG sct_nucnt; /* # of index buffers on upd list */
LONG sct_dutcnt; /* # of data caches on upd list (tran) */
LONG sct_dupcnt; /* # of data caches on upd list */
LONG available4[13]; /* available for future use */
COUNT sct_mxfil; /* total number of c-tree FCBs */
COUNT sct_ndsec; /* # of sectors per node */
UTEXT sflvr; /* server byte order */
UTEXT salgn; /* server alignment */
UTEXT spntr; /* server pointer size */
TEXT availt[25];
TEXT description[128]; /* optional text description */
}
ctGSMS, ctMEM * pctGSMS;
3.4.2.2 User Snapshot Structure

Users may define up to 8 different timing baskets. A begin mark call establishes a high resolution starting time for the specified basket. An end mark call increments the specified basket counter, adds the elapsed time to the timing basket, and clears the starting time. Attempting to begin a mark that is not clear or end a mark that has not begun results in an error. The baskets are numbered from 0 to 7. Their results appear in the user snapshot structure in the smarkcnt and smarktim arrays.

User snapshots may include an optional function-timing array of ctWRKSTT structures as described in the Function Timing section below.

typedef struct ctgums {
    ULONG   client_ver;     /* client version of structure */
    ULONG   server_ver;     /* server version of structure */
    ULONG   fixlen;         /* length of fixed portion of snapshot */
    ULONG   varlen;         /* length of variable region (if any) */
    ULONG   contents;       /* bit map of var len contents */
    ULONG   unused;         /* available for use */
    LONG8   snapshottm;     /* snapshot time stamp: seconds since 70*/
    LONG8   strntsum;       /* user trntime sum^ */
    LONG8   strntmax;       /* user trntime max^ */
    LONG8   scthrtimbas;    /* high res timer ticks per sec^ */
    LONG8   avail8[9];      /* available for use */
    LONG8   smarktim[8];    /* user mark elapsed times^ */
    ULONG   smarkcnt[8];    /* user mark counts */
    ULONG   slOWNR;         /* thread handle (OWNER) */
    ULONG   sisolev;        /* transaction isolation level */
    ULONG   sctops;         /* set operation state bit mask */
    ULONG   sctstate;       /* internal state bit mask */
    ULONG   sabnlog;        /* begtran log for abandoned tran */
    ULONG   strntcnt;       /* user trntime count */
    ULONG   sctutrbeg;      /* user tran begin count */
    ULONG   sctutrend;      /* user tran end count */
    ULONG   sctutrabt;      /* user tran abort count */
    ULONG   snbrfile;       /* number of files in use */
    ULONG   slogtime;       /* logon time */
    ULONG   srqstime;       /* last request time */
    ULONG   sctstate;       /* set operation state bit mask */
    ULONG   sctops;         /* set operation state bit mask */
    ULONG   sctstate;       /* internal state bit mask */
    ULONG   sabnlog;        /* begtran log for abandoned tran */
    ULONG   strntcnt;       /* user trntime count */
    ULONG   sctutrbeg;      /* user tran begin count */
    ULONG   sctutrend;      /* user tran end count */
    ULONG   sctutrabt;      /* user tran abort count */
    ULONG   snbrfile;       /* number of files in use */
    ULONG   slogtime;       /* logon time */
    ULONG   srqstime;       /* last request time */
    ULONG   sctstate;       /* set operation state bit mask */
    ULONG   sctops;         /* set operation state bit mask */
    ULONG   sctstate;       /* internal state bit mask */
    ULONG   sabnlog;        /* begtran log for abandoned tran */
    ULONG   strntcnt;       /* user trntime count */
    ULONG   sctutrbeg;      /* user tran begin count */
    ULONG   sctutrend;      /* user tran end count */
    ULONG   sctutrabt;      /* user tran abort count */
}
### 3.4.2.3 File Snapshot Structure

```c
typedef struct ctgfms {
  ULONG client_ver;    /* client version of structure */
  ULONG server_ver;    /* server version of structure */
  ULONG fixlen;        /* length of fixed portion of snapshot */
  ULONG varlen;        /* length of variable region (if any) */
  LONG8 snapshottm;   /* snapshot time stamp: seconds since 70*/
  LONG8 phyrec;        /* physical file size */
  LONG8 numrec;        /* logical file size */
  LONG8 sernum;        /* serial number */
  LONG8 nument;        /* active entries */
  LONG8 hghtrn;        /* tran# high water mark for idx */
  LONG8 tstamp;        /* update time stamp */
  LONG8 mxfilz;        /* max file size high word */
  LONG8 fredops;       /* file specific # of read ops */
  LONG8 fredbyt;       /* file specific # of bytes read */
  LONG8 fwrtops;       /* file specific # of write ops */
  LONG8 fwrtbyt;       /* file specific # of bytes written */
  LONG8 scthrtimbas;   /* high res timer ticks per sec^ */
  LONG8 avail8[9];
  ULONG idxhgt;        /* index height */
  ULONG fileid;        /* unique file id */
  ULONG servid;        /* unique server id */
  ULONG timeid;        /* time id# */
  ULONG nodsiz;        /* node size */
  ULONG reclen;        /* fixed data record length */
  ULONG logtyp;        /* permanent components of file mode */
} ctGUMS, ctMEM * pctGUMS;
```
ULONG maxkbl; /* maximum key bytes leaf-var */
ULONG maxkbn; /* maximum key bytes non leaf-var */
ULONG filtyp; /* flag for file type */
ULONG keylen; /* key length */
ULONG kmem; /* key member # */
ULONG nnmem; /* number of members */
ULONG suptyp; /* super file type */
ULONG maxmrk; /* maximum exc mark entries per leaf */
ULONG hdrseq; /* wrthdr sequence # */
ULONG floktry; /* total lock attempts */
ULONG flokhhlk; /* header lock attempts */
ULONG flokblk; /* total lock wait count */
ULONG flokhbk; /* header lock wait count */
ULONG flokdlk; /* deadlocks */
ULONG flokdnr; /* total locks denied */
ULONG flokfre; /* total locks freed */
ULONG flokrel; /* total blocks released */
ULONG flokcur; /* current count of locks held */
ULONG fbkcur; /* current count of blocked requests */
ULONG datlmt; /* max number of special cache pages */
ULONG datstpl; /* actual number of special cache pages */
ULONG bufcnt; /* number of buffer pages */
ULONG datcnt; /* number of data cache pages */
ULONG numchn; /* number of channels */
ULONG usrcnt; /* number of users with file open */
ULONG avail4[8];
ULONG contents; /* bit map of var len contents */
LONG segmax; /* maximum # of segments */
LONG seglst; /* # activated segments */
UTEXT updflg; /* update (corrupt) flag */
UTEXT ktype; /* file type flag */
UTEXT autodup; /* duplicate flag */
UTEXT deltyp; /* flag for type of idx delete */
UTEXT keypad; /* padding byte */
UTEXT flflvr; /* file flavor */
UTEXT flalgn; /* file alignment */
UTEXT flpntr; /* file pointer size */
UTEXT sfvnr; /* server byte order */
UTEXT salgn; /* server alignment */
UTEXT spntr; /* server pointer size */
TEXT availt[9];

TEXT filename[256]; /* file name */
} ctGFMS, ctMEM * pctGFMS;
3.4.2.4 Function Timing

If function timings are enabled by the DIAGNOSTIC_SNAPSHOT_WORKTIME option or by the ctPSStimeWorkOn opcode, system and user snapshots have a variable region comprised of an array of ctWRKSTT structures:

```c
typedef struct wrkstt {
    ULONG fnc;        /* function # used during compression of stats */
    ULONG cnt;        /* function call count */
    LONG8 tim;        /* high resolution elapsed function time */
} ctWRKSTT, ctMEM * pctWRKSTT;
```

The first such structure in the array is a summary: its fnc member is zero, and the cnt member indicates the number of actual functions that had non-zero call counts, and the tim member has the total of all the elapsed times for all the functions. Even if the output buffer (for an on-demand snapshot with results returned directly) is too small to hold all the function-timing results, the summary is "complete." The varlen member of the main structure indicates how many bytes of variable information have been returned. The contents member of the main structure will have the ctSNAPSHOTtruncated bit turned on if information had to be truncated.

Each subsequent entry in the array holds a function number, a call count, and a total elapsed time in the function.
Chapter 4

c-treeSQL Server Support for Additional Platforms

The c-treeSQL Server provides a high performance SQL interface into the proven core of the c-tree Server. Tailored for high volume production environments, the c-treeSQL Server includes optimizations such as sophisticated query rewrite techniques to improve nested query performance and join-order optimization to improve performance of queries joining many tables. The c-treeSQL Server extensively caches and buffers information for maximum transaction and query throughput.

FairCom offers two SQL-enabled Servers: the c-treeSQL Server and the c-treeSQL Server Java Edition. The c-tree SQL Server supports interactive SQL, embedded SQL, ODBC and JDBC.

Our c-treeSQL Server Java Edition provides the additional capabilities of stored procedures and triggers. These powerful features provide your applications huge performance gains by utilizing the server to store your most common SQL queries and activities. A single call from your application can consistently and reliably execute complex transactions on the server.


The c-treeSQL Server is supported on the following platforms in this release:

- Windows
- Linux
- HP-UX
- Solaris
- AIX
- Mac OS X

c-treeSQL support on these platforms consists of the following technologies:

- c-treeSQL Server
- c-treeSQL ODBC and JDBC Driver
- c-treeSQL embedded SQL compiler (ESQL)
- c-treeSQL interactive SQL utility (ISQL)
Chapter 5

Memory Files

5.1 MEMORY FILES OVERVIEW

The c-tree Server and c-tree standalone database engine now support pure memory-resident data and index files. The distinguishing characteristic of memory files is that they exist solely in memory, occupying their own memory space separate from the data and index caches.

Memory files satisfy the need for the creation and manipulation of temporary data or index files that are always memory resident and never touch disk. Contrast this with non-memory files, whose contents may be paged out of the database cache and written to disk. Memory files are ideal for true temporary files: they can exist in memory as long as the c-tree Server or c-tree application process is running.

Some applications of memory files include:

- Temporary files: temporary files can be created as memory data or index files. The file contents are always memory resident, and when the memory file is finally closed, it ceases to exist.
- Storing read-only file contents: at server or application startup, the contents of a disk file can be read into a memory file and subsequent read requests can be satisfied from the memory file.
- In-memory list management: applications frequently need to maintain in-memory lists; a memory index provides easy creation and manipulation of items in a B+-tree-indexed list using the c-tree API, rather than requiring the developer to implement custom list management routines.

5.2 USING MEMORY FILES

The main operational differences between memory files and non-memory files involve creating and closing the files. After memory files are created, they are accessed using the standard c-tree API functions, just as for non-memory files. This section discusses the following topics:

- How to create and share memory files
- How to monitor the size of memory files
- Limitations on memory file usage
- A tip for speeding c-tree Server shutdown when using large memory files
5.2.1 Creating Memory Files Using Server Configuration Keyword

The c-tree Server supports creating memory files using a server configuration keyword. This feature allows developers to create memory files using their existing application code, provided that the file is created using an Xtd8 create function such as \texttt{CreateIFileXtd8}. To create a memory file using the server configuration keyword, specify one or more entries of the form:

\texttt{MEMORY\_FILE <file name>\#<max size>}

where the file name may include wild card characters, and the maximum size is optional. If no maximum size is specified, then 4GB is used. If a file is being created and matches one of the MEMORY\_FILE file name entries, then it will be created as a memory file unless it is a super-file host, superfile member, mirrored, segmented or partitioned file.

To cause all possible files to be created as memory files, add the following configuration entry:

\texttt{MEMORY\_FILE *}

The MEMORY\_FILE keyword is useful to quickly test how a file or set of files will behave as memory files.

5.2.2 Creating Memory Files Programmatically

To create a memory file programmatically, include the ctMEMFILE attribute in the x8mode member of the file's XCREblk structure, and specify the maximum file size using the mxfilzhw and mxfilzlw members of the XCREblk.

An attempt to add a new record or key value that causes the memory file to exceed the specified file size limit fails, and returns c-tree error MAXZ\_ERR(667).

A temporary, memory-resident index file can be created independent of any data file. Simply call \texttt{CreateIndexFileXtd8} with an XCREblk set as described above.

See the pseudo-code below to create a memory file:

\begin{verbatim}
xCREblk xcreblk[] = {
    {ctMEMFILE, 0, 0, 104857600, 0,0,0,0,0,0, ctKEEPOPEN, 0,0,0,0,0},
    {ctMEMFILE, 0, 0, 104857600, 0,0,0,0,0,0, ctKEEPOPEN, 0,0,0,0,0}
};
...
if (CreateIFileXtd8(&vcustomer,NULL,NULL,0,NULL,NULL,xcreblk))
{
    ctrt_printf("\nCould not create file %d with error %d.\n",isam_fil,isam_err);
}
\end{verbatim}

5.2.3 Sharing Memory Files Created Programmatically

There is a subtle issue about how to get a memory-resident file to be a shared file. At create, it is exclusive. If it is created and immediately closed, to be reopened for shared access, it would disappear on the close, and the reopen would fail. You can use the \texttt{UpdateFileMode} c-tree API function to change the file mode from exclusive to shared:
CreateIFileXtd8(...);
UpdateFileMode(datno,ctSHARED);
UpdateFileMode(keyno_1,ctSHARED);
...
UpdateFileMode(keyno_n,ctSHARED);

The **UpdateFileMode** call will fail if there are transaction-dependent actions pending. For example, if the **CreateIFileXtd8** call in the above pseudo code were for a transaction-dependent (ctTRANDEP) file, the **UpdateFileMode** call could not be called until the create is committed.

As noted above, the final close of a memory file causes all the existing memory file contents (data records or index nodes) to be lost. A final close refers to a file close that causes the user count of the file to drop to zero. It is possible to make the data file persist even after the final close by including ctKEEPOPEN bit in the splval member of the data file's XCREblk. Then the final close leaves the file open (but without any user attached to the file.) It can then be opened again, and the data still exists. The file will be closed when the server terminates, or when a call is made to the **CloseCtFileByName** c-tree API function:

**CloseCtFileByName** (pTEXT filnam,pTEXT fileword)

It is possible to use the ctKEEPOPEN flag on the create, and then close and reopen the file shared (without calling **UpdateFileMode**), but only if the file's creation is not pending commit. In this case, the sequence would be like:

XCREblk xcreblk[] = {
    {ctMEMFILE, 0, 0, 104857600, 0,0,0,0,0,0, ctKEEPOPEN, 0,0,0,0,0},
    {ctMEMFILE, 0, 0, 104857600, 0,0,0,0,0,0, ctKEEPOPEN, 0,0,0,0,0}
};
...
if (CreateIFileXtd8(&vcustomer,NULL,NULL,0,NULL,NULL,xcreblk))
{
    ctrt_printf("\nCould not create file %d with error %d.\n", isam_fil,isam_err);
}
else
{
    CloseFile(&vcustomer);
    if ((eRet = OpenIFileXtd(&vcustomer,NULL,NULL,NULL)) != 0)
    {
        ctrt_printf("\nUNEXPECTED ERROR %d ON REOPEN.\n",eRet);
    }
    else
    {
        ctrt_printf("\nFile created in memory.\n");
    }
}

5.2.4 Collecting Memory File Statistics

Memory file statistics are collected using the fields of the ctGFMS structure defined in ctporth. These statistics can be programmatically accessed by the developer using the c-tree
Server's performance monitoring utilities. The ctGFMS fields that are maintained for memory files include:

- **memcnt**: current number of memory records
- **hghcnt**: highest number of memory records
- **phyrec**: current total memory allocated for memory records
- **mhghbyt**: highest amount of memory allocated for memory records

### 5.2.5 Memory File Limitations

Memory files are subject to the following limitations:

- A memory file cannot be mirrored, partitioned, or segmented.
- A memory file cannot be backed up by a dynamic dump.
- The ISAM rebuild function **RebuildIFile** does not support rebuilding indexes associated with a memory data file; instead use the low-level **RebuildIIndex** function to rebuild individual indexes.
- The restorable delete (ctRSTRDEL) attribute is not supported for memory files.
- The atomic, recoverable transaction control attribute (ctTRNLOG) is not supported for memory files. However, the atomic, non-recoverable transaction control (ctPREIMG) and the transaction-dependent (ctTRANDEP) attributes are supported for memory files.

### 5.2.6 Tip: Faster Server Shutdown with Memory Files

When the c-tree Server shuts down, it frees individual memory records for memory files that are still open. If large memory files are open, the freeing of the memory records can prolong the server shutdown time. Adding the COMPATIBILITY MEMORY_FILE_SKIP_FREE keyword to the server configuration file causes the server to skip the freeing of individual memory records. When the server process terminates, the operating system will automatically reclaim the memory allocated to memory file records.
Chapter 6

c-tree Plus for .NET

6.1 .NET OVERVIEW

Microsoft® .NET is software for connecting people, information, systems, and devices. c-tree Plus for the Microsoft .NET Framework ("c-tree Plus for .NET") is an easy to use, high-level API, which abstracts the functionality of FairCom’s traditional ISAM and Low-Level APIs. c-tree Plus for .NET is intended as the new standard for c-tree Plus programming in .NET environments.

FairCom has combined the flexibility and performance of the original c-tree API's, with the ease and enhanced productivity found in the .NET architecture. Now, developers can focus on rapid deployment of their mission critical applications rather than how to best implement the low level details of the database functionality.

This section provides only a brief glimpse of what is possible with this new API. You are encouraged to download the complete manual of this new c-tree Plus for .NET architecture. The complete manual contains the entire API class reference as well as tutorials. You can find this in the online documentation section of our Web site located at:

http://www.faircom.com/support/mans.shtml
6.2 **C-TREE .NET ARCHITECTURE**

The c-tree Plus for .NET general architecture is presented in the figure below, organized into seven different levels: session, database, table, field, index, segment, and record. These levels or layers will be used to present a group of common functionality.

A **session** represents a connection between a client and a c-tree Server; no work can be performed before a session becomes active. The session object indicates the c-tree Plus for .NET session, the server name and location, the directory where the databases are located, the user name and password.

A **database** can be considered as a collection of tables, and each database has its own database dictionary that stores information about each table that belongs to that database: the table name, password and path, the active (open) tables, and the number of tables linked to the database. The database object indicates a database in the session and each session can have multiple databases.
A **table** is essentially a c-tree Plus data file and optional index files. There can be, and typically are, more than one table in a database, and a given table may belong to multiple databases. A table may have zero or more records.

A **field** is the basic element of a table, and a collection of fields form a data record. Often a table will have zero or more **indices**, which enhance the retrieval of records from that table.

Indices typically have one or more **segments** that describe the index key structure. The index object indicates an index associated with a particular table, while the segment links the index with the fields.

A **record** is essentially a row entry in a table. A record object indicates a record instance on a particular table. A table may have one or more record objects associated with it. Each record handle may be an independent cursor into the table, or several record objects may share the same cursor into the table.

**Note:** It is important to note that c-tree data and index files can be manipulated directly with or without session or database dictionary support.

### 6.3 C-TREE .NET QUICK GUIDE TO FUNCTIONS

This section describes, very briefly, some basic functionality of the .NET components. It is advised to consult the full documentation described above for complete descriptions of these powerful, yet flexible functions, as the examples shown below only cover a small fraction of what is available in this new interface.

#### 6.3.1 Working with Sessions

The c-tree Plus for .NET interface requires a session object to perform any data structure initialization or manipulation. The following steps should be executed within a session before any database or table operations are attempted:

- Construct a CTSession object
- Logon to a c-tree session by calling the CTSession.Logon method.

Only after initialization, should all database and table operations should be performed. To finalize all database operations when finished working with the database and tables, the following steps should be performed:

- Logout from the c-tree session by calling the CTSession.Logout method.
- Destroy the CTSession object
6.3.2 Creating a Session Object

A valid session object is required to perform any session operation. When a session object is created, we specify the session type with one of three types as shown in the table below. The default parameter of the CTSession constructor is SESSION_TYPE.CTDB_SESSION.

<table>
<thead>
<tr>
<th>SESSION_TYPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTREE_SESSION</td>
<td>Allocate a new session for logon only. No session or database dictionary files will be used. No database functions can be used with this session mode. You must allocate table handles using the session handle.</td>
</tr>
<tr>
<td>CTDB_SESSION</td>
<td>Allocate a new session making full usage of c-tree Plus for .NET session and database dictionaries. With this session mode, a Session dictionary file must exist to perform a session logon and you need to connect to a database before attempting to perform operation on tables.</td>
</tr>
<tr>
<td>SQL_SESSION</td>
<td>Allocate a new session for SQL processing. This session type is not implemented and reserved for future use.</td>
</tr>
</tbody>
</table>

In order to perform any database operations, it is necessary to log on to a c-tree session. A session is terminated with a session logout. To log on to a session, a CTSession object must be instantiated and then CTSession.Logon method should be called to perform the session logon.

Use the CTSession.CreateDatabase method to create a new database. CTSession.CreateDatabase takes a database name and the path where the database dictionary file is to be created. If the database path is empty ("") the database is created in the server directory, for client/server applications, or in the current directory for standalone applications.

When operations with the session are no longer needed, it is necessary to log out from the session by invoking method CTSession.Logout.

A summary of these activities is demonstrated in this brief code snippet:

```csharp
// create a default CTSession object
CTSession ASession = new CTSession();
try
{
    // logon to session using default
    // server name, user name and password.
    CTSession.Logon();
}
catch (CTException err)
{
    Console.Write("Session logon failed with error {0}", err.GetErrorCode());
}
```
// create a new database MyDatabase
try {
    ASession.CreateDatabase("MyDatabase", "");
} catch (CTException err) {
    Console.Write("Create database failed with error {0}\n", err.GetErrorCode());
}

try {
    // Logout of the session
    ASession.Logout();
} catch (CTException err) {
    Console.Write("Logon or Logout failed with error {0}\n", err.GetErrorCode());
}

### 6.3.3 Creating a New Session Dictionary

When operating with sessions of type SESSION_TYPE.CTDB_SESSION or SESSION_TYPE.SQL_SESSION, a session dictionary file named ctdbdict.fsd is required before a session logon is performed. The session dictionary file ctdbdict.fsd is located by default in either the server directory (client/server application) or in the application directory (stand-alone application).

There could be situations where the c-tree Plus for .NET session dictionary file does not exist because it was deleted or this is the very first time the system is being executed. In this case it is necessary to create a session dictionary file before attempting to log on to a session. It is important to note that only one session dictionary file is necessary for the normal operation of c-tree Plus for .NET. Once created, the session dictionary file can be used for all database and table operations.

The following code fragment shows an example on how to create a new session dictionary file:

```csharp
CTSession ASession = new CTSession();
try {
    ASession.Create("FAIRCOMS", "ADMIN", "ADMIN");
} catch (CTException err) {
    Console.WriteLine("Create session failed with error {0}", err.GetErrorCode());
}
```

In the server name, the full network address may be used (FAIRCOMS@10.0.0.1 or FAIRCOMS@my_machine.com for instance). For non-server applications, the parameter is ignored.
and may be set to NULL. A valid user name and password is required to access the c-tree Server.

6.3.4 Working with Databases

A database is a collection of tables and a session may contain several different databases. A database object is required before any table or data operations may take place. The following are typical operations performed on a database:

- Create a database object
- Connect to a database by calling CTDatabase.Connect
- Perform table, index, field and record operations
- When you are done with the database, disconnect by calling CTDatabase.Disconnect

6.3.4.1 Creating a Database Object

A valid database object is required to perform any database operation. You need to pass a valid CTSession object to the CTDatabase constructor.

Before performing any operations with a database, a database object must be connected to a session by calling the CTDatabase.Connect method. The database should already have been created or added to the session.

An existing table may be added or imported to a database by calling the CTDatabase.AddTable method. CTDatabase.AddTable takes as parameters the table name and the table path.

When the database connection is no longer needed, it must be disconnected from the session by calling the CTDatabase.Disconnect method.

The above activities are shown as a code example below:

```csharp
// create a CTDatabase object
CTDatabase ADatabase = new CTDatabase(ASession);
try
{
    // connect to database "MyDatabase"
    ADatabase.Connect("MyDatabase");
}
catch (CTException err)
{
    Console.Write("Database connect failed with error {0}\n", err.GetErrorCode());
}

// add MyTable to the current database
try
{
    ADatabase.AddTable("MyTable", "");
}
catch (CTException err)
{  
  Console.Write("Add table failed with error {0}\n", err.GetErrorCode());
}

try
{
    ADatabase.Disconnect();
}
catch (CTException err)
{
    Console.Write("Disconnect from database failed with error {0}\n",  
        err.GetErrorCode());
}

### 6.3.5 Database Object with Transaction Control

An extra level of data integrity can be achieved when you add an existing table to a database under transaction control. When the transaction is committed, the database dictionary data for the table is committed to disk. If the transaction is aborted, the dictionary data for the table is automatically removed from the database dictionary. The code fragment below shows how to add an existing table under transaction control.

// begin a transaction  
ADatabase.Begin();
try
{
    // add MyTable to the current database  
    ADatabase.AddTable("MyTable", "");
    // commit the transaction  
    ADatabase.Commit();
}
catch (CTException err)
{
    // abort the transaction  
    ADatabase.Abort();
    Console.Write("Add table failed with code {0}\n", err.GetErrorCode());
}

### 6.3.6 Working with Tables

A database may contain multiple tables, and a table may belong to multiple databases. Tables created using the c-tree Plus for .NET interface are kept as files with the extension .dat. Indices are stored in separate files with the extension .idx.

Creating a new table may be one of the most crucial operations performed by a database developer or administrator. The c-tree Plus for .NET API offers a powerful, yet easy to use, mechanism for creating tables.

The create table process involves the following steps as a minimum:

- Allocate a new table handle
• Add fields
• Add indices
• Create the table

6.3.7 Creating a Table Object

A valid table object is required before most table operations can take place. First, pass a valid CTDatabase object to the CTTable constructor.

The table is then populated with fields. The fields are defined at the time of the table creation. Fields are added to the record definition in the order they are declared. The c-tree Plus for .NET API also includes a set of functions that will allow a field to be inserted at a certain place in the record definition and fields can also be deleted from the record definition. The CTTable.AddField method will add a new field to the end of the record definition.

Once fields are defined, indices can be added. Indices are added to the table definition in the order they are declared. The c-tree Plus for .NET API also includes a set of functions that will allow an index to be deleted from the table index definition.

CTTable.AddIndex will add a new index at the end of the table index definition. For each index added to the table, one or more index segments should also be added to define which field combination forms a particular index. CTTable.AddSegment overloaded methods will accomplish the task of adding segments to an index.

After all fields and indices have been defined and the table properties set, it is time to create the table by calling the CTTable.Create method. This method takes as parameters the table name and the create mode.

After a successful open table, and if the table object is no longer needed, the table should be closed to allow all c-tree Plus for .NET, c-tree Plus and operating system buffers to be flushed to disk. It is very good programming practice to always close every open table before the process or thread is terminated.

These table activities are summed up in this example snippet:

```csharp
// create a new table object
CTTable ATable = new CTTable(ADatabase);

// add two fields to the table record definition
ATable.AddField("Field1", FIELD_TYPE.INTEGER, 4);
ATable.AddField("Field2", FIELD_TYPE.CHARS, 30);

// add index 0 - the first index
ATable.AddIndex("MyIndex1", KEY_TYPE.FIXED_INDEX, true, false);
// add index 0 segments
ATable.AddSegment("MyIndex1", "Field1", SEG_MODE.SCHSEG_SEG);
// add index 1 - the second index
ATable.AddIndex("MyIndex2", KEY_TYPE.FIXED_INDEX, false, false);
// add index 1 segments
ATable.AddSegment("MyIndex2", "Field2", SEG_MODE.SCHSEG_SEG);
```
6.3.8 Altering a Table

CTTable.Alter scans the table, field, index, and segment structures to decide which changes need to be made and how to perform the changes. At the very least, it may only update the table DODA if the only change made was, for example, in field types that are compatible with each other (e.g., changing from types CT_INT4 and CT_INT4U). Then, if the only changes occurred in a single index (e.g., a single index was added or deleted or the index properties changed), only that index is rebuilt. If more than one index changed, or more than one index was added or deleted, then it may be necessary to rebuild all indices of the table. If fields were added, deleted, inserted, or the changes in the field property were not compatible with each other, then a full rebuild of the table is performed.

A table is rebuilt by creating a temporary table with the correct current properties taking in consideration all changes. All records are read from the original table and written into the temporary table. Once all data records have been moved, the original table is deleted and the temporary table is renamed with the name of the original table.

// add one field to the table and rebuild it
CTTable ATable = new CTTable(ADatabase);
// open the table
ATable.Open("MyTable", OPEN_MODE.NORMAL_OPEN);
// add one field to the table
ATable.AddField("Wages", FIELD_TYPE.CURRENCY, 8);
// alter the table
try{
   ATable.Alter(ALTER_TABLE.NORMAL);
}catch (CTException err){
   Console.Write("Alter table failed with error {0}\n", err.GetErrorCode());
}

6.3.9 Working with Records

The c-tree Plus for .NET record management API hides from the user all the complexities of maintaining a generic record buffer for tables. While fixed length records may be easier to implement and maintain, variable length records do present a greater challenge for the developer.

When a table contains variable length records, each record read may require buffers of different sizes. The c-tree Plus for .NET record manager performs the function of expanding and shrinking record buffers to deal with the necessities of variable length records. Whereas the c-tree ISAM API requires different API calls for fixed length and variable length records, c-tree Plus for .NET presents to the user one common API that deals with both fixed and variable length records.

Only the application architecture and system resources limit the number of c-tree Plus for .NET record buffers associated with a table.

6.3.10 Creating a Record Object

A valid record object is required before most record operations can take place. You need to pass a valid CTTable object to the CTRecord constructor.

// create a CTRecord object
CTRecord ARecord = new CTRecord(ATable);

After the record object has been created, but before any record operations can take place, the table associated with the record object must be opened.

6.3.11 Adding a Record

To add a new record to a table, you need to perform the following actions:

- Clear the record buffer
- Populate the fields by calling the methods CTRecord.SetFieldAs...
- Add the record by calling CTRecord.Write
The following code demonstrates a record that has two fields assigned to it:

```csharp
// clear the record buffer
ARecord.Clear();
// populate the record buffer
ARecord.SetFieldAsSigned(0, 1000);
ARecord.SetFieldAsString(1, "Joe Smith");
// write the record
try
{
   ARecord.Write();
}
catch (CTException err)
{
   Console.WriteLine("Add record failed with error {0}\n", err.GetErrorCode());
}
```
Chapter 7

Index Ranges

7.1 INDEX RANGES OVERVIEW

This release introduces support for c-tree index ranges. Index ranges permit ISAM record reads in a manner similar to sets, but with more control. An index range is defined by specifying a range of values for one or more key segments. Each record retrieved must satisfy all the specified ranges.

The following segment range types are supported:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTIX_EQ</td>
<td>Equality (key == target)</td>
</tr>
<tr>
<td>CTIX_GT</td>
<td>Strictly greater than (key &gt; target)</td>
</tr>
<tr>
<td>CTIX_GE</td>
<td>Greater than or equal (key &gt;= target)</td>
</tr>
<tr>
<td>CTIX_LE</td>
<td>Less than or equal (key &lt;= target)</td>
</tr>
<tr>
<td>CTIX_LT</td>
<td>Strictly less than (key &lt; target)</td>
</tr>
<tr>
<td>CTIX_NE</td>
<td>Not equal (key != target)</td>
</tr>
<tr>
<td>CTIX_BET</td>
<td>Between inclusive (targetLow &lt;= key) &amp;&amp; (key &lt;= targetHigh)</td>
</tr>
<tr>
<td>CTIX_BET_IE</td>
<td>Between inclusive lower range (targetLow &lt;= key) &amp;&amp; (key &lt; targetHigh)</td>
</tr>
<tr>
<td>CTIX_BET_EI</td>
<td>Between inclusive upper range (targetLow &lt; key) &amp;&amp; (key &lt;= targetHigh)</td>
</tr>
<tr>
<td>CTIX_BET_EE</td>
<td>Between exclusive (targetLow &lt; key) &amp;&amp; (key &lt; targetHigh)</td>
</tr>
<tr>
<td>CTIX_NOTBET</td>
<td>Not between (key &lt; targetLow)</td>
</tr>
</tbody>
</table>

You do not need to specify a range for all segments, but you cannot skip a segment. For instance, you could specify a range for the first segment or the first and second segment, etc. If you wanted to specify ranges for the first and third segment, you would need to specify ranges for the first three segments, and you would make the second segment all inclusive (for example, using a segment range such as CTIX_GE with a zero lower limit).
Each user can have only one range definition for each index. Specifying a new range for an index automatically frees an existing range definition for that index.

7.2 USING INDEX RANGES FROM THE C-TREE ISAM API

Applications that use the c-tree ISAM API follow these steps to use index ranges:

- Allocate an index range.
- Read records using index range functions.
- Free the index range.

7.2.1 Allocating an Index Range

To allocate an index range using the c-tree ISAM API, call the `AllocateRange` c-tree Plus API function. The function prototype is as follows:

```c
ctCONV COUNT ctDECL AllocateRange(COUNT keyno, NINT segcount, pVOID lrange,
                                  pVOID urange, pNINT operators);
```

where

- `keyno`: Specifies the index file number;
- `segcount`: Specifies the number of segments with range specifications;
- `lrange`: Points to a buffer containing the limits for the segments;
- `urange`: Points to an optional buffer containing upper limits for the segments with between ranges (e.g., CTIX_BET); and
- `operators`: Points to an integer array of segment range types.

`AllocateRange` returns NO_ERROR on success.

Index ranges are ISAM context safe. That is, each ISAM context initiated via an `OpenISAM-Context` call can support its own index range.

The `lrange` and `urange` buffers must each be long enough to hold the first `segcount` segments for the index. For example, if `segcount` is 3, and if the first three segments are 10, 8, and 4 bytes long, then the range buffer(s) must be 22 bytes long.

If between ranges are not used, the `urange` buffer is ignored and may be passed as NULL. If between ranges are defined, then there must be a `urange` buffer, and it must be long enough to hold all `segcount` segments (in our example 22 bytes) even if only one between range segment is defined. For instance, if the first segment, 10 bytes long, does not use a between range, but the second segment does, then `urange` must hold the upper limit for the second segment starting at 10 bytes from the beginning of `urange`. Continuing this example, here is some pseudo code that assumes all the segments involved hold ASCII strings:

```c
/*
 ** 3 segment example for keyno 12
*/
```
NINT ops[3] = {CTIX_LT, CTIX_BET, CTIX_NE};
TEXT lbuf[22];
TEXT ubuf[22];
COUNTrc;

memcpy(lbuf,"TUVWXYZ   ",10);
memcpy(lbuf + 10,"00001000",8);
memcpy(lbuf + 18,"AX00",4);

memset(ubuf,0,22);
memcpy(ubuf + 10,"00001999",8);

rc = AllocateRange(12,3,lbuf,ubuf,ops);

The index must be part of an ISAM set of files and segcount must be greater than zero and cannot exceed the number of key segments actually defined for the index.

### 7.2.2 Reading Records Using an Index Range

If the call to `AllocateRange` succeeds, then `FirstInRange`, `FirstInRange`, `LastInRange`, or `LastInRange` must be called to activate the range and return the first or last data record in the range. If no record satisfies the range, then these routines return INOT_ERR(101). If the first/last range call succeeds, then `NextInRange`, `NextInRange`, `PreviousInRange`, and `PreviousInRange` may be used to traverse the range returning the next or previous record in the range. INOT_ERR is returned if no more records satisfy the range. It is permissible to call the first/last range routines any time to re-establish position at the beginning or end of the range. These routines are declared as follows:

```c
ctCONV COUNT ctDECL FirstInRange(COUNT keyno, pVOID recptr);
cfCONV COUNT ctDECL LastInRange(COUNT keyno, pVOID recptr);
cfCONV COUNT ctDECL NextInRange(COUNT keyno, pVOID recptr);
cfCONV COUNT ctDECL PreviousInRange(COUNT keyno, pVOID recptr);
cfCONV COUNT ctDECL FirstInRange(COUNT keyno, pVOID recptr, pVRLEN plen);
cfCONV COUNT ctDECL LastInRange(COUNT keyno, pVOID recptr, pVRLEN plen);
cfCONV COUNT ctDECL NextInRange(COUNT keyno, pVOID recptr, pVRLEN plen);
cfCONV COUNT ctDECL PreviousInRange(COUNT keyno, pVOID recptr, pVRLEN plen);
```

The interpretation of the parameters in the range record read functions is the same as for the ISAM record read functions `FirstRecord`, `FirstVRecord`, etc. The range record read functions return NO_ERROR on success.

### 7.2.3 Freeing an Index Range

Call the `FRERNG` c-tree Plus API function to free a range definition. The function prototype is as follows, where keyno is the file number of the index with which the range is associated:

```c
ctCONV COUNT ctDECL FRERNG(COUNT keyno);
```
7.3 USING INDEX RANGES FROM THE C-TREEDB C API

Applications that use the c-treeDB C API follow the same steps to use index ranges as applications that use the c-tree ISAM API, but they call the c-treeDB C functions that are described in this section and the terminology is slightly different:

- Initiate an index range operation.
- Read records with index range in effect.
- Terminate an index range operation.

7.3.1 Initiating an Index Range Operation

To allocate an index range using the c-treeDB API, call the `ctdbRecordRangeOn` c-treeDB API function. The function prototype is as follows:

```c
CTDBRET ctdbRecordRangeOn(CTHANDLE Handle, NINT SegCount, pVOID lRange, pVOID uRange, pNINT operators);
```

The operation of `ctdbRecordRangeOn` is similar to the c-tree Plus API function `AllocateRange`. `ctdbRecordRangeOn` establishes a new range based on the key segment values specified in the `lRange` and `uRange` buffers, and the operators for each segment.

`Handle` is a record handle. `SegCount` indicates the number of index segment values that should be used for setting the range, and the number of operators, since there must be one operator for each key segment in `lRange` and/or `uRange`.

`lRange` is a buffer with the lower range segment values. Use the function `ctdbBuildTargetKey` to build the `lRange` buffer. `uRange` is a buffer with the upper range segment values. Use the function `ctdbBuildTargetKey` to build the `uRange` buffer.

`operators` is an array of operators. There must be one operator for each key segment in `lRange` and/or `uRange`. The operators `CTIX_EQ`, `CTIX_NE`, `CTIX_GT`, `CTIX_GE`, `CTIX_LE`, `CTIX_LT` are open-ended and use only the `lRange` buffer for range values and the equivalent key segment in `uRange` is ignored and maybe set to null (ACII \0 values). The operators `CTIX_BET`, `CTIX_BET_IE`, `CTIX_BET_EI`, `CTIX_BET_EE` and `CTIX_NOTBET` use both `lRange` and `uRange` buffers to establish the lower and upper bound values. `ctdbRecordRangeOn` returns `CTDBRET_OK` on success.

7.3.2 Reading Records With an Index Range In Effect

Once the range is set, use c-treeDB `ctdbFirstRecord`, `ctdbNextRecord`, `ctdbPrevRecord` and `ctdbLastRecord` functions to navigate the records in the specified range.

The range is set for all index entries that are situated between the lower bounds and upper bounds values. The segment values are stored in `lRange` and `uRange` buffers in the same order and type of the index segment definition.

If record sets and ranges are turned on for the same record handle, ranges take precedence over sets. That is, in this case the c-treeDB functions `ctdbFirstRecord`, `ctdbLastRecord`, `ctdbN-
extRecord and ctdbPrevRecord automatically call the c-tree Plus ISAM range functions FirstInRange, LastInRange, NextInRange and PreviousInRange respectively. When ranges are turned off but sets are still turned on, the record handle automatically redirects the same c-treeDB functions to call the c-tree Plus ISAM set functions FirstInSet, LastInSet, NextInSet, and PreviousInSet, respectively, until record sets are turned off, at which point the c-treeDB functions default to the c-tree Plus functions FirstRecord, LastRecord, NextRecord and PreviousRecord respectively.

7.3.3 Terminating an Index Range Operation

Call the ctdbRecordRangeOff c-treeDB API function to terminate an index range operation. The function prototype is as follows:

CTDBRET ctdbRecordRangeOff(CTHANDLE Handle);

cdbRecordRangeOff returns CTDBRET_OK on success. If no range operation is active, ctdbRecordRangeOff does nothing and returns CTDBRET_OK.

7.3.4 Determining if a Range Operation is in Effect

The ctdbIsRecordRangeOn c-treeDB API function indicates if a range operation is in effect. The function prototype is as follows:

CTBOOL ctdbIsRecordRangeOn(CTHANDLE Handle);

cdbIsRecordRangeOn returns YES to indicate that a range operation is active. If no range operation is active, ctdbIsRecordRangeOn returns NO.

7.3.5 c-treeDB Example Using the C API

/* display all records where age is greater than 65 */
void DisplayAll(CTHANDLE hRecord)
{
    UTEXT lRange[32];
    NINT op[1] = {CTIX_GT};
    NINT fldno = ctdbGetFieldNumberByName(hHandle, "age");
    CTDBRET eRet;

cdbClearRecord(hRecord);
ctdbSetFieldAsSigned(hRecord, fldno, 65);
ctdbSetDefaultIndex(hRecord, 0);
ctdbBuildTargetKey(hRecord, CTFIND_EQ, lRange, 32);
eRet = ctdbRecordRangeOn(hRecord, 1, lRange, NULL, op);
if (eRet == CTDBRET_OK)
{
    eRet = ctdbFirstRecord(hRecord);
    while (eRet == CTDBRET_OK)
    {
        TEXT str[128];
cdbGetFieldAsString(hRecord, 0, str, sizeof(str));
printf("%s\n", str);
eRet = ctdbNextRecord(hRecord);
7.4 USING INDEX RANGES FROM THE C-TREEDB C++ API

Applications that use the c-treeDB C++ API follow the same steps to use index ranges as applications that use the c-treeDB C API, but they call the c-treeDB C++ functions that are described below.

Three new methods (CTRecord class: `RangeOn`, `RangeOff` and `IsRangeOn`) were added to implement the c-treeDB index range functionality to the c-treeDB C++ API.

The `CTRecord::RangeOn` method is similar to the `ctdbRecordRangeOn` function and is used to establish a new index range operation. The function prototype is as follows:

```cpp
void CTRecord::RangeOn(NINT SegCount, pVOID lRange, pVOID uRange, pNINT operators);
```

The meaning of each parameter passed to `CTRecord::RangeOn` is identical to the corresponding parameter passed to `ctdbRecordRangeOn`.

Once a range is established, use `CTRecord::First`, `CTRecord::Last`, `CTRecord::Next` and `CTRecord::Prev` to navigate the records in the range.

The `CTRecord::RangeOff` method is similar to the `ctdbRecordRangeOff` function and is used to terminate an index range operation:

```cpp
void CTRecord::RangeOff();
```

The `CTRecord::IsRangeOn` method indicates if a range is in progress or not:

```cpp
CTBOOL CTRecord::IsRangeOn();
```
8.1 Blocking Record Reads Overview

The c-tree Server’s blocking record read feature permits an application to attempt a record read with an optional blocking condition, and if no records satisfy the read request, to wait until such a record exists or the block times out. This feature provides a convenient way for an application to process specified records from a file as they become available.

8.2 Using Blocking Record Reads

A blocking record read is performed by calling the `BlockingISAMRead` c-tree API function. The `BlockingISAMRead` function operates by performing the requested ISAM operation (FIRST, NEXT, etc.). If successful, it then checks if the optional blocking condition is satisfied. If no record was found or if the record does not satisfy the blocking condition, and a non-zero time-out is specified, then the read blocks (sleeps) for the specified time-out period. The sleep is interrupted if the target file is updated, and the process repeats.

Either a record is found that satisfies the condition, or the block times out. A time out condition is indicated by returning NTIM_ERR(156).

The function prototype for the `BlockingISAMRead` function is shown below:

```
ctCONV COUNT ctDECL BlockingISAMRead(COUNT filno, NINT opcode, LONG timeoutsec,
  pTEXT blockcond, pVOID target, pVOID recptr,
  pVRLEN plen)
```

*filno* is a file number of an open c-tree data or index file.

*opcode* specifies one of the following ISAM read operations:

<table>
<thead>
<tr>
<th>ISAM Read Operation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctBLKIREC_FIRST</td>
<td>Read first record in physical or key order</td>
</tr>
<tr>
<td>ctBLKIREC_NEXT</td>
<td>Read next record in physical or key order</td>
</tr>
<tr>
<td>ctBLKIREC_PREV</td>
<td>Read previous record in physical or key order</td>
</tr>
<tr>
<td>ctBLKIREC_LAST</td>
<td>Read last record in physical or key order</td>
</tr>
<tr>
<td>ctBLKIREC_GT</td>
<td>Read record with key value greater than target key</td>
</tr>
</tbody>
</table>
timeout sec specifies the number of seconds the blocking read will wait before returning if no record satisfies the optional blockcond argument or no record is found. Set timeoutsec to zero to return immediately.

blockcond is an optional conditional expression which may be a logical expression of the same form as used for conditional index support or a function callback as in a SetDataFilter call. Set blockcond to an empty string ("") to specify no condition.

target should be non-NULL only if filno specifies an index and the opcode requires a target value (e.g., ctBLKIREC_GT).

The record is read into the buffer pointed to by recptr.

plen should be NULL for a fixed-length read, and should point to the length of the output buffer (recptr) for a variable length read. *plen is updated to the actual record length upon successful return.

<table>
<thead>
<tr>
<th>ISAM Read Operation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctBLKIREC_GTE</td>
<td>Read record with key value greater than or equal to target key</td>
</tr>
<tr>
<td>ctBLKIREC_EQL</td>
<td>Read record with key value equal to target key</td>
</tr>
<tr>
<td>ctBLKIREC_LTE</td>
<td>Read record with key value less than or equal to target key</td>
</tr>
<tr>
<td>ctBLKIREC_LT</td>
<td>Read record with key value less than target key</td>
</tr>
</tbody>
</table>
Chapter 9

Server-Based Queues

9.1 SERVER-BASED QUEUES OVERVIEW

With c-tree Server-based queues, client programs, connected to the same server, can pass information among each other, emulating a powerful inter-process communications mechanism that is portable, scalable and distributed.

c-tree Server-based queues, also referred to as system queues, extend the concepts of c-tree inter-thread queue routines by providing two additional features:

• System queues operate across the client/server boundary.

• System queues can be named so that different clients connected to the same c-tree Server may share the same queue.

Server-based queues provide the foundation on which the c-tree Server's file notification feature is built. See the next chapter in this guide for details on the file notification feature.

9.2 USING SERVER-BASED QUEUES

This section discusses the tasks involved in using system queues:

• Creating or opening a system queue

• Writing to a system queue

• Reading from a system queue

• Retrieving the size of the next message in a system queue

• Retrieving the number of messages in a system queue

• Closing a system queue

9.2.1 Creating or Opening a System Queue

Before any operation can be attempted with a system queue, a process must create a new queue, or open an existing queue, by calling the function:

ctCONV NINT ctDECL ctSysQueueOpen(pTEXT qname,NINT qmode);

Parameter qname identifies the queue to be open. If the queue specified by qname does not exist, a new queue is created. Parameter qmode is currently reserved for future use and should
be set to zero. \texttt{ctSysQueueOpen} returns a queue handle on success, or a negative number as the error code.

Example:

```c
#include <ctreep.h>

int main(void)
{
    NINT eRet = 0;
    NINT hQueue = -1;
    TEXT buffer[256];
    NINT isam_init = 0;

    /* init ISAM */
    eRet = (NINT)INTISAMX(6, 7, 4, 6, 0, "ADMIN", "ADMIN", "FAIRCOMS");
    if (eRet != NO_ERROR)
    {
        printf("INTISAM failed with error %d\n", eRet);
        goto Exit;
    }
    isam_init++;
    /* create a new queue */
    hQueue = ctSysQueueOpen("MyQueue", 0);
    if (hQueue < 0)
    {
        eRet = -hQueue;
        printf("ctSysQueueOpen failed with error %d\n", eRet);
        goto Exit;
    }
    /* wait 100 ms for a new message */
    eRet = ctSysQueueRead(hQueue, buffer, sizeof(buffer), 100);
    /* check if read time-out */
    if (eRet == NO_ERROR)
    {
        printf("ctSysQueueRead succeeded\n");
    }
    else if (eRet == NTIM_ERR)
    {
        printf("ctSysQueueRead time-out\n");
    }
    else
    {
        printf("ctSysQueueRead failed with error %d\n", eRet);
    }
Exit:
    /* close the queue */
    if (hQueue >= 0)
    {
        eRet = ctSysQueueClose(hQueue);
        if (eRet != NO_ERROR)
        {
            printf("ctSysQueueClose failed with error %d\n", eRet);
        }
    }
    if (isam_init)
    {
        CLISAM();
        return (int)eRet;
    }
```

9.2.2 Writing to a System Queue

Data is placed on the queue by invoking \texttt{ctSysQueueWrite} or \texttt{ctSysQueueLIFOWrite} functions. \texttt{ctSysQueueWrite} adds new data at the end of the queue, while the \texttt{ctSysQueueLIFOWrite} function adds the data at the beginning of the queue, similar to a stack operation. The syntax for the queue write functions is shown here:

\begin{verbatim}
ctCONV NINT ctDECL ctSysQueueWrite(NINT qhandle,pVOID message,NINT msglen)
ctCONV NINT ctDECL ctSysQueueLIFOWrite(NINT qhandle,pVOID message,
NINT msglen)
\end{verbatim}

Parameter \texttt{qhandle} is a system queue handle returned by a call to \texttt{ctSysQueueOpen}. Parameter \texttt{message} is a pointer to a block of memory containing arbitrary data to be placed on the queue and \texttt{msglen} indicates how many bytes are pointed to by \texttt{message}. \texttt{ctSysQueueWrite} and \texttt{ctSysQueueLIFOWrite} return \texttt{NO_ERROR} on success.

Example using \texttt{ctSysQueueWrite}:

\begin{verbatim}
#include <ctreep.h>

int main(void)
{
    NINT eRet = 0;
    NINT hQueue = -1;
    TEXT buffer[256];
    NINT isam_init = 0;

    /* init ISAM */
    eRet = (NINT)INTISAMX(6, 7, 4, 6, 0, "ADMIN", "ADMIN", "FAIRCOMS");
    if (eRet != NO_ERROR)
    {
        printf("INTISAM failed with error %d\n", eRet);
        goto Exit;
    }
    isam_init++;
    /* create a new queue */
    hQueue = ctSysQueueOpen("MyQueue", 0);
    if (hQueue < 0)
    {
        eRet = -hQueue;
        printf("ctSysQueueOpen failed with error %d\n", eRet);
        goto Exit;
    }
    /* write text to a queue */
    eRet = ctSysQueueWrite(hQueue, "This is the first line", 22);
    if (eRet != NO_ERROR)
    {
        printf("ctSysQueueWrite failed with error %d\n", eRet);
        goto Exit;
    }
    /* read the message in the queue */
    eRet = ctSysQueueRead(hQueue, buffer, sizeof(buffer), ctWAITFOREVER);
    /* check if read time-out */
    if (eRet == NO_ERROR)
\end{verbatim}
printf("Read: %s\n", buffer);
else if (eRet == NTIM_ERR)
    printf("ctSysQueueRead time-out\n");
else
    printf("ctSysQueueRead failed with error %d\n", eRet);

Exit:
    /* close the queue */
    if (hQueue >= 0)
    {
        eRet = ctSysQueueClose(hQueue);
        if (eRet != NO_ERROR)
            printf("ctSysQueueClose failed with error %d\n", eRet);
    }
    if (isam_init)
        CLISAM();
    return (int)eRet;

Example with ctSysQueueLIFOWrite:

#include <ctreep.h>

int main(void)
{
    NINT eRet = 0;
    NINT hQueue = -1;
    TEXT buffer[256];
    NINT isam_init = 0;
    NINT i;

    /* init ISAM */
    eRet = (NINT)INTISAMX(6, 7, 4, 6, 0, "ADMIN", "ADMIN", "FAIRCOMS");
    if (eRet != NO_ERROR)
    {
        printf("INTISAM failed with error %d\n", eRet);
        goto Exit;
    }
    isam_init++;
    /* create a new queue */
    hQueue = ctSysQueueOpen("MyQueue", 0);
    if (hQueue < 0)
    {
        eRet = -hQueue;
        printf("ctSysQueueOpen failed with error %d\n", eRet);
        goto Exit;
    }
    /* write text to a queue using ctSysQueueWrite */
    eRet = ctSysQueueWrite(hQueue, "This is the first line", 23);
    if (eRet != NO_ERROR)
    {
        printf("ctSysQueueWrite failed with error %d\n", eRet);
        goto Exit;
    }
9.2.3 **Reading from a System Queue**

Data is read from the beginning of a queue by calling the function `ctSysQueueRead`. The syntax is:

```c
ctCONV NINT ctDECL ctSysQueueRead(NINT qhandle, pVOID buffer, NINT buflen, LONG timeout)
```

Parameter `qhandle` is a system queue handle returned by a call to `ctSysQueueOpen`. The timeout parameter specifies the time in milliseconds that `ctSysQueueRead` will block waiting for data. A timeout of `ctWAITFOREVER` causes `ctSysQueueRead` to block until data is available in the queue. Parameter `buffer` is a pointer to a memory block large enough to hold the next message in the queue, and `buflen` indicates the size in bytes of buffer. `ctSysQueueRead` returns NO_ERROR on success. `ctSysQueueRead` returns NTIM_ERR (156) - timeout error if the time specified by `timeout` parameter expires and no message is available.

For code examples on how to use `ctSysQueueRead`, please see the examples provided in Section 9.2.1 "Creating or Opening a System Queue" on page 9-1 and Section 9.2.2 "Writing to a System Queue" on page 9-3.
9.2.4 Retrieving the Size of the Next Message in a System Queue

The `ctSysQueueMlen` function can be used to obtain the length of the next available message on the queue. The syntax for `ctSysQueueMlen` is:

```
ctCONV NINT ctDECL ctSysQueueMlen(NINT qhandle, LONG timeout)
```

Parameter `qhandle` is a system queue handle returned by a call to `ctSysQueueOpen`. Parameter `timeout` specifies a time in milliseconds that `ctSysQueueMlen` will block if the queue is empty. A `timeout` value of `ctWAITFOREVER` causes `ctSysQueueMlen` to block until data is available in the queue. Use `ctSysQueueMlen` to determine the appropriate size of a buffer before calling `ctSysQueueRead`. `ctSysQueueMlen` returns the length of the next available message, or a negative value on error.

Example:

```c
#include <ctreep.h>

int main(void)
{
    NINT eRet = 0;
    NINT hQueue = -1;
    TEXT buffer[256];
    NINT isam_init = 0;
    NINT len;

    /* init ISAM */
    eRet = (NINT)INTISAMX(6, 7, 4, 6, 0, "ADMIN", "ADMIN", "FAIRCOMS");
    if (eRet != NO_ERROR)
    {
        printf("INTISAM failed with error %d\n", eRet);
        goto Exit;
    }
    isam_init++;
    /* create a new queue */
    hQueue = ctSysQueueOpen("MyQueue", 0);
    if (hQueue < 0)
    {
        eRet = -hQueue;
        printf("ctSysQueueOpen failed with error %d\n", eRet);
        goto Exit;
    }
    /* write text to a queue using ctSysQueueWrite */
    eRet = ctSysQueueWrite(hQueue, "This is the first line", 23);
    if (eRet != NO_ERROR)
    {
        printf("ctSysQueueWrite failed with error %d\n", eRet);
        goto Exit;
    }
    /* retrieve the size of the message in queue */
    len = ctSysQueueMlen(hQueue, 100);
    if (len < 0)
    {
        eRet = -len;
    }

    return eRet;
}
```
printf("ctSysQueueMlen failed with error %d\n", eRet);
goto Exit;
}
printf("Message length is %d\n", len);
/* check the len of the message */
if (len > sizeof(buffer))
{
    printf("buffer is not large enough to receive message\n");
goto Exit;
}
eRet = ctSysQueueRead(hQueue, buffer, sizeof(buffer), ctWAITFOREVER);
/* check if read time-out */
if (eRet == NO_ERROR)
    printf("Read: %s\n", buffer);
else if (eRet == NTIM_ERR)
    printf("ctSysQueueRead time-out\n");
else
    printf("ctSysQueueRead failed with error %d\n", eRet);
Exit:
    /* close the queue */
    if (hQueue >= 0)
    {
        eRet = ctSysQueueClose(hQueue);
        if (eRet != NO_ERROR)
            printf("ctSysQueueClose failed with error %d\n", eRet);
    }
    if (isam_init)
        CLISAM();
    return (int)eRet;
}

9.2.5 Retrieving the Number of Messages in a System Queue

ctSysQueueCount can be used to obtain the number of messages waiting in the system queue. The syntax for ctSysQueueCount is:

cTCONV NINT ctDECL ctSysQueueCount(NINT qhandle)

Parameter qhandle is a system queue handle returned by a call to ctSysQueueOpen. ctSysQueueCount returns the number of messages in queue, or a negative number on error.

Example:

#include <ctreep.h>

int main(void)
{
    NINT eRet = 0;
    NINT hQueue = -1;
    NINT isam_init = 0;
    NINT count;

    /* init ISAM */
eRet = (NINT)INTISAMX(6, 7, 4, 6, 0, "ADMIN", "ADMIN", "FAIRCOMS");
if (eRet != NO_ERROR)
{
    printf("INTISAM failed with error %d\n", eRet);
    goto Exit;
}
isam_init++;
/* create a new queue */
hQueue = ctSysQueueOpen("MyQueue", 0);
if (hQueue < 0)
{
    eRet = -hQueue;
    printf("ctSysQueueOpen failed with error %d\n", eRet);
    goto Exit;
}
/* retrieve the number of messages in queue */
count = ctSysQueueCount(hQueue);
if (count < 0)
{
    eRet = -count;
    printf("ctSysQueueCount failed with error %d\n", eRet);
    goto Exit;
}
printf("Number of messages in queue: %d\n", count);

Exit:
/* close the queue */
if (hQueue >= 0)
{
    eRet = ctSysQueueClose(hQueue);
    if (eRet != NO_ERROR)
        printf("ctSysQueueClose failed with error %d\n", eRet);
}
if (isam_init)
    CLISAM();
return (int)eRet;

9.2.6 Closing a System Queue

When the queue is no longer needed, it must be closed by calling the close function:

crCONV NINT crDECL ctSysQueueClose(NINT qhandle)

Parameter qhandle is a system queue handle returned by a call to ctSysQueueOpen function. ctSysQueueClose returns NO_ERROR on success.
10.1 FILE NOTIFICATION OVERVIEW

This release debuts one of our biggest enhancements to the c-tree Server. Notification is a novel evolution of our enhanced system queue support. Using the facilities of c-tree system queues, client applications can direct the c-tree Server to monitor a data file and place notification messages on a queue when changes are made to the file. Each notification message includes the following details:

- The type of operation (add, delete, or update)
- For a transaction-controlled file, the transaction number in which the change occurred
- The record offset of the modified record.

The following optional information can also be included in notification messages:

- The unique key value involved in the operation. (An update may return both old and new key values.)
- The record image involved in the operation. (An update may return both old and new record images.)
- The node name of the client that performed the operation.

10.2 USING FILE NOTIFICATION

To use the notification process the user must:

1. Open a system queue by calling `ctSysQueueOpen`, which returns a queue handle.
2. Call `ctNotify` to establish the notification process.
3. Call `ctSysQueueRead` to read notification messages from the queue.
4. When notifications are no longer needed, call `ctNotify` to stop notifications or `ctSysQueueClose` to close the notification queue.

10.2.1 Enabling Notification for Actions on a File

This section discusses the steps required to enable notification for actions on a file.
10.2.1.1 **Open Files for Which Notification is Requested**

To enable notification for actions on a c-tree data file, start by opening the data file using a c-tree file open API function. For example:

```c
COUNT datno;

datno = OpenFileWithResources(-1, "customer.dat", ctSHARE);
```

10.2.1.2 **Open a System Queue**

After opening the data file, open a system queue on which notification messages will be placed.

```c
NINT qhandle;

qhandle = ctSysQueueOpen("myNotificationQueue", 0);
```

10.2.1.3 **Establish Notification for the File**

After creating the system queue, establish notification for actions on the file using the `ctNotify` c-tree API function. The `ctNotify` function is used to cause the server to send messages to a system queue when the specified action is taken on the specified c-tree data and index files. The syntax for the `ctNotify` function is:

```c
ctCONV NINT ctDECL ctNotify(NINT opcode, NINT objhandle, NINT qhandle, NINT contents, NINT controls);
```

The `opcode` parameter specifies which actions on the resource should be notified. The following values are supported:

<table>
<thead>
<tr>
<th>Values</th>
<th>Explanation</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctNT_ADDREC</td>
<td>Notify that a new record was added to the data file</td>
<td>ISAM</td>
</tr>
<tr>
<td>ctNT_DELREC</td>
<td>Notify that a record was deleted</td>
<td>ISAM</td>
</tr>
<tr>
<td>ctNT_RWTREC</td>
<td>Notify that a record was modified</td>
<td>ISAM</td>
</tr>
<tr>
<td>ctNT_ISMUPD</td>
<td>Notify on any change (add, delete or rewrite) to data file. This is identical to <code>ctNT_ADDREC</code></td>
<td>ISAM</td>
</tr>
<tr>
<td></td>
<td><code>ctNT_DELREC</code></td>
<td><code>ctNT_RWTREC</code></td>
</tr>
<tr>
<td>ctNT_TOUCH</td>
<td>Notify that the file was updated, only once per transaction. This opcode cannot be used in combination with others and the contents parameter must be 0. No details of the update are conveyed, only a &quot;ping&quot; that the file has been touched.</td>
<td>ISAM</td>
</tr>
<tr>
<td>ctNT_PARTIAL</td>
<td>Notify that a notification request was started in the middle of a transaction and not all the updates generated a notification. One of the first four opcodes in this table must also be specified when using this option.</td>
<td>ISAM</td>
</tr>
</tbody>
</table>
The `objhandle` parameter is the file number of an ISAM data file and `qhandle` is a system queue handle returned by a call to `ctSysQueueOpen`.

The `contents` parameter determines what optional details are returned in the variable-length region of the notification message and may be set to the following values or any combination (by or'ing) of them:

<table>
<thead>
<tr>
<th>Values</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctNT_CON_UNQKEY</td>
<td>Unique key</td>
</tr>
<tr>
<td>ctNT_CON_NODNAM</td>
<td>Node name of actor</td>
</tr>
<tr>
<td>ctNT_CON_RECBUF</td>
<td>Record buffer on add or update (not on delete)</td>
</tr>
</tbody>
</table>

To receive different notification contents for different actions, multiple calls to `ctNotify` are required. For example, to get unique key values on a record delete notification, and full record images on either an add or rewrite notification, two `ctNotify` calls are required: one call with a `contents` field of `ctNT_CON_RECBUF` to set up the add and rewrite notification, and one call with a `contents` field of `ctNT_CON_UNQKEY` for the delete.

The `controls` parameter is reserved for future use and must be set to zero.

`ctNotify` returns `NO_ERROR` on success.

For example, to monitor ISAM updates to a data file, a call of the form below will cause each ISAM update to the data file specified by `datno` to generate an entry in the queue specified by `qhandle`.

```
ctNotify(ctNT_ISMUPD, datno, qhandle, 0, 0);
```

### 10.2.2 Receiving Notifications for Actions on a File

After establishing notification on a c-tree data file, use the `ctSysQueueRead` function to read notification messages from a system queue. Non-transaction ISAM updates are immediately processed and placed into the system queue by the notify system. Transaction (including pre-image) updates are not processed until they are committed. The following sections discuss the format of a notification message and how to read notification messages returned by `ctSysQueueRead`.

#### 10.2.2.1 Notification Queue Message Format

A notification message always starts with fixed a portion followed by an optional variable-length region.

The fixed portion is made by the `ctNOTBLK` structure defined in `ctport.h`:

```c
typedef struct notblk {
    ULONG action;    /* actual opcode */
    LONG   actor;    /* thread ID */
} ctNOTBLK;
```
The `contents` member determines what is in the variable-length region and should be the same used in the `ctNotify` call.

The variable-length portion may contain the following items, depending on the options specified in the `contents` parameter to `ctNotify`:

1. key values (present if `ctNT_CON_UNQKEY` or'ed in contents),
2. actor node name (present if `ctNT_CON_NODNAM` or'ed in contents),
3. full record image (present if `ctNT_CON_RECBUF` or'ed in contents)

These items always appear in this order, even if not all are specified to be returned.

### 10.2.2.1.1 Fixed Portion of Notification Queue Message

The members of the fixed portion of the notification queue message are set as follows:

The `action` member of the `ctNOTBLK` structure contains the particular `opcode` of the monitored event as follows:

<table>
<thead>
<tr>
<th>Values</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ctNT_ADDREC</code></td>
<td>A new record was added to the data file</td>
</tr>
<tr>
<td><code>ctNT_DELREC</code></td>
<td>A record was deleted</td>
</tr>
<tr>
<td><code>ctNT_RWTREC</code></td>
<td>A record was modified</td>
</tr>
<tr>
<td><code>ctNT_CLSOBJ</code></td>
<td>The file was closed (no one has it open)</td>
</tr>
<tr>
<td><code>ctNT_TOUCH</code></td>
<td>The file was updated</td>
</tr>
<tr>
<td><code>ctNT_PARTIAL</code></td>
<td>A notification request was started in the middle of a transaction and not all the updates generated a notification.</td>
</tr>
</tbody>
</table>

The `actor` member contains the thread ID of the user/client that performed the operation.
The `tranhw` and `tranlw` members contain the transaction number in which the operation occurred: `tranhw` is the high-word, `tranlw` the low-word.

The `opcode` member is the opcode originally requested in the `ctNotify` call. This may be different from the action member as the `action` is actual event that occurred, while `opcode` is the combination of events monitored. For instance if you call `ctNotify(ctNT_ISMUPD, ctNT_PARTIAL, ...)` the `action` member can be `ctNT_DELREC` while the `opcode` is `ctNT_ISMUPD | ctNT_PARTIAL`.

The `objhandle` member contains the data file number on which the event occurred or, in case the notification request is for an index, the host index file number on which the event occurred.

The `idxmemno` member contains the index member number on which the event occurred. The actual index file number is given by `objhandle + idxmemno`.

The `contents` member is the actual contents of the variable length part. This may be different from the `contents` parameter of the `ctNotify` call. For example, when the `action` performed is `ctNT_DELREC` no record is returned in the optional full record image part even if it was requested by the `ctNotify` call, and the `contents` value reflects this situation by not having `ctNT_CON_RECBUF` or'ed in.

The `controls` member is reserved for future use.

The `datahw` and `datalw` members are the high word and low word of the record offset in the data file after modification (add, rewrite, delete).

The `auxdhw` and `auxdlw` members, in case of a record update, contain the original record offset in the data file (high word and low word) whether or not the record has been moved.

The `varlen` member of the `ctNOTBLK` structure indicates the length of the variable-length message portion.

**10.2.2.1.2 Optional Key Values**

If the `contents` member of the `ctNOTBLK` contains `ctNT_CON_UNQKEY`, the notification message contains in the variable portion, just after the `ctNOTBLK`, information about the unique key generated. The information is stored in this way:

```c
COUNT rkeyno;
COUNT keylen;
TEXT  key[];  // buffer of keylen bytes */
```

If action is `ctNT_RWTREC` there is an additional field

```c
TEXT  oldkey[];  // buffer of keylen bytes */
```

`rkeyno` is the index number, starting from 1 relative to the data file (1 is the first index, 2 the second index…). It is set to 0 if there is no unique index.

`keylen` is the length of the key.

`key[]` is a buffer of `keylen` bytes containing the key stored now in the index file.
oldkey[] is a buffer of keylen bytes containing the old key stored in the index file before an update, whether or not the key changed.

### 10.2.2.1.3 Optional Actor Node Name

If the contents member of the ctNOTBLK contains ctNT_CON_NODNAM, the 32-byte node name appears next in the notification message (immediately following the ctNOTBLK structure or the key values if key values were requested).

```plaintext
TEXT nodnam[32];
```

nodnam[32] is a null terminated string containing the node name of the actor that modified the file causing the notification triggering.

### 10.2.2.1.4 Optional Full Record Image

If the full record image is requested and the operation is not a delete, the record length and record contents appear next in the notification message:

```plaintext
LONG reclen;
TEXT recbuf[]; /* buffer of reclen bytes */
```

reclen is the record length

recbuf[] is a buffer reclen bytes long containing the record that is currently contained in the datafile.

### 10.2.2.2 Reading Notification Messages

Due to the variable length nature of the notification message, it is necessary to provide the ctSysQueueRead function with a buffer large enough to contain the entire message; otherwise, the ctSysQueueRead call will fail with error TQUE_ERR (638) and the message is not read.

When not requesting any optional values (contents set to 0 in ctNotify call), the notification message length is fixed and the notification message size is sizeof(ctNOTBLK).

When requesting optional information, the message length size can be either retrieved by calling ctSysQueueMlen or evaluated by adding to the fixed message length the lengths of each option requested as described in the following table.

<table>
<thead>
<tr>
<th>ctNotify contents parameter</th>
<th>Message length</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctNT_CON_UNQKEY</td>
<td>+ 2* sizeof(COUNT) +</td>
</tr>
<tr>
<td></td>
<td>If the size in bytes (keylen) of the key of the first unique index in your IFIL is known: 2* keylen</td>
</tr>
<tr>
<td></td>
<td>Else MAXLEN, which is the maximum key length supported by c-tree Plus</td>
</tr>
<tr>
<td>ctNT_CON_NODNAME</td>
<td>+ 32</td>
</tr>
</tbody>
</table>
For instance, suppose that contents is ctNT_CON_UNQKEY|ctNT_CON_RECBUF and the key length is not known and the record length is fixed to 40, then the message length will be:

\[
\text{sizeof}(\text{ctNOTBLK}) + 2 \times \text{sizeof}(\text{COUNT}) + 2 \times \text{MAXLEN} + \text{sizeof}(\text{LONG}) + 40
\]

If the message length is known or can be estimated, a notification message can be read by passing the address of a buffer, whose size is greater or equal to the message size, the buffer size and the timeout value to \text{ctSysQueueRead}. For example:

```c
NINT qhandle; /* Set by call to ctSysQueueOpen */
NINT rc;
pTEXT buffer;

buffer = (pTEXT) malloc (msglen);
/* Read next available queue message with 5 second timeout. */
rc = ctSysQueueRead(qhandle,buffer,msglen,5000);
```

If the message length is not known, it is necessary to determine the size with a call to \text{ctSysQueueMlen}. This function takes the queue handle and an optional timeout, so it can be used to wait until the next message is available in the queue and to determine the size of the message. After determining the message size, allocate a sufficiently-sized buffer and pass its address and size to \text{ctSysQueueRead}. For example:

```c
NINT qhandle; /* Set by call to ctSysQueueOpen */
NINT rc;
NINT msglen;
pTEXT pbuffer;

/* Read size of next available queue message with 5 second timeout. */
msglen = ctSysQueueMlen(qhandle,5000);

/* Allocate buffer to hold queue message. */
pbuffer = (pTEXT)malloc(msglen);

/* Read next available queue message. */
rc = ctSysQueueRead(qhandle,pbuffer,msglen,0);
```

The following code demonstrates how to extract information from the message read from a notification queue.

```c
COUNT ProcessQueueMessage (pTEXT buff) /* buff is a pointer to the message */ { 
    /* cast the buffer to a ctNOTBLK structure */
    /* in order to make easier to extract the */
```
/* information in the fixed portion        */
pctNOTBLK pnotblk = (pctNOTBLK)buff;
/* set info to point at the beginning of the */
/* variable length part                      */
pTEXT info = buff + sizeof(ctNOTBLK);

crt_printf("action actor tranlw opcode objhandle contents
controls datalw varlen\n");
crt_printf("%8d %5d %6d %6d %9d %8d %8d %6d %6d\n",
    pnotblk->action, pnotblk->actor, pnotblk->tranlw, pnotblk->opcode,
    pnotblk->objhandle, pnotblk->contents, pnotblk->controls,
    pnotblk->datalw, pnotblk->varlen);

/* if the action is ctNT_CLSOBJ return */
if (pnotblk->action == ctNT_CLSOBJ)
    return (1);

/* if contents contains ctNT_CON_UNQKEY there */
/* is unique key information to extract      */
if (pnotblk->contents & ctNT_CON_UNQKEY)
{
    COUNT rkeyno;
    COUNT keylen;

    /* extract relative index number */
    cpybuf(&rkeyno, info, 2);
    /* step over the index number */
    info += 2;
    /* extract the key length */
    cpybuf(&keylen, info, 2);
    /* step over the key length */
    info += 2;

crt_printf("Unique Key:\n");
crt_printf("\tIndex number (relative) %d\n", rkeyno);
crt_printf("\tkeylen %d\n", keylen);
    /* your key value handling here         */
    /* the key starts at the memory pointed by info */
    /* and its length is keylen             */

    /* skip over the key */
    info += keylen;

    /* if the action is ctNT_RWTREC there is also */
    /* the old key value to retrieve        */
    if (pnotblk->action == ctNT_RWTREC)
    {
        /* your key handling code here */

        /* skip over the key */
        info += keylen;
    }
    else
    {

ctrt_printf("Unique Key [NOT REQUESTED]\n");
}

/* if contents contains ctNT_CON_NODNAM there */
/* is a node name information to extract */
if (pnotblk->contents & ctNT_CON_NODNAM)
{
    /* print the node name that is pointed */
    /* by info */
    ctrt_printf("Node Name: %s\n",info);
    /* skip over the nodename */
    info += 32;
}
else
{
    ctrt_printf("Node Name [NOT REQUESTED]\n");
}

/* if contents contains ctNT_CON_RECBUF */
/* there is a record image to extract */
if (pnotblk->contents & ctNT_CON_RECBUF)
{
    LONG notrln;
    /* extract the record length */
    cpybuf(&notrln, info, 4);
    /* skip over */
    info += 4;
    ctrt_printf("Record Buffer Length: %d\n",notrln);
    /* your record buffer handling here */
    /* the record buffer starts at info */
    /* and is notrln bytes long */
    info += notrln;
}
else
{
    ctrt_printf("Record Buffer [NOT REQUESTED]\n");
}
/* sanity check to verify that we reached */
/* the end of the message */
if ((buff + sizeof(ctNOTBLK) + pnotblk->varlen) != info)
{
    ctrt_printf("Message parsing problem\n");
}

return (0);

---

10.2.3 Disabling Notification for Actions on a File

Notification for actions on a file is terminated by a physical file close or an explicit call of the form:

ctNotify(opcode, objhandle, qhandle, contents, controls|ctNT_CTL_STOP);
where all the arguments must agree with a previous call to notify except that the \textit{controls} parameter must include the \texttt{ctNT\_CTL\_STOP} flag.

10.2.4 \textbf{Notification Callbacks}

Instead of passing the notification information to a queue, use the c-tree Server SK to make calls to \texttt{ctCallback} to associate a callback function to a notification.

\texttt{ctCallback} is used in the same manner as \texttt{ctNotify} except that the third parameter is a pointer to a callback function instead of a queue handle:

\begin{verbatim}
NINT ctCallback(NINT opcode, NINT objhandle, ctCallbackPtr cbptr,
                NINT contents, NINT controls);
\end{verbatim}

A notification setup with \texttt{ctCallback} causes the function pointed to by \texttt{cbptr} to be called (instead of a message written to a queue). This capability is only available with the Server SDK, and \texttt{ctCallback} can only be called from code compiled into the server itself (using the server SDK).

The prototype for the callback function pointer is:

\begin{verbatim}
typedef NINT (*ctCallbackPtr)(pVOID msg, NINT msglen, pVOID aux,
                              NINT auxlen);
\end{verbatim}

The callback function returns \texttt{NO\_ERROR} on success and a non-zero value on failure. Parameter \texttt{msg} and the optional parameter \texttt{aux} are input parameters. If both \texttt{msg} and \texttt{aux} are passed in, then they should be conceptually pasted together to form one long message.

It is important to note that as currently coded, the target file's header semaphore is held while the callback function is executed. Therefore the callback function cannot introduce pauses or delays or attempt to lock the header of the target file.
Chapter 11

New c-treeDB, ISAM and Low-Level Integration

11.1 OVERVIEW

c-treeDB, short for c-tree DataBase, is a high level, easy to use API abstracting the two popular FairCom APIs, ISAM and Low-level. c-treeDB is intended as the standard for c-tree Plus programming.

If you are creating a new application and have selected c-treeDB as the API of choice to access your data, there may be situations where you need to place calls directly into the ISAM or even low-level layers while remaining in your c-treeDB code. This may be to obtain certain specific services that may not be directly supported by c-treeDB or you may want to rewrite certain c-treeDB functionality to better suit your specific requirements.

It may be common to find situations where you have an existing application written using the ISAM or low-level API, however, you develop new modules using the c-treeDB API and will migrate the existing modules over time to c-treeDB. In either case you will need c-treeDB to support the mix of ISAM or low-level function calls with your c-treeDB code, specifically when you must work with a table's data and index files and record data.

The C functions and C++ methods described below will present new functionality supplementing the c-treeDB C and C++ APIs.

11.2 SWITCHING C-TREE INSTANCES

If a c-treeDB application has multiple sessions, it may be necessary to force a c-tree instance switch before directing calls to ISAM and low-level functions to ensure those calls are made in the correct context.

This is particularly important in LOCLIB applications were you have one session connected to a c-tree Server, the remote session, and another session performing local I/O. In this case, it is very important to closely control which c-tree instance you require before making calls into the ISAM and low-level function layers.

Almost all c-treeDB functions automatically perform a c-tree instance switch for you; you need only take concern with c-tree instance switching in the case where you make ISAM or low-level calls within c-treeDB code.
The following c-treeDB C function performs a c-tree instance switch:

```c
CTDBRET ctdbSwitchInstance(CTHANDLE Handle)
```

This call will force a switch to the c-tree Plus instance indicated by the Session handle. Each session handle has a unique c-tree instance id. When most c-treeDB functions are called, they automatically perform a c-tree instance switch. `ctdbSwitchInstance` is used before a call to a specific c-tree ISAM or low level function to ensure the correct instance is active before instantiating the call. You may pass any c-treeDB handle to `ctdbSwitchInstance`. `CTDBRET_OK` is returned on success.

Similarly, the following c-treeDB C++ method performs a c-tree instance switch:

```cpp
void CTBase::SwitchInstance()
```

This method will force a switch to the c-tree Plus instance indicated by the Session object. Each session object has a unique c-tree instance id. When most c-treeDB C++ methods are called, they automatically perform a c-tree instance switch. If any errors are detected, a CTException is thrown.

The following is an example demonstrating a server administration logon in a LOCLIB implementation then forcing a c-tree instance switch to the remote instance and calling some `ctreeUserOperation` function.

```c
/* declare and allocate the remote and local session handles */
CTHANDLE hRemote = ctdbAllocSession(CTSESSION_CTREE);
CTHANDLE hLocal = ctdbAllocSession(CTSESSION_CTREE);

/* logon to c-tree server using the remote session handle */
if (ctdbLogon(hRemove, "FAIRCOMS", "ADMIN", "ADMIN") != CTDBRET_OK)
    printf("Remote ctdbLogon failed\n");

/* logon to local session using the local session handle */
if (ctdbLogon(hLocal, "local", "ADMIN", "ADMIN") != CTDBRET_OK)
    printf("Local ctdbLogon failed\n");

/* perform a c-tree instance switch and call ctreeUserOperation function */
if (ctdbSwitchInstance(hRemote) != CTDBRET_OK)
    printf("ctdbSwitchInstance failed\n");
else
    CtreeUserOperation("!mkdir faircom", buffer, sizeof(buffer));
```

### 11.3 SWITCHING ISAM CONTEXTS

Each time a record handle is allocated with `ctdbAllocRecord`, the allocated record handle acquires its own ISAM context, which means each record position operates independently from the other records. Record operations that move the current record position of one record handle will not interfere with other record handles.

If a c-treeDB application requires a call to the ISAM or low-level functions, it should ensure those calls are made in the correct ISAM context. All c-treeDB record handling functions automatically perform an ISAM context switch.
The following c-treeDB C function is used to perform a context switch:

```
CTDBRET ctdbSwitchContext(CTHANDLE Handle)
```

This call will force a switch to the c-tree Plus ISAM context indicated by the record handle. Each record handle has a c-tree ISAM context id associated with it. When most c-treeDB record handling functions are called, they will automatically perform a c-tree ISAM context switch. `ctdbSwitchContext` is called before specific c-tree ISAM or low level calls to make sure the correct ISAM context is active before making those calls. The handle must be a record handle. No other handle is acceptable. CTDBRET_OK is returned on success.

Similarly, the following c-treeDB C++ method should be used to perform a context switch:

```
void CTRecord::SwitchContext()
```

This method will force a switch to the c-tree Plus ISAM context indicated by the record object. Each record object may have its own c-tree ISAM context id. If any errors are detected, a CTException is thrown.

The following code snippet demonstrates use of the `ctdbSwitchContext` function to call the c-tree ISAM function `ResetRecord`.

```
/* force a context switch */
if (ctdbSwitchContext(hRecord) != CTDBRET_OK)
    printf("ctdbSwitchContext failed\n");

/* call ResetRecord */
if (ResetRecord((COUNT)ctdbGetDatno(hRecord), SWTCURI))
    printf("ResetRecord failed\n");
```

### 11.4 Obtaining Table Data and File Number

Most c-tree ISAM and low-level functions require a data or index file number to operate correctly. Data file operations may require a data file number while all index operations will require an index file number.

The c-treeDB C and C++ APIs now provide several new functions and methods to extract the data and index file numbers from c-treeDB record or table handles.

#### 11.4.1 Obtaining Data File Number

The following c-treeDB function will retrieve a data file number (or “datno”) from a table handle or any handle that can be converted into a table handle such as a record, segment, index and field handles:

```
NINT ctdbGetDatno(CTHANDLE Handle)
```

Retrieve the table datno. Handle must be a table handle, or a handle that can be converted into a table handle. Return the table datno on success or -1 on failure. If `ctdbGetDatno` returns -1, the error code can be retrieved by calling the `ctdbGetError` function.

The following c-treeDB method will similarly retrieve a data file number from a table object:

```
NINT CTTable::GetDatno()
```
If the `GetDatno` method fails, a `CTException` is thrown.

An example using the c-treeDB C API:

```c
CTDBRET DeleteTable(CTHANDLE hSession, pTEXT tablename)
{
    CTDBRET Retval = CTDBRET_OK;
    CTHANDLE hTable = ctdbAllocTable(hSession);

    if (hTable)
    {
        /* open the table exclusive */
        if ((Retval = ctdbOpenTable(hTable, tablename, CTOPEN_EXCLUSIVE)) !=
            CTDBRET_OK)
            return Retval;

        /* delete a file */
        if ((Retval = (CTDBRET)DeleteRFile((COUNT)ctdbGetDatno(hTable)) !=
            CTDBRET_OK))
            return Retval;
    }
    else
        Retval = CTDBRET_NOMEMORY;
    return Retval;
}
```

### 11.4.2 Obtaining index file number

Three c-treeDB functions have been added to the c-treeDB API, which allow the retrieval of an index file number from a c-treeDB handle.

- `ctdbGetIdxno`, will retrieve an index file number from a c-treeDB handle and is declared as follows:

  ```c
  NINT ctdbGetIdxno(CTHANDLE Handle)
  
  Handle must be an index or segment handle.
  ```

- `ctdbGetIdxnoByName` will retrieve an index file number given an index name and is declared as follows:

  ```c
  NINT ctdbGetIdxnoByName(CTHANDLE Handle, pTEXT indexname)
  
  Handle must be a table handle, or a handle that can be converted into a table handle. IndexName is a string containing the index name.
  ```

To retrieve the index file number by index number, call the c-treeDB function `ctdbGetIdxnoByNumber` declared as follows:

```c
NINT ctdbGetIdxnoByNumber(CTHANDLE Handle, NINT index)

Handle must be a table handle, or a handle that can be converted into a table handle. index is a c-treeDB index number. The first index number is zero.
```

These c-tree DB functions will return the index number on success or -1 on failure. If -1 is returned, the error code is retrieved with a call to the `ctdbGetError` function.
Corresponding methods were added to the c-treeDB C++ API. The following method is used to retrieve the data file number from a CTTable object:

\[ \text{NINT CTTable::GetDatno()} \]

This retrieves the table datno. A CTException is thrown if an error occurs.

The following methods are used to retrieve the index file number:

\[ \text{NINT CTIndex::GetIdxno()} \]

This method retrieves the index file number from the index object.

\[ \text{NINT CTTable::GetIdxno(const CTString& IndexName)} \]

This method retrieves the index file number from the table object, given the index name.

\[ \text{NINT CTTable::GetIdxno(NINT index)} \]

This method retrieves the index file number from the table object, given the c-treeDB index number.

In all cases, if the GetIdxno method fails, a CTException is thrown.

Below is a snippet demonstrating the c-treeDB C API function:

```c
/* retrieve the first key of first index */
TEXT keyval[256];

if (FirstKey(ctdbGetIdxnoByNumber(hTable, 0)), keyval)
    printf("FirstKey failed\n");
```
Chapter 12

Record Batch Enhancements

12.1 RECORD BATCHES OVERVIEW

For years, the c-tree Server has supported batched record read and delete operations by key value. Batched operations allow a client to efficiently perform operations on a set of records in a c-tree data file. For example, a batch delete can be used to delete a subset of records or even all the records in a c-tree data file in a single call to the c-tree Server. Likewise, a batch read can be used to efficiently read a subset of records matching the specified key values, retrieving the records in blocks of the specified size.

The c-tree Server V8.14 introduces the following enhancements to c-tree batch support:

- Batch record read in physical order
- Batch record read in key decreasing order
- Batch record read under control of index range definition
- Batch record insertion

12.2 USING RECORD BATCH ENHANCEMENTS

To use the c-tree record batch enhancements, call the c-tree API function DoBatchXtd. This function takes the same parameters as DoBatch except that the mode parameter is a ULONG instead of a UCOUNT. The function prototype is shown below:

```c
ctCONV COUNT ctDECL DoBatchXtd(COUNT filno, pVOID request, pVOID bufptr,
VRLEN bufsiz, ULONG mode);
```

The following sections discuss the use of the new mode parameters.

12.2.1 Batch Read in Physical Order

A batch record read in physical order is performed by calling DoBatchXtd with a mode of BAT_PHYS, specifying a request block comprised of 5 LONG integers. The first three values are the standard btotal, bavail and breturn from the PKEYREQ structure. The last two values are the high-order and low-order four-byte record position. For the first call to DoBatchXtd, these last two LONG integers specify the starting position for the physical retrieval. On return from the first and subsequent calls to DoBatchXtd these last two LONG integers specify the
next record location (i.e., the physical record position following the last record included in the output buffer). Subsequent DoBatchXtd calls continue to return records in physical order until the logical end of the data file is reached (i.e., no more records remain in physical order).

**Note:** The next record position is returned for informational purposes. It is not used on a subsequent call (BAT_NXT) to retrieve additional records. The server maintains the starting point for the next batch retrieval internally.

### 12.2.2 Batch Read in Key Decreasing Order

A batch record read can be performed in key decreasing order by calling DoBatchXtd with a mode of BAT_LKEY. Such a call returns records having a key value less or equal to the specified key value.

### 12.2.3 Batch Read with Index Range Definition

A batch record read can be performed under control of an index range definition by following these steps:

1. Call AllocateRange to establish the range criteria.
2. Call DoBatchXtd with a mode of BAT_RNGE.

When the BAT_RNGE mode is specified, the batch call behaves as when using the BAT_PKEY mode except that the records selected satisfy the range criteria established by the call to AllocateRange. The siglen and target members of the partial key request (PKEYREQ) structure are ignored when the BAT_RNGE mode is specified.

### 12.2.4 Batch Record Insertion

The c-tree Server now supports inserting a batch of records into a c-tree data file. Batch functions can now be used to load records from one file into another file of the same format.

To batch load records into a file, call DoBatchXtd as follows:

1. Specify the BAT_INS mode and either BAT_RET_BLK or BAT_RET_REC to indicate the format in which the records are organized in the input buffer. See the next section for a discussion of the new BAT_RET_BLK batch read mode.
2. For the request parameter, pass a pointer to an array of 5 LONG integers immediately followed by a buffer containing the records to be inserted.
3. Set bufsiz to the size of this additional buffer in bytes. (Ordinarily, bufsiz specifies the size of the output buffer pointed to by bufptr, but bufptr is ignored in calls for BAT_INS.)
4. Only the third of the 5 LONG integers (breturn) is used on input. Set it to the number of records in the buffer following the 5 LONG integers.

On return from a BAT_INS call, only the DoBatchXtd return value is significant.
For transaction controlled files, the batch insertion operation is treated as one all or nothing operation. If no explicit transaction is started, each BAT_INS call will start and end its own transaction. Even if an explicit transaction is started, each call to BAT_INS is treated independently through save points. One failed BAT_INS call does not invalidate prior or subsequent BAT_INS calls.

An optional mode that can be used with the BAT_INS mode is BAT_INP_RAW. Its effect is described as follows:

- **BAT_INP_RAW**: In heterogeneous client/server implementations, turns off the conversion of record images, key values and record positions passed to a batch function.

**Note**: Currently, all calls for batch insertion (BAT_INS) behave as if BAT_INP_RAW is turned on, regardless of the actual mode parameter.

### 12.2.4.1 Additional Useful Batch Read Options

When using a batch read for the purpose of reading blocks of records to pass to a batch insert call, the following batch read modes may be useful to include in the batch read call:

- **BAT_RET_BLK**: Return a contiguous block of records as if they had been lifted directly from data file including VHDR structure for variable-length records. For batch reads, this mode can only be used with the BAT_PHYS mode. This mode is designed to return blocks of records in an appropriate format for a batch insert operation.

**Note**: Batch inserts using BAT_RET_BLK record organization should be marginally faster than BAT_RET_REC record organization when the target file has associated keys, and even better when it has no keys because the data records are written with one write operation when BAT_RET_BLK is in use. By comparison, BAT_RET_REC returns record images preceded by their 4 or 8 byte record position and 4-byte length which must be stripped before writing the records to the target file.

- **BAT_RET_RAW**: In heterogeneous client/server implementations, turns off the conversion of record images, key values and record positions returned by a batch retrieval call. This may prove especially useful with calls to retrieve a batch intended to be inserted into another file with BAT_INS.

- **BAT_LOK_ONE**: An alternative locking strategy: only locks the record during the record read; original locking strategy keeps locks on during entire batch processing.

- **BAT_LOK_BLK**: Converts BAT_LOK_RED and BAT_LOK_WRT to blocking read and write locks, respectively.

**Example:**

The following pseudo-code shows a strategy for using batch retrievals from one file to load into another file with the same record format:

```c
ULONG     batmode;
LONG      irq[8192];
pPKEYREQ  pirq = (pPKEYREQ) irq;
```
COUNT   retval;

CHGBAT(0);
batmode= BAT_GET | BAT_PHYS | BAT_RET_BLK | BAT_LOK_RED |
               BAT_LOK_BLK | BAT_LOK_ONE;

/*
** start batch retrieval at current ISAM position. [Can start at any
** legitimate record offset.]
*/
irq[4] = GETCURP(src_datno); /* low order word */
irq[3] = ctGETGH(); /* high order word */

/*
** first batch [0] retrieval
*/
retval = BATSETX(src_datno, pirq, irq+5,
               (VRLEN) sizeof(irq) - 5 * sizeof(LONG), batmode);

/*
** prepare batmode for subsequent retrievals
*/
batmode &= ~BAT_GET;
batmode |=  BAT_NXT;
TRANBEG(ctTRANLOG);

do {
    printf("\nstatus=%d tot=%ld avl=%ld ret=%ld nxthw=%lx nxtlw=%lx",
           retval, irq[0], irq[1], irq[2], irq[3], irq[4]);
    }

    if (retval)
        break;

    /*
    ** switch to batch [1] for insertion
    */
    CHGBAT(1);
    if (BATSETX(dest_datno, pirq, 0x1,
          sizeof(irq) - 5 * sizeof(LONG),
          BAT_INS | (batmode & (BAT_RET_REC | BAT_RET_BLK))))
    {
        printf("\nBATSETX BAT_INS: %d\n", isam_err);
        break;
    } else
        printf("\nBATSETX BAT_INS: success");

    CHGBAT(0);
    retval = BATSETX(src_datno, pirq, irq + 5,
               (VRLEN) sizeof(irq) - 5 * sizeof(LONG),
               BAT_NXT /*batmode*/);
} while (1);
TRANEND(ctFREE);

CHGBAT(0);
BATSETX(src_datno,NUL,NU,0,BAT_CAN);

CHGBAT(1);
BATSETX(dest_datno,NUL,NUL,0,BAT_CAN);
Chapter 13

Other Features by Product

13.1 C-TREE SERVER

This section discusses other significant features introduced in the c-tree Server V8.14.

13.1.1 Extended Transaction Number Support

The FairCom transaction processing logic used by the c-tree Server uses a system of transaction number high-water marks to maintain consistency between transaction controlled index files and the transaction log files. When log files are erased, the high-water marks maintained in the index headers permit the new log files to begin with transaction numbers which are consistent with the index files.

With previous releases, if the transaction number high-water marks exceed the 4-byte limit of 0x3ffffff0 (1,073,741,808), then the transaction numbers overflow, which will cause problems with the index files. On file open, an error MTRN_ERR(533) is returned if an index file's high-water mark exceeds this limit. If a new transaction causes the system's next transaction number to exceed this limit, the transaction fails with a OTRN_ERR (534).

6-byte transaction numbers essentially eliminate this shortcoming. With this new feature, 70,000,000,000,000 transactions can be performed before server restart. A transaction rate of 1,000 transactions per second would not exhaust the transaction numbers now available for over 2,000 years.

In c-tree Plus V8 and later, 6-byte transaction numbers are used by default. They will not be used on an individual file creation in the following cases:

1. If there is no extended create block; or
2. If the ctNO6BTRAN bit in the x8mode member of the extended file create block (XCREblk) is turned on; or
3. If the ctNO_XHDRS bit is turned on in x8mode.

You can override the default so that 4-byte transaction numbers are instead used by default by adding the COMPATIBILITY 6BTRAN_NOT_DEFAULT keyword to the server configuration file.

Ordinary data files are unaffected by this modification, and they are compatible back and forth between servers with and without 6-byte transaction support. Except for superfile hosts, the ct6BTRAN mode is ignored for data files. Index files and superfile hosts are sensitive to the
ct6BTRAN mode: (1) an existing index file or superfile supporting only 4-byte transaction numbers must be converted or reconstructed to change to 6-byte transaction number support; and (2) the superfile host and all index members of a superfile must agree on their ct6BTRAN mode (either all must have the ct6BTRAN mode on or all must have it off), or a S6BT_ERR(742) occurs on index member creation.

Note: A previously existing index will only use 4-byte transaction numbers and an attempt to go past the (approx.) 1,000,000,000 transaction number limit will result in an OTRN_ERR(534). See Section 3.11.4 “Transaction High Water Marks” in the c-tree Plus Programmer’s Reference Guide for more information.

Note: Files supporting 6-byte transactions are not required to be huge, but they do use an extended header.

An attempt to open a file using 6-byte transactions by code that does not support 6-byte transactions will result in HDR8_ERR(672) or FVER_ERR(43).

13.1.1.1 Configurable Extended Transaction Number Options

To check for files that do not support extended transaction numbers, add the following keyword to the c-tree Server configuration file:

DIAGNOSTICS EXTENDED_TRAN_NO

This keyword causes the server to log each physical open of a non-extended transaction number file to the CTSTATUS.FCS file. The reason to check for a file that does not support extended transaction numbers is that if all files do not support extended transaction numbers, then the exceptions could cause the server to terminate if the transaction numbers exceed the original 4-byte range and one of these files is updated. By "all files" we mean superfile hosts and indices; data files are not affected by the extended transaction number attribute.

To enforce the use of only files with extended transaction numbers, add the following keyword to the c-tree Server configuration file:

COMPATIBILITY EXTENDED_TRAN_ONLY

This keyword causes a R6BT_ERR(745) on an attempt to create or open a non-extended-transaction-number file. A read-only open is not a problem since the file cannot be updated.

These configuration options have no effect on access to non-transaction files, as transaction numbers are not relevant to non-transaction files.

13.1.2 Dynamic Advanced Encryption

FairCom offers developers a choice of using either standard FairCom proprietary file encryption or including other more advanced encryption routines, such as AES (Rijndael), Blowfish, or Twofish. Previously, the ability to use Advanced Encryption was set at compile time for c-tree Servers. With this release we offer users the ability to enable Advanced Encryption at run time, through the use of a server configuration keyword. When Advanced Encryption is enabled, the c-tree Server prompts for a master password at server startup. Developers can use
the c-tree Server SDK to replace this prompt with an application-specific method of retrieving the master password.

### 13.1.2.1 Enabling Advanced Encryption Support

Follow these steps to enable advanced encryption support:

1. Run the `ctcpvf` utility to generate an encrypted password for use when launching the Advanced Encryption enabled Server.

2. To enable Advanced Encryption, place the following keyword in the `ctsrvr.cfg` configuration file prior to launching the Server:

   ```
   ADVANCED_ENCRYPTION YES
   ```

   **Note:** Advanced Encryption is disabled by default. Any time you change the advanced encryption setting, you should delete the `FAIRCOM.FCS` file (which contains user and group information) before restarting the server. All user and group information must be recreated if `FAIRCOM.FCS` file is deleted.

3. Support for Advanced Encryption on the client side requires the client library to be built with the `#define ctCAMO` & `#define ctCAMOsdk` inserted in the c-tree makefile file as follows:

   ```
   echo #define ctCAMO >>$(fcTOM)/ctoptn.h
   echo #define ctCAMOsdk >>$(fcTOM)/ctoptn.h
   ```

### 13.1.2.2 Encrypting Files Using Advanced Encryption


**Note:** With the introduction of Dynamic Advanced Encryption definition, the c-tree Server supports both Standard and Dynamic Encryption. Accordingly, the definition of the mod parameter in `ctSETENCRYPT(pTEXT mod, pTEXT key, VRLEN keylen)` has been enhanced to allow `ctSETENCRYPT` to handle Standard Encryption as well as Advanced Encryption in the same Server executable. With Standard Encryption the parameter mod was unused. A NULL mod parameter now specifies Standard Encryption while not NULL specifies the type of Advanced Encryption.

### 13.1.3 Automatic Segmented File Support

On systems that do not support files greater than 2GB or 4GB, c-tree can still support huge files by creating tables as segmented files. The size of each segment stays below the OS maximum file size limit, but the aggregate size exceeds the limit.

A new feature has been added which permits a c-tree server configuration keyword to force any huge file to be created as a segmented file. The form of the configuration file entry is:

```
HUGE_TO_SEG_MB <segment size in MBs> [#<maximum # of segments>]
```
where the maximum number of segments is optional and defaults to sixteen (16). For example, to specify a segment size of 1GB, and a maximum of 8 segments for a total file size of up to 8GB, an entry would look like

```
HUGE_TO_SEG_MB   1024#8
```

If the file has been created with a maximum size in the XCREblk structure (see parameters for the extended create file function), then the number of segments will be computed to accommodate the maximum size.

**Note:** If dynamic dumps are used, then it would be appropriate to use the !EXT_SIZE script option so that the dump stream file would also be broken into automatic segments.

### 13.1.4 Enhanced Precision for International Languages

FairCom now supports the optional use of `CharUpper()` instead of `toupper()` when calling the c-tree function `ctrt_toupper()`. This enhanced support is for Windows only.

The differences between these two functions are rather subtle, but when dealing with international languages it becomes quite obvious. This is best understood using an example:

- `toupper()` converts "Téc1" to "TéC1"
- `CharUpper()` converts "Téc1" to "TÉC1"

### 13.1.4.1 Client/Server Support

Follow these steps to enable this feature in client/server mode:

1. Specify the following keyword in the c-tree Server configuration file:
   
   ```
   COMPATIBILITY USE_CHARUPPER
   ```

2. Add ```define ctUSE_CHARUPPER``` to the makefile used to compile your c-tree client library as shown below:

   ```
   echo #define ctUSE_CHARUPPER >>$(fcTOM)\ctoptn.h
   ```

### 13.1.4.2 Standalone Support

To enable this feature in standalone mode, add `define ctUSE_CHARUPPER` to the makefile used to compile your c-tree standalone library as shown below:

```
echo #define ctUSE_CHARUPPER >>$(fcTOM)\ctoptn.h
```

### 13.1.5 Server Logs Stack Trace in Case of Critical Error

For Servers running on Solaris, a new diagnostic feature was added that provides a stack trace showing calls for all threads. This is performed automatically when a fatal error occurs. The implementation invokes the `pstack` utility and redirects its output to the file `pstack<server_pid>_<sernum>.log`, where `<server_pid>` is the process ID of the c-tree Server process and `<sernum>` is a serial number maintained by the server to ensure unique log names. The server also writes a message to its status log, `CTSTATUS.FCS`, indicating that a process
stack trace was dumped. For example, the following message in _CTSTATUS.FCS_ refers to the file _pstack_454_01.log:

```
Dumped stack for server process 454, log=1, loc=73, rc=0
```

13.1.6 **Create a List of Arbitrary Names in Server Configuration File**

The c-tree Server now allows for a list of arbitrary names to be kept in the server's configuration file (_ctsrvr.cfg_). This name list is a user-defined, general-purpose name list that can be used by client applications.

A typical use for this feature is to keep a list of data or index files that require special processing by the client application. For example, an application may introduce system queue and notification processing only for data files that are named in an APP_NAME_LIST. A list may also contain user definitions of relationships between different data files.

13.1.6.1 **Application Name List Usage**

The new server configuration keyword APP_NAME_LIST permits the server configuration file to create a list of arbitrary names that can be retrieved by clients. The names are any text string not containing spaces. If spaces are desired, the text string can be placed in quotes, but the quotes will be part of the text string on retrieval.

As many APP_NAME_LIST entries as required may be created in the configuration file. For example: a list of file names may be created and then the names of the files can be retrieved by the client application. Further, by convention, one could use a delimiter to add parameters to the end of the string. Assuming, for the sake of example, that a pound sign (#) is used as a delimiter, then some entries could be added as follows:

```
APP_NAME_LIST       first.dat#parm11#parm12#parm13
APP_NAME_LIST       second.dat#parm21#parm22#parm23#parm24
```

It is up to the client application to determine how to parse the APP_NAME_LIST string that is returned in its entirety by calls to the c-tree _GetSymbolicNames_ API function.

13.1.6.2 **Application Name List Restrictions**

The text string used in APP_NAME_LIST entries have the following restrictions:

1. The text string cannot contain a vertical bar (|) character; and
2. The string cannot exceed the maximum file name length (255 characters, or 512 characters when multi-byte characters are enabled).

While any delimiter can be used in the APP_NAME_LIST text string, except for the vertical bar character (|), FairCom recommends the pound character (#). Also, if the server is likely to be used by applications from multiple vendors, FairCom further recommends that the first parameter of an APP_NAME_LIST text string be considered a sub-list name so that different applications can select only the items that are relevant to them. For example:

```
APP_NAME_LIST       first.dat#LIST_1#parm12#parm13
```
Use c-tree function `GetSymbolicNames` to retrieve the APP_NAME_LIST strings. The first APP_NAME_LIST text string is retrieved with the following call:

```
GetSymbolicNames(-1, buffer, buflen, FIRST_ITEM);
```

If `GetSymbolicNames` is successful, it returns NO_ERROR and the first APP_NAME_LIST in the server's configuration file is placed in buffer. If buflen is too small, `GetSymbolicNames` returns error VBSZ_ERR(153). If no APP_NAME_LIST entry is found, `GetSymbolicNames` returns error INOT_ERR(101).

The second and any subsequent APP_NAME_LIST text strings are returned by consecutive calls of the form:

```
GetSymbolicNames(-1, buffer, buflen, NEXT_ITEM);
```

When APP_NAME_LIST strings in the server's configuration file are exhausted, `GetSymbolicNames` returns INOT_ERR(101). There is no way, without repeated calls to `GetSymbolicNames`, to know in advance how many APP_NAME_LIST entries exist. Calling `GetSymbolicNames` with the FIRST_ITEM mode always attempts to return the first item in the list, resetting the client position to the beginning of the list.

### 13.1.6.3 Application Name List Example

The following sample function displays all APP_NAME_LIST strings that exist in a server's configuration file:

```c
void DisplayAppNameList(void)
{
    TEXT buffer[MAX_NAME + 1];
    NINT count = 0;
    if (GetSymbolicNames(-1, buffer, sizeof(buffer), FIRST_ITEM))
        printf("No APP_NAME_LIST present.\n");
    else do {
        printf("%d: %s\n", (count+1), buffer);
        count++;
    } while (!GetSymbolicNames(-1, buffer, sizeof(buffer), NEXT_ITEM));
}
```

### 13.1.7 ctFeatCHANNELS Now Default for All Servers

The ctFeatCHANNELS feature permits more flexible usage of multiple I/O channels. Without this feature, the ctDUPCHANEL file mode bit enables a file to use NUMCHANEL simultaneous I/O channels, where NUMCHANEL is set at compile time, and has traditionally been set to two. For superfile hosts with ctDUPCHANEL, $2 \times$ NUMCHANEL I/O channels are established. Previously the default was not to turn on ctFeatCHANNELS, but now the default has changed to turn on ctFeatCHANNELS.
Adding ctFeatCHANNELS at compile time enables two new server configuration keywords:

```plaintext
SET_FILE_CHANNELS <file name>#<nbr of I/O channels>
DEFAULT_CHANNELS <nbr of I/O channels>
```

- **SET_FILE_CHANNELS**: Permits the number of I/O channels to be explicitly set for the named file regardless of whether or not the file mode, at open, includes ctDUPCHANNEL. A value of one for the number of I/O channels effectively disables ctDUPCHANNEL for the file. A value greater than one turns on DUPCHANNEL and determines the number of I/O channels used. The number of I/O channels is not limited by the compile time NUMCHANNEL value. You may have as many SET_FILE_CHANNELS entries as needed.

- **DEFAULT_CHANNELS**: Changes the number of I/O channels assigned to a file with ctDUPCHANNEL in its file mode at open, unless the file is in the SET_FILE_CHANNELS list. The default number of channels is not limited by the NUMCHANNEL value.

**Note**: When ctFeatCHANNELS is enabled, multiple I/O channels are disabled for newly created files. The multiple I/O channels take affect only on an open file call. Also, depending on default number of I/O channels, a superfile host not in the SET_FILE_CHANNELS will use no more than 2 * NUMCHANNEL I/O channels.

### 13.2 C-TREESQL SERVER

This section discusses other significant features introduced in the c-treeSQL Server V8.14.

#### 13.2.1 Support SQL User-Defined Scalar Functions

Scalar functions are an integral part of the support provided by a DBMS for query expressions. A DBMS typically provides several built-in scalar functions that transform data in different ways. Sometimes, however, there is a need for a custom-transformation of the data—a transformation that is not performed by any of the functions provided by the DBMS. This problem is solved by the concept of a user-defined scalar function (UDF): a scalar function that is defined by the user. A DBMS that supports UDFs allows the user to define his/her own functions that transform data in some custom manner.

Support for User-Defined Scalar Functions has now been introduced in the c-treeSQL Server. User-Defined Scalar Functions are an extension to the existing built-in scalar functions and return a single value each time one is invoked. These functions can be used in queries in the same way that system defined scalar functions are used.

The user defines functions by creating Java Stored Functions, modules written in Java that are similar to the ones written for stored procedures and triggers. The Java code snippet contained in the User-Defined Scalar Function definition is processed by c-treeSQL into a Java class definition and stored in the database in text and compiled form. User-Defined Scalar Functions can be created, executed and dropped using ISQL, ODBC and JDBC.
13.2.1.1 Creating a User-Defined Scalar Function

Use the **CREATE FUNCTION** command to create User-Defined Scalar Functions. Any user with resource privilege can create User-Defined Scalar Functions. The syntax for the **CREATE FUNCTION** statement is:

```sql
CREATE FUNCTION [ owner_name.]function_name
 ( [parameter_decl , ...] )
RETURNS (data_type)
 [ IMPORT java_import_clause ]
BEGIN
java_snippet
END
```

**parameter_decl :: [ IN ] parameter_name data_type**

**Note:** When creating stored procedures, the keywords **BEGIN**, **RETURNS**, **IMPORT** and **END** must start at the first column of the line. The function name is limited to a size of 32 characters.

The example below creates a User-Defined Scalar Function named 'str_cat' that takes two input arguments and returns the concatenated string.

```sql
ISQL> CREATE FUNCTION str_cat(IN org_string VARCHAR(20), IN string_to_concat VARCHAR(20))
RETURNS VARCHAR(40)
IMPORT
import java.math.*;
BEGIN
String new_str = org_string + string_to_concat ;
return new_str;
END
```

13.2.1.2 Invoking a User-Defined Scalar Function

Invoke User-Defined Scalar Functions in exactly the same manner as built-in scalar functions. User-Defined Scalar Functions can be used in the SELECT list or in the WHERE clause. They can be used as parameters of other scalar functions or in any expression. The parameter passed to a User-Defined Scalar Function can be a literal, field reference, or any expression.

**Example - With Constants**

```sql
ISQL> SELECT str_cat('abcd','efgh') FROM syscalctable;
```

**STR_CAT(ABCD,EFGH)-----------------------------**

abcdefgh
1 records selected

**Example - With Constants and Column References**

```sql
ISQL> SELECT empfname, str_cat(empfname,emplname) FROM emp WHERE str_cat('mary','john') = 'maryjohn';
```

**EMPFNAME STR_CAT(EMPFNAME,EMPLNAME)-----------------------------**

Example - With parameter reference (ODBC/JDBC)

```
SELECT str_cat(?,?) FROM emp
SELECT * FROM emp WHERE str_cat(?,?) = 'MaryJohn'
```

### 13.2.1.3 Deleting a User-Defined Scalar Function

The **DROP FUNCTION** deletes a User Defined Scalar Function. The syntax for the function is shown below:

```
DROP FUNCTION function_name
```

The example below shows how to use the **DROP FUNCTION**.

```
ISQL> DROP FUNCTION str_cat;
```

### 13.2.1.4 User-Defined Scalar Function Security

Users issuing the **CREATE FUNCTION** statement must have the DBA privilege or RESOURCE privilege. The owner or users with the DBA privilege can execute or drop any User-Defined Scalar Function and may grant the EXECUTE privilege to other users. When a user with EXECUTE privilege executes a User-Defined Scalar Function, the User-Defined Scalar Function owner's privileges (rather than the user's privileges) are checked for the objects that the User-Defined Scalar Function accesses. This enables a user to execute a User-Defined Scalar Function successfully even when they do not have privileges to directly access the objects that are accessed by the User Defined Scalar Function, so long as they have EXECUTE privilege on the User Defined Scalar Function.

### 13.2.2 c-treeSQL Conformance with Java 2 Platform

c-treeSQL has been enhanced for conformance with JDK 1.4. However, only JDK 1.3 is supported on AIX platforms.

### 13.2.3 Run-Time Settings for Java Stored Procedure Support

The following table lists the Java related Runtime Settings that are needed for the c-treeSQL Server with the Java stored procedure support. This is needed for initializing the JVM within the c-treeSQL Server. These settings are not required for a c-treeSQL Server built without Java Stored Procedure support.

**Table 13-1: Java Related Runtime Settings for c-treeSQL Server with JSP**

<table>
<thead>
<tr>
<th>Environment Variable Name</th>
<th>Environment Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDKHOME</td>
<td>DRIVE:\PATH_TO_JAVA\</td>
</tr>
<tr>
<td>CTREEHOME</td>
<td>DRIVE:\PATH_TO_CTREE\</td>
</tr>
</tbody>
</table>
13.2.4 Case-Sensitivity Issues

Currently, sorting of fetched rows is performed as a case sensitive operation. Some applications may require sorting that should, instead, be case insensitive with NULL values appearing first in the sort. For instance the values a, Z, B, g, NULL should be sorted as NULL, a, B, g, Z.

FairCom has implemented a new feature such that a database can be implied to be case sensitive (current default) or case insensitive. When a database is enabled as case insensitive, the sorting, comparisons and identifier will be considered case insensitive.

This feature is activated with the c-treeSQL Server configuration keyword

\texttt{SQL\_OPTION DB\_CASE\_INSENSITIVE}

This should be done before creating the template database. Once the template database is created, ALL the databases will have the same settings of the template. A future release of c-treeSQL will correct this template limitation.

When using case insensitive searches, all search conditions, sorting and grouping is done in a case insensitive manner. Metadata and character indexes are also considered to be case insensitive. Case insensitivity is enabled on a database by database basis.

13.2.5 Set Environment Variable for Server Process

The c-tree Server can now set environment variables for the server process on startup. This functionality, implemented via the new keyword \texttt{SETENV}, is primarily directed at the c-treeSQL Server Java Edition. For example, the c-treeSQL Server Java Edition looks at the CLASSPATH to find required classes for proper operation of the server. When the c-treeSQL Server Java Edition is running as a Windows 32 bit versions Service, the Server is launched automatically by the SCM (Service Control Manager) when the computer starts up. Setting the CLASSPATH via a keyword eliminates the manual operation of setting the environment variables. The syntax is as follows:

\texttt{SETENV <var>=<value>}

Where \texttt{var} is the environment variable and \texttt{value} is the setting. For example:

\texttt{SETENV CLASSPATH=\%CLASSPATH\%;C:\FairCom\ctreeJQL\classes}

The input line length for the server configuration file was increased from 255 to 1024 to allow for very long configuration values.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Environment Variable Name} & \textbf{Environment Variable Value} \\
\hline
CLASSPATH & \%JDKHOME%\jre\lib\rt.jar; \\
 & \%CTREEHOME%\ctreeSDK\ctreeAPI\bin.jql\classes \\
JVM\_LIB & \%JDKHOME%\jre\bin\server\jvm.dll \\
JAVA\_COMPILER & \%JDKHome%\bin\javac.exe \\
\hline
\end{tabular}
\caption{Java Related Runtime Settings for c-treeSQL Server with JSP}
\end{table}
13.3 C-TREEDB

This section discusses other significant features introduced in the c-treeDB API V8.14.

13.3.1 Create c-treeDB Databases Accessible to c-treeSQL

Enabling SQL access to databases created using the c-treeDB API previously required using the SQL import utility, ctsqlimp, to add references to the c-treeSQL system tables for the tables and indexes in the c-treeDB database. The c-treeDB API now supports creating databases for which changes to table and index definitions are automatically applied to the c-treeSQL system tables.

This support is enabled by using a new c-treeDB session logon mode, CTSESSION_SQL. Calling `ctdbAllocSession(CTSESSION_SQL)` creates a session in which table and index definition changes performed on a database are automatically applied to the c-treeSQL system tables. This allows changes made using the c-treeDB interface to be reflected at the c-treeSQL level without having to import the table. Tables that are added and deleted are also reflected in the c-treeSQL system tables.

13.3.1.1 Limitations

Session mode is still experimental at this time. The following limitations exist:

1. Problems with table structures will prevent update of SQL system tables.
2. Support of alter table is currently not implemented (i.e., tables altered at the c-treeDB level are not reflected in the SQL system table).
3. No check is performed to verify that there is a c-treeSQL Server available.

13.3.2 Detect c-tree Operational Model Used By c-treeDB

The c-treeDB function `ctdbGetLibType` provides the ability to detect c-tree Plus operational model used when compiling a c-treeDB C library. The function prototype is as follows:

```c
CTLIB_TYPE ctDECL ctdbGetLibType(CTHANDLE Handle);
```

`ctdbGetLibType` returns a value of type `CTLIB_TYPE` which has one of the following mutually-exclusive operational model bits set:

- `CTLIB_SINGLE` single user
- `CTLIB_MUSER` multi user
- `CTLIB_CLIENT` client
- `CTLIB_LOCLIB` loclib
- `CTLIB_SERVER` server side library

The return value has the following bits set if the library supports optional features such as multi-threading and transaction processing:

- `CTLIB_THREA` multi thread
- `CTLIB_TRAN` transaction processing
13.3.3 Test for Presence of $DELFLD$ Field in a Table

FairCom added a function to the c-treeDB C API to test for the presence of the internal delete flag ($DELFLD$) field in a table. $DELFLD$ is a hidden field that c-treeDB automatically creates as the first field in the record buffer. $DELFLD$ is defined as a 4-byte CT_ARRAY field (to ensure proper alignment of subsequent fields) and its only purpose is to keep a place at the beginning of the record to hold the c-tree delete record flag byte when a record is deleted. This field is handled internally by c-treeDB and should never be modified by the user.

The function prototype for the `ctdbHasDelField` function is as follows:

```c
CTBOOL ctdbHasDelField(CTHANDLE Handle);
```

`ctdbHasDelField` receives a table handle as parameter and returns YES if the table was created with a $DELFLD$ field. `ctdbHasDelField` returns NO if the table was not created with a $DELFLD$ field.

An equivalent method `HasDelField` was added to CTTable class in the c-treeDB C++ API to reflect the changes made to the c-treeDB C API. The new method syntax is:

```c
CTBOOL CTTable::HasDelField();
```

`HasDelField` returns YES if a table was created with a $DELFLD$ field, or NO if a table was not created with a $DELFLD$ field.

13.3.4 c-treeDB Support for c-tree Features

The c-treeDB C and C++ APIs have been enhanced to support c-tree features available through the low-level and ISAM APIs. The following sections discuss c-treeDB support for these features.

13.3.4.1 File Mirroring

The FairCom mirroring feature makes it possible to store important files on different drive volumes, partitions or physical drives. If the primary storage location is damaged or lost, the mirroring logic will automatically detect the situation and switch to the secondary or "mirrored" storage area.

The mirrored file is easily specified by appending a vertical bar (|) after the table name followed by the table mirror name. For example, to mirror a table named "customer" to "mirror", specify the table name as "customer|mirror". If no path is specified for the mirrored table, both tables will be located in the same directory. If the primary table and the mirrored table are to be located in different directories, then the path names must be specified the same way as the table names: "primary-path|mirrored-path".

13.3.4.1.1 Creating Mirrored Files using the c-treeDB API

To create a mirrored table, use the `ctdbCreateTable` function passing as the table name a proper mirrored naming convention:

```c
if (ctdbCreateTable(hTable, "customer|mirror", CTCREATE_TRNLOG) )
```
printf("Create table failed\n");

If you need to specify different locations for the mirrored tables, use \texttt{ctdbSetTablePath} to specify the mirrored paths:

\begin{verbatim}
if (ctdbSetTablePath(hTable, "primary_path|mirrored_path"))
    printf("ctdbSetTablePath failed\n");
\end{verbatim}

13.3.4.1.2 Opening Mirrored Files using the c-treeDB API

To open an existing mirrored table, use \texttt{ctdbOpenTable} specifying the proper mirror naming convention:

\begin{verbatim}
if (ctdbOpenTable(hTable, CTOPEN_NORMAL, "customer|mirror"))
    printf("ctdbOpenTable failed\n");
\end{verbatim}

The same principle applies when opening a table using \texttt{ctdbOpenTableByUID}.

13.3.4.1.3 Adding Mirrored Files to a c-treeDB Database

A mirrored table can be added to a c-treeDB database dictionary by calling \texttt{ctdbAddTable} function and specifying the table name and path using the appropriate mirror naming convention:

\begin{verbatim}
if (ctdbAddTable(hDatabase, "customer|mirror", "primary_path|mirror_path"))
    printf("ctdbAddTable failed\n");
\end{verbatim}

If a table is created originally without mirroring, it can subsequently be mirrored as follows:

1. Copy the original table to the mirror location.
2. Change the \texttt{ctdbOpenTable} function to use the mirror naming convention.

13.3.4.1.4 Suspending Mirroring

Under c-tree Server operation, all mirroring can be suspended by adding the following entry to the server configuration file (ctsrvr.cfg):

\begin{verbatim}
MIRRORS NO
\end{verbatim}

This may be useful when the mirroring hardware is not operational and the use of the primary data is necessary. By default, read and write operations on mirrored tables will continue without returning an error if either one of the files fail, but the other succeeds. When this happens, the failed file is shut down and subsequent I/O operations continue only with the remaining "good" file. If mirroring is used in the client/server model, the \texttt{SystemMonitor} function receives an event when one of the files succeed and the other fails.

13.3.4.1.5 Mirroring Limitations

The c-treeDB alter table function will not operate on mirrored tables. If \texttt{ctdbAlterTable} is called for a mirrored table, nothing is done and it returns error \texttt{CTDBRET_NOTSUPPORTED}. 
Mirroring is supplied for c-tree Server and single-user operations. It applies to all c-tree Plus file modes including transaction processing. Once a table is created and opened with mirroring, all subsequent file opens must be mirrored, except when the table is open exclusive.

Please refer to the c-tree Plus Programmer's Reference Guide for more detailed information on mirroring support.

### 13.3.4.2 Keep Locks on Transaction Termination

When a transaction is terminated by calling either the `ctdbCommit` or `ctdbAbort` function, all locks are automatically freed, including the locks acquired outside the transaction. An application might want to control which locks are kept when the transaction ends. To support this ability, c-treeDB introduced a function to control how locks are handled when a transaction terminates.

`ctdbSetKeepLock` establishes the mode by which locks will be handled when the transaction terminates. The function prototype is as follows:

```c
CTDBRET ctdbSetKeepLock(CTHANDLE Handle, CTKEEP_MODE mode);
```

The function receives a session `Handle` and a `mode` specifying how locks will be handled. `ctdbSetKeepLock` returns `CTDBRET_OK` on success. The valid `mode` values are:

- **CTKEEP_FREE**: Release all locks acquired before and during a transaction. This is the default mode when a session handle is allocated.
- **CTKEEP_LOCK**: Keep all locks acquired before and during a transaction.
- **CTKEEP_OUT**: Release only locks obtained within a transaction and/or locks on records updated within a transaction.
- **CTKEEP_OUTALL**: Unconditionally keep all locks acquired before a transaction began. Free locks obtained within the transaction.

`ctdbGetKeepLock` returns the current keep lock status, which is one of the modes listed above. The function prototype is as follows:

```c
CTKEEP_MODE ctdbGetKeepLock(CTHANDLE Handle);
```

FairCom added two methods to the c-treeDB C++ CTBase class to reflect the functions added to the c-treeDB C API. The usage of the functions is the same as for the c-treeDB C versions of the functions. Their function prototypes are as follows:

```cpp
void CTBase::SetKeepLock(CTKEEP_MODE mode);
CTKEEP_MODE CTBase::GetKeepLock();
```

### 13.3.4.3 c-treeDB C++ Support for Setting Key Type

FairCom added a method to the CTIndex class to permit the setting of the index key type. The c-treeDB C API already supports a `ctdbSetIndexKeyType` function that permits users from setting the index key type, but this functionality was not reflected in the c-treeDB C++ API.
The new method syntax is:

```c
void CTIndex::SetKeyType(CTDBKEY keytype);
```

The valid `keytype` values are:

- **CTINDEX_FIXED**: Fixed length key
- **CTINDEX_LEADING**: Leading character compression
- **CTINDEX_PADDING**: Padding compression
- **CTINDEX_LEADPAD**: Leading and padding compression

`SetKeyType` does not return a value but will throw an exception if the index key type cannot be set.

### 13.3.4.4 Support Creating Multiple Indexes in One Physical Index File

Previously, the c-treeDB API had limited support for specifying index file names. When a table was created using the `ctdbCreateTable` function, all indices were created inside one index file that shares the table name. After the table was created, each added index was placed in a separated file and this file name was the same as the index name. Please note that the index name is the logical name that identifies the index and the index file name is the name of the index file on disk.

#### 13.3.4.4.1 New c-treeDB C API Functions

FairCom added two functions, `ctdbSetIndexPathName` and `ctdbGetIndexPathName`, to the c-treeDB C API to eliminate this limitation.

`ctdbSetIndexPathName` specifies a path and a file name for the index file. Its function prototype is as follows:

```c
CTDBRET ctdbSetIndexPathName(CTHANDLE Handle, pTEXT path, pTEXT filename);
```

*Handle* is an index handle returned by `ctdbAddIndex` or `ctdbGetIndex` calls. Parameter *path* specifies the directory location of the index file. A NULL value for *path* indicates that the index file is to be located in the same directory as the data file. Parameter *filename* specifies the name of the index file. If *filename* is NULL, this index is to be a member of the previous index file. If the *filename* of all index files are NULL, then all indices will be placed in one physical index file located in the same directory as the data file, and the index file name is the same as the data file name, with the current index file extension. An application can change the current index file extension by calling `ctdbSetIndexExtension`.

`ctdbGetIndexPathName` retrieves the current index file name associated with an index. Its function prototype is as follows:

```c
pTEXT ctdbGetIndexPathName(CTHANDLE Handle);
```

`ctdbGetIndexPathName` may return NULL indicating that the index is a member of an index superfile.
13.3.4.4.2 New c-treeDB C++ API Methods

FairCom added two methods, SetIndexFileName and GetIndexFileName, to the c-treeDB C++ API CTTable class to reflect the new c-treeDB C API set and get index file name functions. The c-treeDB C++ CTIndex class was also modified to include the same two methods above, allowing its functionality to be accessed from both CTTable and CTIndex classes.

Use the following methods to set the index file path and filename:

```cpp
void CTTable::SetIndexFileName(NINT IndexNumber, const CTString* path,
                                const CTString* filename);
void CTTable::SetIndexFileName(const CTString& IndexName, const CTString* path,
                                const CTString* filename);
```

If filename is NULL, the index file name is cleared. These two overloaded members of CTTable class allow the identification of the index by number or by name.

Use the following methods to retrieve the index file path and filename:

```cpp
CTString GetIndexFilename(NINT IndexNumber);
CTString GetIndexFilename(const CTString& IndexName);
```

These overloaded methods identify the index by number or by name.

Two methods were also added to the CTIndex class.

To set the index file path and filename, use:

```cpp
void SetIndexFilename(const CTString* path, const CTString* name);
```

If filename is NULL, the index file name is cleared.

To retrieve the index file name, use:

```cpp
virtual CTString GetIndexFilename();
```

13.3.4.4.3 Example: Creating Multiple Indices in One Physical Index File

Below is an example that demonstrates creating tables with three indices, the first two indices in the file myindex1 and the third index in the file myindex2:

```cpp
CTHANDLE hTable = ctdbAllocTable(hDatabase);
CTHANDLE hIndex;

ctdbAddField(hTable, "name", CT_FSTRING, 20);
ctdbAddField(hTable, "age", CT_INT2);
hIndex = ctdbAddIndex(hTable, "index1", CTINDEX_FIXED, NO, NO);
ctdbSetIndexFilename(hIndex, NULL, "myindex1");
ctdbAddSegmentByName(hTable, 0, "name", CTSEG_SCHSEG);
ctdbAddIndex(hTable, "index2", CTINDEX_FIXED, NO, NO);
ctdbAddSegmentByName(hTable, 1, "age", CTSEG_SCHSEG);
hIndex = ctdbAddIndex(hTable, "index3", CTINDEX_FIXED, NO, NO);
ctdbSetIndexFilename(hIndex, NULL, "myindex2");
ctdbAddSegmentByName(hTable, 2, "name", CTSEG_SCHSEG);
ctdbAddSegmentByName(hTable, 2, "age", CTSEG_SCHSEG);
if (ctdbCreateTable(hTable, "mytable", CTCREATE_NORMAL) != CTDRET_OK)
    printf("ctdbCreateTable failed with code %d\n", ctdbGetError(hTable));
```
13.3.5 c-treeDB Support for c-tree API Functions

13.3.5.1 EstimateKeySpan

A new function, ctdbEstimateSpan, can be used to estimate an approximate number of records between two key target values. The syntax for the new function is:

LONG ctdbEstimateSpan(CTHANDLE Handle, pVOID key1, pVOID key2);

Handle is a record handle and key1 and key2 are two key target values used to obtain the estimated number of records. If ctdbEstimateSpan returns 0, use the ctdbGetError function to retrieve the error code. If ctdbEstimateSpan returns 0 and ctdbGetError returns CTDBRET_OK, then there are no records between the two key values supplied.

The estimation is based on the record handle current index. The current index may be changed by calling ctdbSetDefaultIndex. The table must have at least one index to be able to use this function.

ctdbEstimateSpan, which is based on the c-tree low level function EstimateKeySpan, does not traverse the index to compute the values. Instead, it makes about ten calls to the c-tree low level function KeyAtPercentile to determine the relative location of the target values.

The key target values used by ctdbEstimateSpan can be created using ctdbBuildTargetKey.

A new method EstimateSpan was added to the CTRecord class to bring the c-treeDB C++ code in-line with the c-treeDB C changes. The EstimateSpan method has the following syntax:

LONG EstimateSpan(pVOID key1, pVOID key2);

EstimateSpan is the equivalent of the ctdbEstimateSpan function.

13.3.5.1.1 c-treeDB C API Example

Suppose that a student table has two fields (Name, of type CT_FSTRING and Age, of type CT_INT2) and one index on field Age. The sample function estimates the number of students with ages between 10 and 12:

LONG EstimateStudents(CTHANDLE hRecord)
{
    TEXT key1[32];
    TEXT key2[32];

    /* build the first key for age = 10 years */
    ctdbClearRecord(hRecord);
    ctdbSetFieldAsSigned(hRecord, 1, 10);
    if (ctdbBuildTargetKey(hRecord, CTFIND_EQ, key1, 32))
        return 0;
    /* build the second key for age = 12 years */
    ctdbClearRecord(hRecord);
    ctdbSetFieldAsSigned(hRecord, 1, 12);
    if (ctdbBuildTargetKey(hRecord, CTFIND_EQ, key2, 32))
        return 0;
/* estimate the number of students */
    return ctdbEstimateSpan(hRecord, key1, key2);
}

13.3.5.2 KeyAtPercentile

A new function, ctdbRecordAtPercentile, was added that allows one to read the record
located at, approximately, the given percentile value. The syntax for the this function is:

CTDBRET ctdbRecordAtPercentile(CTHANDLE Handle, NINT percent);

Handle is a record handle and percent indicates the percentile value. The valid values for percent
are from 0 to 100, indicating 0% to 100%. ctdbRecordAtPercentile returns
CTDBRET_OK on success.

The record is located using the record handle current index. You may select a new current
index by calling ctdbSetDefaultIndex function. The table must have at least one index to be
able to use this function.

The record returned is an approximate location indicated by the percentual value passed to ctdbRecordAtPercentile.

ctdbRecordAtPercentile, which is based on the c-tree low level function KeyAtPercentile, is
very efficient since it does not traverse all of the key values in order to determine the record
located at the specified percentile. However, ctdbRecordAtPercentile is only an approxima-
tion since it assumes that key values are uniformly distributed among all of the b-tree leaf
nodes.

cmdbRecordAtPercentile may be used to support scroll bar positioning, found in many GUI
windowing environments, in the cases when the position must be maintained in key sequential
order.

The method AtPercentile was added to CTRecord class to bring the CTDB C++ code in line
with the CTDB C new functionality provided by ctdbRecordAtPercentile.

The AtPercentile method has the following syntax:

void CTRecord::AtPercentile(NINT percent);

AtPercentile is the equivalent of the ctdbRecordAtPercentile function. If a record cannot be
located, a CTException exception object is thrown.

13.3.5.3 SetNodeName

c-treeDB client applications can associate names with c-treeDB client connections using a new
c-treeDB API function.

The c-treeDB C API function ctdbSetCurrentNodeName function prototype is as follows:

CTDBRET ctdbSetCurrentNodeName(CTHANDLE Handle, pTEXT NodeName);

ctdbSetCurrentNodeName sets a client-side node name. Handle is a session handle and
NodeName is an string specifying the node name. The specified node name appears in the
ctadmn utility under the "List clients" option. \texttt{ctdbSetCurrentNodeName} may only be called after a successful logon.

The c-treeDB C++ API method \texttt{CTSession::SetCurrentNodeName} function prototype is as follows:

\begin{verbatim}
void CTSession::SetCurrentNodeName(const CTString& NodeName);
\end{verbatim}

No errors are returned, but \texttt{CTException} is thrown if an error is detected setting the node name.

### 13.3.5.3.1 c-treeDB C API Example

```c
/* set the c-tree Server node name for this workstation */
if (ctdbSetCurrentNodeName(AnyHandle, "This is my node") != CTDBRET_OK)
   printf("ctdbSetCurrentNodeName() failed\n");
```

### 13.3.6 Remove Table Function

The \texttt{ctdbDeleteTable} function did not work if a table was not member of a database. \texttt{ctdbDeleteTable} takes as parameters a database handle, a table name and optionally a table password. The \texttt{ctdbRemoveTable} function was added to enable any table to be deleted, including tables that are not members of a database. A corresponding new method \texttt{Remove} was added to the \texttt{CTTable} class.

The syntax for \texttt{ctdbRemoveTable} is:

\begin{verbatim}
CTDBRET ctdbDECL ctdbRemoveTable(CTHANDLE Handle);
\end{verbatim}

The syntax for \texttt{Remove} is:

\begin{verbatim}
void CTTable::Remove();
\end{verbatim}

Handle is a table handle. The \texttt{ctdbRemoveTable} function deletes a c-tree data file and the associated index files from disk. If the table was opened under a database handle, the table is closed and \texttt{ctdbDeleteTable} (or \texttt{CTDatabase::DeleteTable} in the case of \texttt{CTTable::Remove}) is called. If the handle is not active, the table is opened exclusive and then deleted. If the table handle is not active, you must set the path, file extension and password to the handle before calling \texttt{ctdbRemoveTable}.

### 13.3.7 User Defined Path Prefixes

A new feature was requested to allow the ability to define paths prefixes. Previous versions of c-tree, while allowing some degree of definition, still required database files to be path relative to the current directory. This problem was addressed by allowing the user to define the path, and this path name is now associated with the session handle, without persisting on disk.

The following new c-treeDB functions were added:

- To set a new path prefix. Set \texttt{pathPrefix} to NULL to remove a previous path
  \begin{verbatim}
  CTDBRET ctdbSetPathPrefix(CTHANDLE hSession, pTEXT pathPrefix);
  \end{verbatim}
To get a path prefix - if no path prefix set, return NULL

```c
pTEXT ctdbGetPathPrefix(CTHANDLE hSession);
```

The following new methods were added to the CTSession class:

- To set a new path prefix
  ```c
  void CTSession::SetPathPrefix(const CTString& pathPrefix);
  ```
- To clear a previous path prefix
  ```c
  void CTSession::SetPathPrefix();
  ```
- To retrieve the current path prefix. (If a path prefix is set, `GetPathPrefix` returns YES and the path is set to `pathPrefix`. If no path prefix is set, `GetPathPrefix` return NO and `pathPrefix` is cleared.)
  ```c
  CTBOOL CTSession::GetPathPrefix(CTString& pathPrefix);
  ```

A path prefix can be set anytime after the session handle is allocated. If a path prefix is set before a session logon, the new path prefix will affect the location of the session dictionary file. If a path prefix is set after a session logon, but before a database connect, then the path prefix affects only the database dictionary and any tables that are manipulated during that session.

A path prefix can be removed at any time by setting a NULL value for the path prefix. You can use `ctdbGetPathPrefix` to check if a path prefix is set or not. If ctdbGetPathPrefix returns NULL, then no path prefix is set.

**User Defined Path Prefixes**

### 13.4 **DBEXPRESS DRIVER**

dbExpress is a cross-platform, database-independent and extensible interface that provides a set of methods for dynamic SQL processing. dbExpress represents the new generation of data-access methods for Kylix™, Delphi™ and C++Builder™. dbExpress drives are lightweight database drivers that provide fast access to SQL database servers. For each supported database, dbExpress provides a driver that adapts the server-specific software to a set of uniform dbExpress interfaces.

This release introduces the c-treeDBX driver, the implementation of a dbExpress driver for FairCom's SQL database servers, the c-treeSQL Server, and the c-treeSQL Server Java Edition. The c-treeSQL Server Java Edition is the Java enabled implementation of the SQL technology from FairCom, and includes support for JDBC, and stored procedures and triggers in Java.

### 13.5 **C-TREEVCL/CLX**

This section discusses significant features available in c-treeVCL/CLX V8.14.
13.5.1 Additional Event Handlers for TCTable

FairCom added event handlers to TCTable to handle exceptions and allow the application to take actions in the following cases:

13.5.1.1 When Record Deletion Fails

FairCom added an OnDeleteError event to TCTable to handle exceptions that occur when an attempt to delete a record fails. When OnDeleteError is first invoked, the Action parameter is always set to daFail. If the error handler corrects the error condition that caused the handler to be invoked, set the Action parameter to daRetry before exiting the handler. When Action is daRetry, the delete operation is tried again. If the error condition cannot be corrected, you can suppress the display of an error message by setting Action to daAbort instead of daFail.

13.5.1.2 When Record Edit Fails

FairCom added an OnEditError event to TCTable to handle exceptions that occur when an attempt to edit a record fails. When the OnEditError event handler is first invoked, the Action parameter is always set to daFail. If the error handler corrects the error condition that caused the handler to be invoked, set the Action parameter to daRetry before exiting the handler. When Action is daRetry, the edit operation is tried again. If the error condition cannot be corrected, you can suppress the display of an error message by setting the Action parameter to daAbort instead of daFail.

13.5.1.3 When Record Posting Fails

FairCom added an OnPostError event to TCTable to handle exceptions that occur when an attempt to post a record failed. When OnPostError is first invoked, the Action parameter is always set to daFail. If the error handler can correct the error condition that caused the handler to be invoked, set the Action parameter to daRetry before exiting the handler. When the Action parameter is set to daRetry, the post operation is tried again. If an error condition cannot be corrected, you can suppress the display of an error message by setting the Action parameter to daAbort instead of daFail.

13.5.1.4 When Inserting or Appending a New Record

FairCom added an OnNewRecord event to TCTable to take specific actions as an application inserts or appends a new record. OnNewRecord is called as part of the insert or append process. An application might use the OnNewRecord event to set initial values for a record or as a way of implementing cascading insertions in related datasets.

13.6 SUPPORT SETTING DATA AND INDEX FILE EXTENSIONS

The VCL component offers two new properties for the TCTable Object that allow the user to provide data and index file extension names. The properties are called DataExt and IndexExt.
These can be initialized at design time or run time. The following is run time programming syntax:

\[
\text{CtTable1->DataExt = ",tbd";} \\
\text{CtTable1->IndexExt = ",tbi";}
\]

13.7 CATCHING EXCEPTIONS SPECIFIC TO C-TREEVCL

The TCtError class now inherits from the EDatabaseError class. This change allows for applications to catch exceptions that are specific to c-treeVCL by using ECtError, or more general database exceptions by using the EDatabaseError class.

13.8 ADDITIONAL C-TREEVCL ENHANCEMENTS

c-treeVCL provides a direct interface to c-treeDB functions exported by c-tree client or standalone DLLs. In this particular case, c-treeVCL provides the function declaration in Object Pascal language used by Delphi.

The following enhancements were applied to the c-treeDB Pascal interface:

1. Changed the incorrect declaration of the `ctdbSetDefDateType` function to its correct form:

   ```pascal
   ctdbSetDefDateType:function(Handle:CTHANDLE; DType:CTDATE_TYPE):CTDBRET; cdecl;
   ```

2. Changed the declaration of the `ctdbCopyRecordBuffer` function to match the current declaration in c-treeDB:

   ```pascal
   ctdbCopyRecordBuffer:function(source:CTHANDLE; dest:CTHANDLE):CTDBRET; cdecl;
   ```

3. Updated the declaration of `ctdbMoveField` function by renaming the first parameter from `hTable` to `hField` to indicate that the first parameter should be a field handle instead of a table handle.

4. Added the declaration of `ctdbGetTableCount` function:

   ```pascal
   ctdbGetTableCount:function(hDatabase:CTHANDLE):NINT; cdecl;
   ```

None of the above functions are used by the c-treeVCL components, but are necessary for the completeness of the c-treeDB API declaration in Object Pascal.

13.9 C-TREESQL ODBC DRIVER

This section discusses significant features introduced in the c-treeSQL ODBC Driver V8.14.

13.9.1 c-treeSQL ODBC Driver for Unix

The c-treeSQL ODBC Drivers are now available on Unix systems, including all Linux distributions supported by FairCom. The c-treeSQL ODBC implementation for Unix uses a method called "direct link" where the ODBC client application links directly to FairCom ODBC libraries, making unnecessary the use of "middleware" ODBC management software to manage ODBC connections and data sources.
The Open Database Connectivity (ODBC) interface from Microsoft has emerged as the standard mechanism for client applications to access data from a variety of different data sources through a single interface.

To become accessible from ODBC client applications, database environments must provide a software driver on the client system were the application resides. The driver translates the standard ODBC function calls into calls the database server can process, and returns the resulting data to the application.

UNIX ODBC client applications that use the c-treeSQL ODBC direct link driver must include the following header files in any source modules that make calls to the ODBC API:

```
#include <sql.h>
#include <sqlext.h>
```

These files are found in the `ctreeSDK\ctreeESQL\include` directory

UNIX ODBC client applications must link the following libraries to resolve any calls to the ODBC API:

```
ctreeSDK\ctreeODBC\ctreeSQL_ODBC\lib\libodbc_c.a
ctreeSDK\ctreeESQL\lib\libctesql.a
```

### 13.9.2 Direct Link c-treeSQL ODBC Driver for Unix and Windows

Based on the Windows 32 bit versions API, a direct link ODBC driver for Windows systems is now supported. This gives users a choice of a normal ODBC driver where the connections and data sources for c-treeSQL Server are managed by the ODBC Manager, and a direct link driver where applications just link directly to FairCom ODBC libraries, making unnecessary the use of the ODBC Manager.

The direct link approach avoids the necessity of dealing with the ODBC Manager when installing ODBC applications using FairCom's ODBC driver, which may be advantageous on large scale systems or mass distributed products.

Windows ODBC client applications that use the c-treeSQL ODBC direct link driver must include the following header files in any source modules that make calls to the ODBC API:

```
#include <sql.h>
#include <sqlext.h>
```

These files are found in the `ctreeSDK\ctreeESQL\include` directory

Client applications must link the following libraries to resolve any calls to the ODBC API:

```
ctreeSDK\ctreeODBC\ctreeSQL_ODBC\lib\libodbc_c.lib
ctreeSDK\ctreeESQL\lib\libctesql.lib
```

### 13.9.3 Support Configurable c-treeSQL ODBC Driver Login Timeout

The c-treeSQL ODBC Driver now supports the SQL_ATTR_LOGIN_TIMEOUT connection attribute. This attribute sets a driver login timeout for the c-treeSQL ODBC Driver. An attempt
to connect to the c-treeSQL ODBC Driver will fail if it is not completed within the specified
time limit. The default value is 15 seconds. The attribute value can be changed by calling the
ODBC API function SQLSetConnectAttr.

13.10 C-TREESQL JDBC DRIVER
This section discusses significant features introduced in the c-treeSQL JDBC Driver V8.14.

13.10.1 New Architecture
With this release, the c-treeSQL architecture has been changed to improve the JDBC client and
JDBC server performance. The new architecture bypasses the JNI layer on the server-side, thus
resulting in a significant improvement in the performance of JDBC applications.

Because the new c-treeSQL JDBC driver architecture no longer has a JNI layer in the server,
there is no need to initialize the JVM in the server. This has been done without any additional
overhead in the JDBC Driver. The new JDBC driver can interact with a c-treeSQL Server built
in the Non-Java configuration, whereas previously the c-treeSQL Server Java Edition was
required for JDBC support.

13.10.2 JDBC Driver Bulk Fetch Support
The c-treeSQL JDBC driver has been enhanced for fetching the data in bulk from the server.
Now the driver can fetch 64K of data from the server in a single call. This reduces the number
of network calls significantly and improves performance.

13.10.3 Connection Pooling Support
The c-treeSQL JDBC Driver now supports connection pooling. Specifically, the DataSource,
PooledConnection and ConnectionPoolDataSource APIs are now supported. Because a num-
ber of third-party applications, including IBM's Websphere, use these APIs, this support allows
these applications to use the JDBC Driver as a data source.

13.10.4 JDBC 3.0 Specification Conformance
FairCom applied the following changes to the c-treeSQL JDBC Driver in order to conform to
the JDBC 3.0 specification.

13.10.4.1 ParameterMetaData Interface
ParameterMetaData is one of the required interfaces supported in the c-treeSQL JDBC Driver
in order to conform to the JDBC 3.0 specification. A new class, DharmaParameterMetaData,
implements this interface. Please refer to c-treeSQL JDBC Driver guide for further details on
this new class.
13.10.4.2 Modified Methods

The new methods added in DharmaDatabaseMetaData are listed in the JDBC Driver User guide. Only the methods that were modified in order to conform to the JDBC 3.0 specification are listed here.

- **getTables**: Returns descriptions of the tables that match the given catalog, schema, table name, and type criteria. The following new columns are returned at the end of the existing list of columns in the resultset: TYPE_CAT, TYPE_SCHEM, TYPE_NAME, SELFREFERENCING_COL_NAME, REFGENERATION.

- **getColumns**: Returns descriptions of the columns that match the given catalog, schema, table name, and column name criteria. The following new columns are returned at the end of the existing list of columns in the resultset: SCOPE_CATLOG, SCOPE_SCHEMA, SCOPE_TABLE, SOURCE_DATA_TYPE.

- **getUDTs**: Returns descriptions of the user-defined types that match the given catalog, schema, type name, and type criteria. In this case, it returns an empty resultset object with column information. The following new column is returned at the end of the existing list of columns in the resultset: BASE_TYPE.

- **getSchemas**: Now returns the catalog for each schema as well as the schemata. The following new column is returned at the end of the existing list of columns in the resultset: TABLECATALOG.

13.11 C-TREE PLUS

In addition to the major features already discussed, this section discusses a significant feature introduced in the c-tree Plus API V8.14.

13.11.1 Support Next/Previous Operation After Record Add

A number of customers have asked for the ability to read the next or previous record after adding a record without needing to re-read the record just added. Prior to this enhancement, the current ISAM position was not set to the newly-added record. Therefore an INOT_ERR(101) error would result if you tried to read either the next of previous record. Now c-tree sets the current ISAM position after a record is added so that the next or previous record can be read without re-reading the record just added.
Chapter 14

c-tree Utilities

14.1 C-TREE SERVER STATUS MONITORING UTILITY, CTSYSM

The c-tree Server status monitoring utility, ctsysm, facilitates the monitoring of error, warning, and informational messages logged to the server status log, CTSTATUS.FCS, by the c-tree Server. Using this utility, the c-tree Server status log can be monitored by an automated external process such as the Tivoli monitoring system from IBM.

Using a customizable configuration file, the ctsysm utility provides:

- A well-defined format for server messages to make them readable by automated systems
- Additional context as to whether the message is an error or purely an informational message
- The c-tree Server subsystem that is involved
- A recommended action
The following diagram shows a conceptual view of the operation of the ctsysm utility:

14.1.1 Using the ctsysm Utility

To use this utility, the c-tree Server must be configured to log messages to the system log. This can be achieved by adding the following keyword to the server's configuration file:

SYSLOG CTSTATUS

With this keyword in place, the server logs each entry in the CTSTATUS.FCS file to the system log files (SYSLOGDT.FCS and SYSLOGIX.FCS).

The utility can read the system log from the beginning each time it starts up, or it can save its current position and start again from that set position by specifying the -f command line option.

The following is the supported command line usage of the c-tree Server Status Monitoring Utility:

```bash
```

- `-s svn` c-tree Server name
- `-u uid` user name
-p upw   user password
-r rpt   repeat interval in seconds (e.g., "-r 1" is a
  one-second delay before checking for new messages)
-c cfg   config file
-f fil   save/restore state to file
-l log   status log name (SYSLOGDT.FCS)

Example:

The following command causes ctsysm to monitor the c-tree Server system log for the server
FAIRCOMS (-s FAIRCOMS) for status messages, checking for new messages every second (-
 r 1), saving the position of the last entry read to the file ctsysm.log (-f ctsysm.log).

ctsysm -s FAIRCOMS -u ADMIN -p ADMIN -r 1 -c ctsysm.cfg -f ctsysm.log

The ctsysm utility outputs messages in the following format:

odusstement> <code> <subsystem> <action> <text>

Sample output from the utility is shown below (each message is output as a single line but is
shown split into two lines here):

Thu Jul 29 16:44:04 2004 I0455 STUP NONE - User# 01
  Alternative server name... FAIRCOMS
Thu Jul 29 16:44:04 2004 I0492 STUP NONE - User# 01
  Compatibility bit maps: 00000000 00002000x
Thu Jul 29 16:44:04 2004 I0491 STUP NONE - User# 01
  Diagnostic bit maps: 00000000 00000000x
Thu Jul 29 16:44:04 2004 I0490 STUP NONE - User# 01
  64-bit File Address Support
Thu Jul 29 16:44:04 2004 I0489 STUP NONE - User# 01
  6 Byte Transaction Numbers
Thu Jul 29 16:44:04 2004 I0488 STUP NONE - User# 01
  NOWAIT usrsema enabled

Note: When the repeat (-r) option is used, the utility can be stopped by sending it a SIGINT
signal.

14.1.2 ctsysm Configuration File

The configuration file, ctsysm.cfg, contains server messages classified by the server sub-
systems that generate the log entries, the recommended actions, and details for each message.
The configuration file is simply a text file, an abbreviated copy of which is shown later in this
document. The configuration file can be used as-is, although some of the default recommended
actions are general and administrators may want to customize these.

The subsystem list consists of entries in this format:

<subsystem_keyword> <subsystem_description>

The action list consists of entries in this format:

<action_keyword> <action_description>

The messages are in the following format:

<code> <subsystem> <action> <num_lines> <text>
where `<code>` is a 5-character code beginning with a message type and followed by a 4-digit message number (for example, E0001). Possible message codes include:

- **F**: Fatal error
- **E**: Error
- **W**: Warning
- **I**: Information
- **U**: Unclassified

Because the configuration file is a text file separate from the server executable, it can easily be updated as new or application-specific information about specific messages becomes available, without having to update the server executable.

### 14.1.3 ctsysm Configuration File Sample

Below is a portion of a sample of the current configuration file (`ctreeSDK\ctreeAPI\ctree\source\ctsystm.cfg`).

```plaintext
[Version]
c-tree Server Status Log Monitor Configuration File V8.14.030729

[Subsystems]
CADM Connection administration
COMM Communication
DIAG Diagnostic information
DDMP Dynamic dump
FMNT File maintenance
.
.
.

[Actions]
CCFG The server found an unsupported or invalid option in the server configuration file. Check the server configuration file.

CFGW An option specified in the server configuration file might lead to undesired behavior. Review the message to understand the situation.

CKDS The operation failed due to insufficient disk space. Free up disk space to allow the operation to complete.
.
.
.

[Messages]
;==================================================
;Server configuration messages
;==================================================
```
14.2 **SUPERFILE DATA EXPORT UTILITY, CTSFEX**

The new *ctsfex* utility exports data from a superfile to another superfile or to individual files identical to the superfile members. This high-performance utility exports the data without index updates, transaction control, or file extension. After the data export is completed, the indices are rebuilt and transaction control and file extension properties are restored on the newly-created files. This utility also supports a multi-threaded version allowing each thread to process a superfile member. The syntax of this utility is shown below:

```
```

- `-f`: Create target as a superfile
- `-d`: Create individual target files in the specified directory
- `-c`: Create the target files without exporting data
- `-v`: Open the superfile and its members to validate a superfile
- `-V`: Used in conjunction with `-v`, this option validates records with a read
- `-r`: Force overwrite of target; otherwise the program fails if the target file exists
- `-o`: Force open of a corrupted file by using the ctOPENCRPT file mode.
- `-m`: Show the progress by displaying a record counter
-e: Enable data encryption on target files
-t: Set the maximum number of concurrent threads to be used
-n: Set the index node size to be used (standalone mode only)
-s: Specify the server name for connecting to the server
-u: Specify the user id for connecting to the server
-p: Specify the user password for connecting to the server
-ctscmp <args>: Invoke the functionality of ctscmp superfile compact utility.

14.3 ENHANCED SUPERFILE COMPACT UTILITY, CTSCMP AND CTSCMPI

The superfile compact utility, ctscmp, compacts and rebuilds a superfile and all of its members provided the required IFIL resources are present in the superfile data members. It creates a compacted version of the file in place, using a temporary file named CTREE.TMP as a destination and then renaming it.

FairCom applied the following enhancements to the superfile compact utility:

1. Added a command line option, -scanonly, that scans the superfile without compacting it. The scanonly option generates details from the header, extended header, IFIL, and DODA. It outputs the number of key values, active records, deleted records and resource records. Using this utility with the -scanonly option is useful for comparing superfiles before and after compacting them.

2. The utility now supports processing data and index member names having extensions other than .dat and .idx.

3. Added a '% complete' progress indicator to the ctscmp utility.

4. The ctscmp utility uses a standard error and warning message format. Error messages begin with the text 'Error:' and warning messages with the text 'Warning:', so errors and warnings can easily be located in the ctscmp output.

5. Detect the presence of a superfile member whose member number exceeds the last member number in the superfile host header and display a warning in this situation. Previously, ctscmp crashed in this situation due to an out of bounds array reference.

FairCom also introduced an ISAM-based version of the superfile compact utility named ctscmpi. This version of the superfile compact utility uses ISAM c-tree functions rather than low-level c-tree functions to add records to the new compacted superfile. The advantage of using the ISAM approach is that records are indexed as they are added, so the utility does not require a separate rebuild phase to repopulate the indexes as the ctscmp utility does. Using this technique can significantly reduce the time required to compact a superfile.

The syntax and details of the command-line arguments for the new features are described below.
ctscmp <filename> [-scanonly] [Y] [<sect>]
ctscmpi <filename> [-scanonly] [Y] [<sect>]

- <filename>: The name of the superfile to compact or scan
- -scanonly: Scan the superfile only; do not compact it.
- Y: An optional parameter; if specified, the utility compacts the file without prompting to confirm (otherwise you will be prompted).
- <sect>: An optional sector size of the resulting file (sect*128 = page size) to use for the new superfile. If <sect> is omitted, the new superfile is created using the same page size as the original superfile.

14.4 INSPECTING INDEX CONTENTS AND CHECKING INDEX INTEGRITY

Two new utilities have been added for dumping and verifying indices: ctdmpidx and ctvfyidx. These utilities are useful for viewing index header details, viewing nodes and key values, and verifying the integrity of an index. The syntax of these utilities is shown below:

ctdmpidx [-<page size>] <filename> <member #> [<rflg>]
ctvfyidx [-<page size>] <filename> <member #> [<rflg>]

The optional parameter page size equals sector size * 128 (third parameter in InitCtree). If page size is not entered, a default value of 16 will be used. filename specifies the index file targeted for analysis. The member # refers to the index member number. A physical index file can contain one or more indices. Each index has a member number (0, 1, 2, 3, etc.). For example, the sample index file custordr.idx provided with the FairCom ODBC Driver contains a total of two indices. Depending on whether you specify 0 or 1 you will be looking at either the order number index or the customer number index. rflg represents an optional recovery flag switch and is only applicable when compiled with TRANPROC. Any character will enable rflg, which will result in c-tree skipping automatic recovery.

14.4.1 Examples

Below is an example of launching ctdmpidx along with output showing the header and prompt for an index node address.

ctdmpidx -2048 custmast.idx 0

header info for file #0 [custmast.idx]:

<table>
<thead>
<tr>
<th>configuration version:</th>
<th>81x</th>
<th>index root ptr:</th>
<th>1000x</th>
</tr>
</thead>
<tbody>
<tr>
<td>node size:</td>
<td>2048</td>
<td>superfile last member#:</td>
<td>0</td>
</tr>
<tr>
<td>data record length:</td>
<td>0</td>
<td>extend file size:</td>
<td>0</td>
</tr>
<tr>
<td>max bytes per leaf node:</td>
<td>1348</td>
<td>max bytes per non-leaf:</td>
<td>2030</td>
</tr>
<tr>
<td>file mode:</td>
<td>16392</td>
<td>key length:</td>
<td>4</td>
</tr>
<tr>
<td>file type:</td>
<td>1</td>
<td># of additional members:</td>
<td>0</td>
</tr>
<tr>
<td>update flag:</td>
<td>0x</td>
<td>index member number:</td>
<td>0</td>
</tr>
<tr>
<td>key type:</td>
<td>0</td>
<td>file id number:</td>
<td>0</td>
</tr>
<tr>
<td>duplicate flag:</td>
<td>0</td>
<td>server id number:</td>
<td>0</td>
</tr>
</tbody>
</table>
# of active entries: 4
delete stack ptr: 0x
last byte used: 1800x
last physical byte: 1800x
permanent file mode flags: 0
leaf anchor ptr: 1000x
file flavor: 2
key padding byte: 20x
header record ptr: 0x

serial number: 0x
index high transaction #: 0
delete number: 0x
delete type: 0
serial number: 0x
delete type: 0
leaf anchor ptr: 1000x
superfile member #: 0
file alignment: 8
maximum name length: 255
resource header ptr: 800x

Enter node address in hex: 1000
NODE: 1000x

trans active: no
predecessor node: 0x
successor node: 0x

# of entries: 4
# of bytes: 32

leaf flag: 1 (0 => non-leaf / 1 => leaf)
member #: 0

<table>
<thead>
<tr>
<th>element</th>
<th>offset</th>
<th>key value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101fx</td>
<td>31303030</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>105ex</td>
<td>31303031</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>3</td>
<td>1099x</td>
<td>31303032</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 2</td>
</tr>
<tr>
<td>4</td>
<td>10d9x</td>
<td>31303033</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 0 0 3</td>
</tr>
</tbody>
</table>

Below is an example of launching ctyfyidx along with output showing the results of the index verification.

ctfyidyx -2048 custmast.idx 0

Index page scan finds entries=4  header=4
Index nodes per level of tree structure - [0: 1]
Internal Index Verify: SUCCESSFUL

### 14.5 Updating File IDs

A new utility program called ctfileid was created and is part of a standard distribution found in ctreeSDK\ctreeAPI\ctree\samples\special\utils. This utility provides a convenient and safe way to update the fileid. The file is opened exclusively, ensuring that the server does not have it open. Launch ctfiileid without any command parameters and the menu of options will be displayed. The syntax for this utility is shown below:


- -i: Also update indices related to data file
- -o: Force open of corrupted files (ctOPENCRPT)
• -q: Quiet (do not output to stdout)
• -n <size>: Set node size (stand-alone only)
• -s <svn>: c-tree Server name
• -u <uid>: User name
• -p <upw>: User password

14.6 MOVING FILES WITHOUT STOPPING THE C-TREE SERVER

The new ctmove utility can be used to move files while the c-tree Server is running. This is a useful tool to minimize downtime during a file recovery. The move operation is protected by opening the file. The syntax of this tool is shown below:

```
ctmove [-o] file1 file2 [-n size] [-s svn] [-u uid] [-p upw]
```

• -o: Force open of a corrupted file by using the ctOPENCRPT file mode.
• -n: Defines the node size to be used (Standalone only).
• -s: Specifies the server name for connecting to the server.
• -u: Specifies the user id for connecting to the server.
• -p: Specifies the user password for connecting to the server.

14.7 C-TREESQL MAINTENANCE UTILITY, CTSQLUTL

A new utility has been created as a general purpose program to perform maintenance on the c-treeSQL Server. At this time only the "rename column" (-rencol) command has been implemented.

When the ctsqlutl utility is launched with the -rencol command, the program browses through the internal SQL dictionaries and updates the records with the new column name.

The ctsqlutl utility syntax is as follows:

```
ctsqlutl [options] -rencol table_name column newcolumn
```

• table_name: Name of the table
• column: Current name of the column you are going to rename
• newcolumn: Name of the columns after renaming

The valid options are:
• -o: Owner of table
• -d: Database name (default: ctreeSQL)
• -s: c-treeSQL Server name (default: FAIRCOMS)
• -u: userid for logging to c-treeSQL Server
• -a: Password for authentication
• -h: Display usage help

### 14.8 REBUILD WITH DUPLICATE RECORD PURGE IN CLIENT/SERVER MODE

The IFIL rebuild utility, `ctrbldif`, previously only supported the `-purge` option in standalone mode. This option instructs the rebuild to mark records having illegal duplicate key values as deleted. The utility now supports this option in both standalone and client/server mode. The usage for `ctrbldif` in client/server mode is now:

```
ctrbldif DataFileName [-purge] [<UserId>] [<UserPassword>] [<ServerName>]
```

where `-purge` indicates that duplicate records should be purged.

### 14.9 C-TREESQL IMPORT UTILITY MODIFICATIONS

The following enhancements were made to the c-treeSQL import utility (`ctsqlimp`).

#### 14.9.1 Support for Precision and Scale Information

Previously, the `ctsqlimp` utility did not provide support for precision and scale information for numeric fields. The precision and scale field information maintained in the system columns dictionary is now updated during table import with `ctsqlimp`.

#### 14.9.2 Check for Invalid Characters in Field Names

The c-treeSQL import utility will check for the correct syntax of field names when importing tables into c-treeSQL Server.

The SQL standard dictates that SQL identifiers such as table, column and index names are of two types: conventional and delimited. The conventional identifiers must begin with a letter and contain only letters, digits or underscore. The delimited may contain other ASCII printable characters, but must be enclosed in double quotation marks.

The c-treeSQL import utility now checks if a field name complies with SQL syntax and display a warning message if the field names does not comply. The warning message informs the user that the column resulting after the field importation must be enclosed within double quotation marks (""") when used in SQL statements. Furthermore the use is prompted to skip the field in case the column is to be hidden since if does not comply with conventional SQL naming convention.

By adding a switch (`-k`) to the import utility, `ctsqlimp`, users can now skip importing fields that do not comply with the conventional identifier rules without being prompted every time an incompatible field is encountered.

`ctsqlimp` already has a "non-interactive" option (`-i`) which avoids prompting the user by having a list of pre-defined default answers. The actions taken by default are the most conservative
and safe. In case an unconventional field is encountered, the default action is to import the field and inform the user that in SQL statements, the resulting column name must be enclosed in double quotation marks.

The new command line option -k can be used in conjunction with the -i option in order to change the default behavior in case an unconventional field is encountered and automatically skip it.

14.9.3 Support for Null Fields and Field Default Values

The c-treeSQL import utility previously imported all table fields into c-treeSQL with the NULL FIELD flag set to "N" indicating that the field cannot be null, independent of whatever a table was created with support for NULL FIELDS or not.

The import utility has been changed to check if a table being imported was created with NULL FIELD support (this is true for tables created with c-treeDB APIs). If the table being imported was created with NULL FIELD support, the import utility updates the SQL dictionaries accordingly, by setting the NULL FIELD flag to "Y". If a table being imported has no NULL FIELD support, the NULL FIELD flag of SQL dictionary will be set to "N" and the SQL dictionary default value for that field will be set to 0 for numeric fields, or " " for string fields.

Default values are not available for SQL_LONGVARCHAR and SQL_LONGVARBINARY field types and the NULL FIELD flag for those fields are always set to "Y", even though null values are not supported for these types of fields. A side effect would be, for example, inserting a NULL value in a LONG VARCHAR field and reading back an empty string when the field is retrieved from a record. Please note that this only affects tables created without NULL FIELD support.

14.9.4 Support for Importing Table with an Alternative Name

The c-treeSQL import utility derives the table name to be stored in the SQL system tables from the physical file name, for instance the table custmast.dat is imported as custmast. Now it is possible to specify a SQL table name on the command line by using the -n option.
c-tree Plus Version 8.14 contains some of the most significant enhancements to performance that have been implemented to date. In addition, a myriad of new features in a variety of other subsystems have been added, all of which we anticipate our developer-partners will put to use immediately in their applications. While this Update Guide describes the latest innovations in this exciting new release, we thought it important to remind our users of the many other great features that make c-tree one of the best database application development products on the market.

15.1 C-TREE SERVER SDK

c-tree Plus includes the ability for you to create your own c-tree Server. You can also use the FairCom "core engine" within your own custom server. The c-tree Server SDK provides the advanced engineer the means to take control of the Server side of Client/Server development projects. Using FairCom's proven commercial database server, this technology exposes critical server-side subsystems and gives the developer unique insights and access to the source code of these components. The c-tree Server SDK is seamlessly integrated into the c-tree Plus product. While a separate license is required to deploy the Servers you create, everything you need for development and testing is included. Look for MASSIVE new additions to this SDK in a future release!

15.2 RUN-TIME DATA FILTERS

Run-time c-tree data filters provide a powerful, efficient way to filter records read from a file using criteria specified at run time. Run-time data filters are similar to using a c-tree conditional index, in which a conditional expression associated with an index determines whether or not a key value is added to the index for a particular record. However, conditional index expressions are typically associated with an index when it is created and, if changed, require reindexing in order to update the index contents based on the new conditional expression. Run-time data filters, by comparison, can be enabled for a file at any time on a client-by-client basis.

A run-time data filter allows record reads using standard c-tree ISAM API functions to transparently skip over records that do not match the data filter expression. In client/server mode, this filtering occurs on the server side, meaning that records that do not meet the criteria are not sent over the network. For this reason, using run-time data filters improves performance while
also simplifying application logic by allowing the c-tree Server, rather than the application, to filter the data.

15.3 PARTITIONED FILES

The c-tree Server supports a unique feature known as Partitioned Files. A partitioned file appears to be one file (or more accurately, one data file and its associated index files), but is actually a set of files whose contents are partitioned by the value of the partition key. Both the data files and index files are partitioned. This permits data with a defined range of values for the partition key to be rapidly purged or archived (instead of having to delete record-by-record each record within this range).

Partitioned Files maintain your data in separate c-tree Plus data/index files, while allowing access from a single host file. This feature is implemented at the ISAM level with virtually no API changes, and is transparent to our c-treeDB database layer and other higher-level interfaces into c-tree Plus. You can easily use standard c-tree Plus data searches on the entire file or on a member file. This allows the developer to easily purge or archive individual member files.

This unique feature requires a custom c-tree Server to be built with your individual partition rule support. Contact FairCom for assistance with generating your custom Server in addition to reviewing Chapter 15 "c-tree Server SDK" in the c-tree Plus Programmer's Reference Guide.

15.4 ADVANCED FILE ENCRYPTION

The c-tree Server supports encryption of data, index and transaction log files using a proprietary encryption algorithm. This provides the means to add an extra level of confidentiality to an application's data. FairCom designed its proprietary encryption algorithm for speed and efficiency, focusing on minimizing performance loss. This Standard File Encryption is not intended as a replacement for OS or other security systems, but rather, as a supplement to existing security. Standard File Encryption is suitable for most needs. Advanced File Encryption is offered as an alternative for those applications needing the highest levels of data confidentiality and integrity.

FairCom's goal of Advanced File Encryption is to expand options for developers and produce a suite of protocols that will protect the user data by what is loosely called strong encryption. The tradeoff for the enhanced security is a certain amount of performance overhead.

FairCom offers developers several well established advanced encryption protocols. To use Advanced File Encryption, the developer must build both a custom server with the ctCAMOsdk #define as well as the client library. By default, FairCom uses the Advanced Encryption Standard (AES) as selected by NIST. This block cipher algorithm is currently known as Rijndael. FairCom also supports DES, 3DES, Blowfish, Twofish, and others. FairCom recommends reviewing Chapter 15 "c-tree Server SDK" in the c-tree Plus Programmer's Reference Guide for information on building your own custom c-tree Server enabled with Advanced File Encryption.
15.5 UNICODE SUPPORT

Unicode is an effort to standardize the representations of all languages in computer format. Early standards, like ASCII, only encoded letters for English. Efforts to internationalize started with extending ASCII, but was limited by a 255 character set that would fit into one byte. Unicode incorporate the characters of all the major government standards for ideographic characters from Japan, Korea, China, Taiwan, and many other countries. Although Unicode is thought of as a wide-character encoding with 16 bits per character, Unicode standards include 8 bit multi-byte encoding (UTF8), 16 bit wide character encoding (UTF16) and 32 bit wide character encoding (UTF32).

c-tree Plus supports both UTF8 and UTF16 character encodings, as well as the conversion routines that permit unambiguous translation between the two formats. When building the c-tree Plus libraries, execute mtmake with the "u" flag to prepare the library for Unicode support. Standalone builds need the ICU libraries from the ICU web site, as described in Chapter 14 "Unicode Support" in the c-tree Plus Programmer's Reference Guide. Both the client and the c-tree Server must support Unicode. Please verify this when you install your server, or contact the FairCom office nearest you for assistance.
Chapter 16

Critical, Serious, and Other Fixes

The following fixes have been implemented within c-tree V8.14. This chapter is broken up into three sections: critical, serious, and other issues. Within the three sections, square bracket syntax are used to provide an understanding of which product the fix has been made to. Below is a table that explains this syntax:

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[c-tree]</td>
<td>Fixes impact c-tree in general</td>
</tr>
<tr>
<td>[c-treeDB]</td>
<td>Fixes impact the c-treeDB interface</td>
</tr>
<tr>
<td>[c-tree ODBC]</td>
<td>Fixes impact the c-tree ODBC Drivers</td>
</tr>
<tr>
<td>[Server]</td>
<td>Fixes impact the c-tree Server</td>
</tr>
<tr>
<td>[SQL]</td>
<td>Fixes impact the c-treeSQL Server</td>
</tr>
<tr>
<td>[SQL ESQL]</td>
<td>Fixes impact the c-treeSQL ESQL interface</td>
</tr>
<tr>
<td>[SQL ISQL]</td>
<td>Fixes impact the c-treeSQL ISQL interface</td>
</tr>
<tr>
<td>[SQL JDBC]</td>
<td>Fixes impact the c-treeSQL JDBC Driver</td>
</tr>
<tr>
<td>[SQL ODBC]</td>
<td>Fixes impact the c-treeSQL ODBC Driver</td>
</tr>
<tr>
<td>[Utility]</td>
<td>Fixes impact utilities</td>
</tr>
<tr>
<td>[r-tree]</td>
<td>Fixes impact the r-tree report generator</td>
</tr>
<tr>
<td>[VCL]</td>
<td>Fixes impact the c-treeVCL/CLX interface</td>
</tr>
</tbody>
</table>

16.1 CRITICAL ISSUES

16.1.1 [c-tree] Local Disk Read I/O Problem

A data flushing problem was discovered for Windows 32 bit versions related to the FPUTF-GET model. Because of an operating system failure to ensure the integrity of system pages, FairCom found it necessary to place low-level locks on read/write operations to ensure the OS would properly return complete (in fact, not partially updated) data pages. Before this change,
it was possible for a data page to be partially updated by the OS, giving c-tree a corrupted page of data. This change to add locking before reads and writes solved this problem, although it adds additional overhead related to performance.

16.1.2 [c-treeDB] ctdbWriteRecord Bug

c-tree will allocate space for a larger record size when a variable length record is updated that is larger than the current record size. It was discovered that if the write of this updated record failed, the field offsets for the newly allocated record buffer were not updated. This issue was resolved by explicitly updating the field offsets in the newly allocated record buffer.

16.1.3 [Server] PREIMG Transaction Undo

A file created with the ctPREIMG flag should not generate transaction log entries — these files support transactions but not recovery. It was discovered that a ctPREIMG file inadvertently generated log entries as if the file were created with the ctTRNLOG flag. A terr occurred during the undo of a nearly committed transaction. To avoid this problem, the undo logic makes a check to prevent the unexpected log entries for PREIMG files.

16.1.4 [Server] KDEL_ERR During Automatic Recovery

A problem that caused the server automatic recovery to terminate with error KDEL_ERR(4) has been rectified. This problem was detected during QA and was related to the recovery process that involved files created with the TRANDEP extended mode.

16.1.5 [Server] Duplicate Transaction Number

An internal synchronization failure related to the creation of a new transaction log resulted in different transactions being assigned the same transaction number, with catastrophic results. This scenario was avoided by forcing a synchronization object to control the assignment of the transaction number to a new log.

16.1.6 [Server] Checkpoint Deadlock

A synchronization deadlock was attributed to transaction cleanup of write-through files that forced a header write. The deadlock situation was avoided by holding off the header write-through until the synchronization object could be acquired.

16.1.7 [Server] Long Transaction in Superfile Causes Failure

A code path was discovered that could lead to a synchronization object being compromised, potentially leading to a catastrophic failure. This obscure scenario consisted of a long transaction involving superfiles to be abandoned by a dynamic dump. Checking if the synchronization object was already owned before attempting to reacquire it easily solved this.
16.1.8  **[Server] New Index Node Failure**

The logic to detect flaws in the abort node list processing will generate an L59 etcatend. This is a technique to capture symptomatic data to help diagnose the actual failure. In this case an abandoned transaction, caused by a waiting dynamic dump, resulted in a corrupted index node. The logic to clean index nodes in this scenario was not able to process the node, resulting in the L59. The logic to handle a node that cannot be processed will mark the index file as bad and close the low level physical file, failing with BIDX_ERR (527). CTSTATUS.FCS receives a "Trouble processing new index node" message.

A new server configuration keyword has been created to retain the traditional operation of capturing diagnostics in the event of an L59 error. The DIAGNOSTICS key word can take L59 as an argument if one wishes to return a L59 etcatend instead of the close file error.

**Note:** The new Server keyword is DIAGNOSTICS L59.

16.1.9  **[Server] Unexpected Client Termination While Aborting a Transaction**

An unexpected client termination occurred when attempting to abort a transaction. The source of the problem was traced to the rollback of a transaction dependent delete that encountered a file already closed by the user, thus yielding a terr(8890). To account for this scenario, code was added to CLISAM() to check for a transaction. If the file is involved in a transaction, an abort will be issued. This removes the possibility of an undo of a transaction encountering a file already closed.

16.1.10 **[Server] V8 Server Abnormal Termination**

An obscure issue in an early version of the c-tree Server Version 8 could cause the server to terminate abnormally under restricted conditions. When a user performed a logon, causing a write to the server's transaction log file, and at the exact same time a new transaction log was being created (internally rolling over to a new log), an invalid memory pointer was referenced causing the abnormal termination. This issue has been resolved.

16.1.11 **[Server] Server Hangs after ISAM Read Operation**

A subtle issue caused the server to hang when an ISAM read operation was performed with a record offset of zero and ISAM locks enabled. The read operation failed with ZREC_ERR (29) but the data file header was locked causing the server to hang. This issue has been resolved.

16.1.12 **[Server] c-tree Server Crash when Dump Stream File Open Fails**

If the c-tree Server's dynamic dump thread fails to open the dump stream file, the server may crash due to a NULL pointer reference while logging the error condition to the server's status log. The case reported to FairCom occurred when the dump script specified the name of a directory or the name of a non-existent path for the dump stream filename (using the !DUMP server’s configuration file keyword).
Because the dynamic dump thread had already logged the dump stream file name to the server status log, FairCom removed the logging operation that led to the crash.

### 16.1.13 [Server] Automatic Recovery Error

An obscure error condition occurred during the c-tree Server automatic recovery process. This problem is most apparent when using c-tree data file resources. A resource record is identified with a header on the front of the record as FFEF. When this recover bug occurred, the beginning of the resource record was clobbered, thus damaging the FFEF header indicator. This would result in an RRED_ERR(407) - Attempt to get non-resource info. Although this issue is most likely related to resources records, it could conceptually happen to a standard data record under the same condition. This issue has been fixed.

### 16.1.14 [SQL] SQL Crash Fixed

c-treeSQL Server experienced a catastrophic failure under the following circumstances:
1. Create a table with iSQL
2. Exit iSQL and delete the table using an OS delete command
3. Reenter iSQL and query the table just deleted.

The first query will result in an error. The second time a query operation is performed on this table the c-treeSQL Server locks up. This failure related to iSQL was resolved.

### 16.1.15 [SQL] c-treeSQL Server Memory Leak Source of Server Termination

A memory leak in the c-treeSQL Server was discovered while researching the following scenario:

Create a table with iSQL, exit iSQL and delete the table using an OS delete command, reenter iSQL and query the table just deleted.

The memory leak was eliminated by correctly initializing a pointer variable involved with getting the table id.

### 16.1.16 [SQL] Prevent Server Crash When SQL Clients are Active at Shutdown

The server shutdown process terminated abnormally as a result of releasing memory for an SQL client thread that continued to execute. This behavior was eliminated by avoiding the release of memory when there are client threads still active at the end of server shutdown.


The superfile compact utility, `ctscmp`, reads the superfile directory index of the original superfile and creates a member in the new superfile for each member name read from the superfile's
internal directory index. However, in the event of an error reading the superfile directory index, `ctscmp` did not detect the error and created a new superfile missing data for the members it could not read from the superfile directory index. The worst case situation is a superfile with a corrupt superfile directory index from which no member names can be read, which led `ctscmp` to create a new empty superfile and to overwrite the original superfile with the new empty superfile. `ctscmp` has been modified to check for an error reading the superfile directory index and in this case `ctscmp` now terminates with an error.

### 16.2 SERIOUS ISSUES

#### 16.2.1 [c-tree] Graphical Windows 32 bit versions Standalone Blocks RBLIFIL

The `RebuildIFile` function was improved to avoid a potential hang for Windows 32 bit versions clients built for the standalone model when a discrepancy exists between the IFIL structure and the header fields. In this particular configuration, `RebuildIFile` will call a function that prompts the user to resolve the conflict. In a Windows 32 bit versions graphical environment the prompt is suppressed, yet still waiting for a response, giving an impression that the process is hung. Since it is not possible to determine a graphical environment with 100% accuracy, the function has been modified to use the value from the header.

#### 16.2.2 [c-tree] Use AVLFILNUM() to Get a Valid File Number in `ctcntidx()`

The function `ctcntidx()` was modified to address a problem with superfile r-tree support in the single user TRANPROC operational model. An attempt was made to open reserved file number 0, resulting in an FUSE_ERR. To avoid this error, `AvailableFileNbr()` was called to get a valid file number.

#### 16.2.3 [c-tree] Multiple c-tree Instances in Each Native Thread

Application threads bound to a server kernel now have REGCTREE support consistent with the REGCTREE support found in multi-threaded clients. Each native thread can now have multiple c-tree instances by having each instance use a different OWNER value. When a registration context is switched for a thread, the new OWNER value is plugged into the OWNER macro. Therefore the same physical thread has more than one OWNER value.

#### 16.2.4 [c-tree] Rebuild Not Setting Index Corrupt Flag

RBLIFIL was not setting the index corrupt flag in the event of a failed rebuild. This issue has been fixed. If RBLIFIL returns DUPX_ERR, KDUP_ERR, KSRL_ERR, or some error other than DUPJ_ERR or DUPL_ERR, one or more of the index files will have their corrupt flags turned on.
16.2.5 [c-tree] Stop Server Start-Up with Incompatible Log Files

The internal format of the transaction logs used by the c-tree Server changed between V6.x and V7.x. It is not possible to use V6.x transaction log file with a v7.x server. In order to ensure that customers will not have problems, internal checks have been added to ensure the V7.x c-tree Server will not start up if it detects V6.x transaction logs. Prior to this check, a V7.x server would start, yet not properly operate with the older V6.x logs.

16.2.6 [c-tree] Invalid Approach to Detect CT_STRING Field Type Length

An issue related to the use of the field type CT_STRING, when placed in the fixed length portion of a c-tree record that is then used by the conditional expression evaluator (i.e.: Conditional Index or Data filter feature), has been fixed. The approach in which the conditional expression evaluator used to detect the length of the CT_STRING field type was wrong. Now the conditional index field scanning logic has been changed so that the DODA field length is used rather than the string length when skipping over CT_STRING fields in the fixed-length portion of the data record.

16.2.7 [c-tree] UPDCIDX Interaction with PRMIIDX and ctDROPIDX

UPDCIDX did not work as expected when updating a conditional index expression on data and index files created under TRANDEP (transaction dependent create) mode. According to c-tree ISAM documentation, the prescribed way to update a conditional index expression is:

1. Delete the index by calling ctDROPIDX
2. Call PermIndex with ifil->dxtdsiz == ctNO_IDX_BUILD
3. Call UpdateConditionalIndex to update the conditional index expression
4. Call RebuildIndex to re-populate the index

This procedure for updating conditional indices did not work because, under TRANDEP, if there are pending ctDROPIDX calls, then no calls can be made to UpdateConditionalIndex for any indices associated with the underlying data file.

This issue has been resolved.

16.2.8 [c-tree] NXTREC/PRVREC Issue When Called After ADDREC

An unexpected record was returned when NXTREC or PRVREC were called after an ADDREC call since the low level current index node was not updated on every add operation. The same applies to the variable length record variants such as NXTVREC and PRVVREC functions.

After fixing this issue, ADDREC or RWTREC will modify the low level current index node so that NXTREC/PRVREC operations behave as expected. A low level ADDKEY call will not affect the NXTKEY/PRVKEY behavior.
16.2.9 [c-tree] SETFLTR Memory Leaks Fixed

The memory for a data file filter is released when SETFLTR was called with no expression or when the file is closed. But if SETFLTR function was called to replace the existing filter expression, the memory for the data file filter expression was not released before it was updated with the new expression.

The code to initialize the temporary conditional expression used in the SETFLTR function did not release the memory allocated for the symbolic field names, unless an error was detected.

These SETFLTR memory leak issues are now fixed.

16.2.10 [c-tree] c-tree Plus Filter Problem Comparing Strings

Comparing CT_FSTRING strings that are as large as the field (i.e. there is no string terminator in the string), is a common situation for ctdb tables, however, the filter logic may not match a string that should actually match. The case where a CT_FSTRING is not terminated with a string terminator was not considered. The filter logic, in calculating the string length, would search for the string terminator. In its absence, the logic would scan to the end of the record instead, thus leading to the wrong string should a terminating character be found elsewhere outside the CT_FSTRING field. The filter logic was changed to scan for the string terminator only up to the end of the field.

16.2.11 [c-tree] DLOK_ERR on UNIX in Multi-Threaded FPUTFGET Mode

On Unix systems, different threads are granted locks to file regions already locked by another thread. To ensure proper lock behavior in multi-threaded FPUTFGET mode on Unix systems, c-tree maintains lock information for the current process for all files and threads check this lock table before attempting system locks. However, this feature relies upon generation of unique IDs for all open c-tree files, which is implemented using the ftok() system call. This system call was found to sometimes generate the same value for different filenames. Having more than one file with the same ID caused c-tree to fail to distinguish record locks for the two files. When a record was locked in the first file, an attempt to lock at the same offset in the second file failed with error DLOK_ERR (42).

FairCom corrected the error by generating the unique file ID using the c-tree thread handle and file descriptor.

16.2.12 [c-tree] Conditional Expression Involving \0 Not Properly Evaluated

The conditional expression scanner did not properly convert \0 to a null character. This behavior now functions as expected.

16.2.13 [c-tree] Duplicate Record Purge During Alter Table Rebuild

The conditional expression scanner did not properly convert \0 to a null character. This behavior now functions as expected.
16.2.14 [c-treeDB] c-treeDB Memory Violation

A memory violation error was detected when c-treeDB C and C++ record handling routines operated on CT_4STRING and CT_2STRING field types and the specified field lengths for CT_4STRING was less than 4 and CT_2STRING was less than 2.

When c-treeDB record manager initialize a record buffer for the first time, it calculates a "minimal" record length, in case of variable length records, setting all fields to empty. CT_4STRING fields occupy a minimum of 4 bytes, while CT_2STRING fields occupy a minimum of 2 bytes of the record buffer. The c-treeDB record initialization code used the defined length of the field without checking for the minimum space requirements for CT_4STRING and CT_2STRING fields, resulting in a record buffer that was too short and causing the memory violation. This issue is now fixed.

16.2.15 [Server] Key Compression with Keys > 255 Bytes Cause Server Crash

Compression of keys greater than 255 bytes in length requires special consideration of the single-byte prefix and suffix compression counts. The counts cannot exceed 255. The suffix (padding) compression count is treated properly. The prefix (leading character) compression count was not. In practice, most of the time the prefix count is handled without problem, but it is possible for the prefix byte to be truncated to one byte.

When ctFeatXTDCMP is turned on, then files with extended headers (i.e., not V6 compatible) do not have this problem. The code has been modified to ensure that when ctFeatXTDCMP is off or if the file does not have an extended header, then the prefix count will not exceed 255 bytes.

16.2.16 [Server] Abnormal Server Shutdown Due to Sharing Conflict

A scenario existed where the Server would not properly shutdown. The problem occurred when the Server and a Client thread had a resource sharing conflict. The Server was in the process of shutting down and a Client was in the process of logging on. This prevented termination of the client thread which resulted in the Server logging an abnormal shutdown. The resource conflict has been resolved allowing the client thread to terminate gracefully.

16.2.17 [Server] Windows 32 bit versions Service Background Issue

Typically, when the c-tree Server is running as a Windows 32 bit versions Service (background process), all interactive dialog screens are suppressed. A case was discovered that caused a particular dialog message to be invoked erroneously while in this background mode. This has been resolved.

In the case where the Server was running as a non-interactive WIndows Service (32-bit version), the dialog box was never displayed and the user was unable to respond to the prompt. This scenario left a thread suspended waiting for a response to the dialog box prompt. A modi-
A situation existed that involved an uninitialized variable in a bound server configuration. This variable was then used as a pointer which caused the process to generate an exception violation thus terminating the application. This situation was isolated to a bound server model and initializing the variable resolved the problem.

16.2.19 [Server] Service Control Manager (SCM) Reports Untimely Response

When running the Server as a Windows 32 bit versions Service it was discovered that unusually long Server startups or shutdowns would result in the SCM reporting that the service did not respond in a timely fashion. This circumstance arises when events such as long recovery times at startup, a long shutdown due to flushing the cache, or waiting for active clients occurs. This situation has been avoided by increasing the timeout value that the SCM is expecting a response.

16.2.20 [Server] Windows 32 bit Versions c-tree Server Frees Invalid Handle

A wildcard specification in a dynamic dump script for Windows 32 bit versions platforms attempted to close an invalid handle. This error was traced to the c-tree function, ct_findfile. Instead of returning 0 when no files were found, it would return the value from the platform specific findfile() function. This behavior only occurred on Windows 32 bit versions platforms. The issue has been resolved by modifying ct_findfile() to return zero (0) under this circumstance.

16.2.21 [Server] Server Windows 32 bit versions Service Utility Hanging Error

A problem where the server utility, ctntinst.exe, could hang has been rectified. This utility is used to install the server as a Windows 32-bit versions Service. If the c-tree server, when started as a Service, failed to become operational within a specific time frame, the utility would improperly wait, making it appear to hang. More specifically, if you started the a server in a new directory, the extra time it took for the server to create its initial LOG files would have been enough of a delay to trigger the utility to hang. This has been fixed.
16.2.22  [Server] Bound Server Shutdown Problem

A subtle issue which caused a shutdown exception violation, only under the BOUND Server model, has been resolved. If a thread calls CLISAM and then ctThrdTERM, an access violation occurred. This has been resolved. Users of the bound server model should note this fix.

16.2.23  [Server] Killing a Shared Memory Client

A user reported an exception violation for the c-tree Server for Windows while using the Shared Memory communications protocol. This exception was isolated and resulted when killing a client with the CTADMN administration program. This issue has been resolved.

16.2.24  [Server] Error 36 During Automatic Recovery

An obscure condition caused the generation of error 36 - Read error: check sysiocod error - during c-tree Server automatic recovery procedure. This error condition caused the very last portion of the log file to not be flushed to disk, causing the recovery procedure to generate the read error. This issue is now fixed.

16.2.25  [Server] Netware NLM Server Shutdown Issues

On rare occasions, a server shut down caused an abend. This was the result of a large number of memory and semaphore resources freed during the shutdown phase. The c-tree Server's shutdown logic required additional (non-preemptive) defers, to alleviate this problem. Any NLM user, who experiences aborts during server shutdown, should note this adjustment.

16.2.26  [SQL] c-treeSQL Server Shuts Down if Unexpected Error Occurs

In case of an unrecoverable error, the SQL server calls a panic function which logs the error and terminates the current running thread. In most of the cases this thread termination caused the SQL server to hang. This panic handling has been modified such that instead of terminating the current thread, the whole server is shut down.

16.2.27  [SQL] c-treeSQL Server Shutdown Issue

Two conditions having caused the c-treeSQL Server to invoke an exception violation during shutdown have been rectified.

1. When the c-treeSQL Server starts up, and if it's default database (c-treeSQL) does not exist, it will proceed to automatically create this default database. If the user "shuts down" the server while the "create process" is running, it was possible to get an exception violation. A user reported this issue when the c-treeSQL server was started as a "Windows Service" and then immediately asked to shutdown. This issue has been resolved.

2. An issue when running the c-treeSQL server as a "Windows Service", not specific to the database create issues mentioned above, has been fixed. Internal logic during shutdown which caused access to a NULL pointer has been resolved.
16.2.28 [SQL] SQL Client Could Open Files With Conflicting File Modes

A scenario existed that allowed a c-treeSQL client to inherit superuser privileges. Consequently an SQL client could be granted the right to open a file already opened exclusively. Ensuring the c-treeSQL client has the appropriate bit set in the user profile prevents this behavior.

16.2.29 [JDBC] JDBC Exception if 20 or More Parameters in a Prepared Statement

It was possible to receive and ArrayIndexOutOfBoundsException using the JDBC driver when creating prepared statements with more than 20 parameters. This restriction has been removed, and the allocated array will now dynamically adjust when setting the columns number.

16.2.30 [ISQL] Syntax Errors due to Formatting issues Fixed

If a line began with one or more spaces or tabs in a multiline SQL statement, some recognized SQL reserved words which were part of a column name, were parsed as a separate keyword. This didn't happen when there was no prepending white space. The ISQL parsing engine will now correctly parse these statements.

16.2.31 [r-tree] Add Sector Size Command Line Option in r-tree to Open Superfiles

r-tree's rtdrvr executable has been upgraded to accept a command line parameter allowing the sector size to be specified. This upgrade addresses problems related to a script file that contains superfile members with an incompatible sector size. A superfile acts differently than a standard index file where a different sector size value is allowed given that it is greater.

16.2.32 [r-tree] r-tree Bad Sort Key Logic

A customer experienced a difference when running the r-tree report engine under Solaris vs. Windows. An internal issue, which caused sorting to malfunction in HIGH_LOW environments, has been resolved.

16.2.33 [r-tree] r-tree External DODA Fixes

Two r-tree related issues were reported and fixed. The first issue relates to an r-tree report script returning a syntax error because a filename in an external DODA is ignored by RTSCRIPT, as the file name cannot be evaluated. The second issue relates to symbolic field names in an external DODA getting clobbered, thus causing the script error.
16.2.34 [Utility] c-treeSQL Import Utility Cannot Import Files with Extensions Other Than .dat

The SQL Import utility ctsqimp.c was enhanced. You may now import files with extensions other than .dat.

16.2.35 [Utility] c-treeSQL Import Utility Avoids Data Padding Prompt if the Table is Empty

The SQL Import utility, ctsqimp.c, performs analysis to determine the best padding character for the data type CT_FSTRING. ctsqimp.c will prompt the user for a padding character in the event that the file to be imported is empty. ctsqimp.c has been modified to avoid the prompt in the case of an empty file and use the default value of zero (0) as the padding character.


A scenario existed whereby the admin utility ctdmn prevented new security settings to take affect. If the keywords COMPATIBILITY FORCE_WRITETHRU and COMPATIBILITY WTHRU_UPDFLG were set in the ctsrvr.cfg file, then setting file security features through ctdmn (Owner, Permission or Password) was not successful. The problem was not flushing the header during the file security operation due to the combination of COMPATIBILITY settings. This issue has been resolved by ensuring the header is flushed under these circumstances.

16.2.37 [Utility] Superfile Compact Utility Improvements

The superfile compact utility, ctscmp, built for non-TRANPROC models failed due to an attempt to access a key value in the member name index. The TRANPROC version ignored this value, but the non-TRANPROC version returned the value and attempted to access the member, which lead to the error 401 when reading the DODA. A custom function was developed to ignore uncommitted key values in this situation.

16.2.38 [Utility] Last Member Number Cause Superfile Compact Utility Error

The superfile compact utility, ctscmp was enhanced to deal with a superfile that contains a large last member number. Previously this utility would exit with an error message. Now it is assumed that the last member number is corrupted and arbitrarily assigns 100000, allowing the utility to continue compacting. A future enhancement is planned to detect the possibility that a superfile may have had many members added and deleted, thus exceeding the limit of 100000 even though there are few members.
16.2.39 [Utility] Dynamic Dump Restore

The Dynamic Dump Restore utility - `ctrdmp` - may fail to restore a dynamic dump, displaying the following error message:

```
RB: rollback time precedes earliest checkpoint in log
```

This behavior was introduced by recent changes to the code required to clean-up unneeded log files. The clean-up routine deleted required log files when the dynamic dump spanned more than four log files, preventing the correct dynamic dump restore. This issue has been resolved.

16.2.40 [Utility] `ctdump` Always Prompts for the Script Name

It was noted that the `ctdump` utility always prompted for the script name, even if this was passed as one of the command line arguments. The problem was introduced when new features were added to this utility. The issue was resolved and the utility functions as before.

16.2.41 [VCL] c-treeVCL BLOB Field Stack Overflow Fixed

Delphi threw a “Stack overflow” exception when a VCL’s `ftMemo` or `ftBlob` field was used against one of c-tree blob’s fields, such as CT_2STRING, CT_4STRING, etc. A fix was applied to the c-treeVCL components to address this issue.

This issue also affected c-treeCLX for Kylix and the same code fixes have been applied to the Kylix c-treeCLX components.

16.2.42 [VCL] c-treeVCL Memory Leak

The TCtIndex and TCtDataSet classes were modified to cache the objects instantiated by the Fields, Indexes and Segments properties. The cached objects are now destroyed when the owner objects are destroyed.

16.3 OTHER ISSUES

16.3.1 [c-tree] Obscure Infinite Loop in `fndkey()`

An obscure condition was reported where c-tree would hang (infinite loop) in its internal find key routine (`fndkey()`). This condition was caused by a malformed or bad index entry (i.e., a programming error using c-tree low-level `ADDKEY`). Adjustments have been made to "break-out-of-the-loop" with the following index error: `BIDX_ERR(527)`.

16.3.2 [c-tree] Win32 Service Install Utility Eliminated Runtime Dependency

The commercial release of Version 7.12 contains a utility that installs the c-tree Server as a Windows 32 bit versions service (ctntinst.cpp). This utility, when compiled in debug mode, had a dependency on a specific Microsoft runtime dynamic library (`msvcp60d.dll`). This dependency has been removed.
16.3.3  **[c-tree] File Creation Resource Errors**

File creation resource errors fall into three categories along with their subsequent dispositions as follows:

- File mode of ctTRNLOG/ctPREIMG and an extended file mode of ctTRANDEP. The file will be deleted as part of regular transaction processing. This occurs naturally due to the presence of ctTRANDEP in the extended file mode.

- File mode of TRNLOG/PREIMG without ctTRANDEP. The file will remain on disk to be deleted by the application.

- File mode lacking the presence of TRNLOG/PREIMG and ctTRANDEP. In this case the file will be deleted.

16.3.4  **[c-tree] 64-bit Integer Multiply and Divide Errors**

Problems with 64-bit integer multiply and divide were discovered. These occurred under rare circumstances and were vulnerable only during rebuilds of files between 40GB and 44GB. Recasting a variable solved the multiply problem, while extensive modifications were required to correct the divide problem.

16.3.5  **[c-tree] Incorrect Number of Active Data Records in a Superfile**

Failure to test all applicable attributes of a space management record could have resulted in an incorrect number of active data records calculated for a superfile. This applies to non-huge files when ctHUGEFILE is enabled. Ignoring this member of the record header for non-huge data file members avoids this problem. A related scenario involving TABN_ERR(78) resulted in a ctecatend L7. This problem has been avoided by adding TABN_ERR(78) to the list of allowable errors during update of the space management information.

16.3.6  **[c-tree] WRTDEMO Cleanup**

The return value from the r-tree `report()` function is made available to `ViewRtreeError()` to allow display of the error description in the event an error is generated.

16.3.7  **[c-tree] SETNODE Adjustment - Client-Side Label**

A minor issue was fixed in the SETNODE function which could cause the node name to not be properly stored on the client-side structure.

16.3.8  **[c-tree] STPUSRA and ctSNAPSHOT Unresolved Functions at Link Time**

The new c-tree API functions `STPUSRA` and `ctSNAPSHOT` were inadvertently left out of the applicable (client, standalone, etc...) c-tree library during the build process. This problem was resolved by adding these functions to the m-tree file list to ensure they are included in the module definition file used to build the c-tree Library.
16.3.9 [c-tree] Header File Write Without Header Lock

A subtle issue caused a header file write without a header lock under certain circumstances during an open file operation. This issue affects only the FPUTFGET mode and has been fixed.

16.3.10 [c-tree] Client-Side Huge File frmkey Error

A call to frmkey behaved differently between the stand-alone and client-side models. The high order record offset value was not checked for a HUGE index file when frmkey was called from the client-side. The iTRFMKEY routine, called by frmkey in client-side mode, was modified to change this behavior.

16.3.11 [c-tree] Error Check for SERIALNUM During Record Update

An obscure situation was discovered which produced a KBUF_ERR(121) condition. Users who experienced this error when calling the SERIALNUM function should take note.

Internal routines about to update a file may call an internal function (chkredf()) to determine if the user has update permission, or to determine if the ISAM key buffers have been maintained. Since a call to the SERIALNUM routine (when SETOPS has turned on OPS_SERIAL_UPD) invokes this internal function (chkredf()) it is possible for SERIALNUM to return a KBUF_ERR(121) even though no index update could be implied from the SERIALNUM call. KBUF_ERR indicates that an ISAM update is not possible because the internal key buffers have not been maintained.

The code has been modified to skip the KBUF_ERR check on a call from SERIALNUM by turning off the appropriate file mode bit and resetting it after the call.

16.3.12 [c-tree] CT_CHAR Schema Segment Mapped to Unsigned Integer

Schema segments now automatically map a c-tree field type to a key segment type (mode). Before this modification, a CT_CHAR field (a one-byte field) was mapped into a REGSEG segment mode. This meant no transformation, and the byte values were treated as unsigned integers. Now the code defaults to treating a CT_CHAR field as a signed, one-byte integer. Please note that CT_CHARU was, and still is, treated as an unsigned one-byte field.

Adding #define ctPREV_CHARSEG at compile time will cause the code to revert to the old treatment (unsigned). A server compiled with the new default will revert to the old treatment if the following command is included in the server configuration file:

COMPATIBILITY   CHAR_SCHSEG

16.3.13 [c-tree] Change the Bound Server Interpretation of the ctThrdInit numthreads Parameter

The first parameter of ctThrdInit, numthreads, is interpreted differently depending on whether the mode parameter includes the ctThrdFIXED_THREADS bit. If so, then the number of
threads is fixed as specified, otherwise, the number of threads can grow. However, in the "bound" server model, the number of threads cannot grow.

Our new behavior enforces the bound server model to function more consistently with the ordinary server models and follows the conventions given in the Function Reference Guide. If you specify a maximum number of "users" in the configuration structure, then that determines the bound server thread limit. If you do not specify this number and the mode includes the ctThrdFIXED_THREADS bit, then the numthreads parameter is used to specify the maximum. Otherwise, the bound server model sets the maximum number of threads to the compile-time maximum.

### 16.3.14 [c-tree] Automatic Recovery Error INOT_ERR (101)

#define ctFeatZERO_RECBYT_OK permits zero record positions to be stored in an index. An index search that finds no entry returns zero and sets uerr_cod to INOT_ERR (101). Automatic recovery uses internal indices that may be searched and legitimately find no entry. This raised the INOT_ERR and may cause automatic recovery to return this error. The automatic recovery logic was modified to properly handle this INOT_ERR case. Now, when an internal index search finds no entry during recovery, uerr_cod is reset from INOT_ERR(101) to NO_ERROR(0). Any other error, other than INOT_ERR, is not changed.

### 16.3.15 [c-tree] No PUTFIL Error When Turning Off TRANLOG Support

If PUTFIL is called in the middle of a transaction for a file that has been updated in the same transaction, then turning off transaction support leaves updates in pre-image space that become invisible. A read that would have searched pre-image space now goes directly to disk. As a result, some subsequent operations, such as GETRES, may fail. GETRES may find 0xff's instead of a valid resource header resulting in RRED_ERR (407). PUTFIL now properly returns XTRN_ERR(770) in this situation.

### 16.3.16 [c-tree] PUTHDR to Suspend and Restore Serial Segment Handling

A call of the form PUTHDR(datno,YES,ctSUSSRLSEGHdr) will cause the serial segment handling for data file datno to be suspended. PUTHDR(datno,NO,ctSUSSRLSEGHdr) will revive the serial segment handling. The file must be opened exclusively on these PUTHDR calls. Any non-zero value in the second parameter invokes serial segment suspension; a zero value in the second parameter restores support. There is no check that the status is changing (i.e., no error is returned if the request is superfluous). Furthermore, the change is completely independent of transaction control: a roll back will not undo the effect of any prior calls to PUTHDR that change the serial segment status. Explicit calls must be made to change the status.

When serial segment suspension is invoked, everything is as before except that the serial number is not (automatically) inserted into the data record position that holds the serial segment. The header value is still updated. Suspension is useful for adding data to a file in which there is prior knowledge that the data already has a value in the serial segment that is consistent with existing data records and it is desired to maintain these values. It is likely that the header may
have to have its serial number value(s) updated using a call to PUTHDR with the ctSERNUM hdr mode once the suspension is revoked.

16.3.17 [c-tree] Allow ctTRANSMODE with no ctTRNLOG Under TRANPROC

cTRANMODE and ctPIMGMODE are designed to permit a file to automatically transfer between transaction support and no transaction support depending on what type of c-tree engine the data is under. With a slight change in parameters, ctTRANMODE and ctPIMGMODE can now be used in a create under a TRANPROC engine with the file mode NOT including transaction support. The file will be created transaction "ready" but the transaction support will not be turned on. If the file is closed and reopened under TRANPROC, then the reopen will automatically switch to transaction support.

To use this approach, the x8mode member of XCREblk should have ctTRANMODE or ctPIMGMODE turned on; ctTRANDEP and/or ctRSTRDEL turned off; and the file mode should NOT include ctTRNLOG or ctPREIMG.

16.3.18 [c-tree] Mac: mbOpen() Not Returning Proper System Error for File Open

The system error was not returned properly from c-tree's internal "mbopen()" function call. It was found that a system variable "errno" was used for the MAC, when there is no such thing as "errno" on the MAC. Removing the internal reference to "errno" solved this issue.

16.3.19 [c-tree] Mac: Improper Use of PBHOpenSync() and PBHCreateSync() Return Values

c-tree was incorrectly interpreting return values for these Mac OS functions. Changes have been implemented in the c-tree code as recommended by Apple's documentation for these functions.

16.3.20 [c-treeDB] Duplicate Record Purge During Alter Table Rebuild

Implemented a new CTDB alter table mode CTDB_ALTER_PURGEDUP to deal with cases when an alter table is performed on data files with duplicated index info. You should OR in CTDB_ALTER_PURGEDUP mode with other alter table modes.

```
if (ctdbAlterTable(hTable, CTDB_ALTER_NORMAL | CTDB_ALTER_PURGEDUP))
    printf("Alter table failed\n");
```

The new CTDB_ALTER_PURGEDUP mode can also be used with tables opened with CTOPEN_CORRUPT flag. If a table was corrupted you can open the corrupted table with the CTOPEN_CORRUPT mode and than call ctdbAlterTable with CTDB_ALTER_PURGEDUP mode to rebuild the corrupted data.

```
if (ctdbOpenTable(hTable, "corrupted_table",
```
16.3.21 [c-treeDB] CTDB Could Not Iterate a Table in "natural" Order

During an application migration it was observed that the source table did not have a
CTDB_ROWID_IDXNO index nor a CTDB_RECBYT_IDXNO, yet, the ability to iterate forward through the table without having to guess which user-defined index to use was desired. This situation occurred as not all indices included all rows in the table.

CTDB iterates through a table with the functions ctdbFirstRecord, ctdbLastRecord, ctdbNextRecord and ctdbPrevRecord. With the previous implementation, index 0 (zero), or the first user-defined index, is used by default to iterate the table. If no user-defined index was present, CTDB used the RECBYT or the ROWID index to iterate. Only if the table had no indices would CTDB iterate using the data file.

FairCom added a new constant, CTDB_DATA_IDXNO, to set the current default index to the table data file. With this new code, you can iterate through a table in "natural" order by setting the default index to CTDB_DATA_IDXNO.

16.3.22 [c-treeDB] CTDB Could Not Specify Read-Only Mode in the Same Manner as ISAM ctREADFIL

It was identified that it would be an advantage for c-tree CTDB to have a method to open a table read-only as was possible via the ISAM interface (using ctREADFIL). FairCom has added a new CTOPEN mode, CTOPEN_READONLY, to implement a read-only table open.

16.3.23 [c-treeDB] Alter Table Does Not Preserve ROWID Values During Rebuild

It was requested the row id field be preserved as a relationship key to link rows in different tables. The CTTable::AlterTable function did not preserve this field during all rebuild types.

Now, if a table has ROWID values, CTDB AlterTable will preserve these values for full rebuilds. This behavior change also correctly identifies tables that do not have ROWID, but rather, user implemented SRLSEG or CHSRL fields and segments. These additional fields are also preserved when performing a full rebuild. The next data file sequential number is also adjusted for the next expected sequential number.

16.3.24 [c-treeDB] CTField Class Member Function Error 4016

The c-treeDB C++ CTField class member functions CTField::SetType and CTField::SetLength were simplified by making explicit calls to the equivalent c-treeDB C set and get field functions. This is an improvement over calling a generic c-treeDB C ctdbSetFieldProperties function, which required parameters not applicable to be set as NULL. This change corrects
member functions `SetType` and `SetLength` from failing with error CTDBRET_FIELDEXIST (Field already exists).

16.3.25 **[c-treeDB] CTDB Not Calling ctThrdInit() in Multi-Threaded Models**

The c-treeDB API has been enhanced to detect the failure of a multi-threaded client application to call `ctThrdInit()`. All applications compiled for the multi-threaded model require a call to `ctThrdInit()` prior to any other c-tree calls. Failure to comply with this restriction will cause the generation of an exception violation. With this enhancement, `ctThrdInit()` is called transparent to the application.

16.3.26 **[c-treeDB] Wrong Field Data After ctdbCopyRecordBuffer**

A scenario existed where it was necessary to have several record objects sharing the same record context. Passing the original record object instead of a table object and creating a new record can accomplish this. It was discovered that the new record failed to be updated with correct data field pointers. Specifically, `ctdbCopyRecordBuffer` was not calling the internal function `ctdbUpdateRecord()`. This problem was resolved by calling the appropriate internal functions to update the data field pointers after the record buffer copy.

16.3.27 **[c-treeDB] ctdbCreateTable API Enhanced**

The c-treeDB API `ctdbCreateTable()` function returned conflicting error messages under rare circumstances. These circumstances were that the index table existed prior to making the call to `ctdbCreateTable()`. The data file was created on disk, however the add table operation to the data dictionary failed with CTDBRET_NOSUCHTABLE. A second attempt to create the table resulted in error DOPN_ERR (tried to create existing data file). This situation was resolved by removing the data and index tables from disk after the Add Table operation fails.

16.3.28 **[c-treeDB] Scan Function Problem**

The database scan functions return the wrong path in the following two situations:

- If a table is a superfile member.
- If the table path information in the database dictionary is terminated with a path separator character.

16.3.29 **[c-treeDB] Alter Table Function Index Member Deletion Error**

A subtle issue occurred where the c-treeDB alter table function failed to correctly delete an index member when performing an index rebuild on a table that has a number of indexes on different physical index files. This issue is fixed.
16.3.30 [c-treeDB] Variable Length Record Error

A subtle issue occurred where error 37 occurs when adding records with two or more consecutive CT4_STRING fields in the variable length portion of the record and no data was put in the CT4_STRING fields prior to writing the record. This issue is fixed.

16.3.31 [c-treeDB] Alter Table Issues

Internal testing detected a number of issues related to c-treeDB alter table routine, ctdbAlterTable:

• When performing a full alter table, the presence of CT_CHAR fields in a table caused an error preventing the termination of the alter table process.

• CTCREATE_NONULFLD, CTCREATE_NODELFLD and CTCREATE_NOROWID create modes are ignored when a table is re-created as a result of a full alter table. The new table is always created with the three optional fields regardless the original settings.

• ctdbAlterTable fails if the table extension is different from the default of .dat. The full alter table process should preserve the original table and index file extension.

These issues have been fixed.

16.3.32 [c-treeDB] Defer Begin Transaction Added

The ctdbBegin function was modified to automatically include a new c-tree transaction mode ctDEFERBEG when starting a new transaction. This change will cause a performance improvement if a transaction is terminated (ctdbAbort or ctdbCommit) without any data having been changed. When a transaction is started with ctDEFERBEG, and no data is added, updated or deleted, the transaction log entries are not flushed to disk, thus resulting in a performance improvement.

16.3.33 [c-treeDB] ctdbAddDatabase Does not Check if Database Dictionary Exists

The c-treeDB ctdbAddDatabase() function did not check for the existence of the database dictionary file before adding the database description to the session dictionary file. The ctdbAddDatabase function has been modified to check if the database dictionary file exists on disk before adding the database to the session dictionary.

16.3.34 [c-treeDB] Conditional Index not Working

The c-treeDB C API function ctdbUpdateCndxIndex, update conditional index, did not work as expected when setting or updating a conditional expression on a table created with TRN-LOG and TRANDEP (transaction dependent control), instead returning error 466 (IIDT_ERR) - Incremental Index: dfilno not a ISAM file. Please note that all tables created by c-treeDB C or C++ APIs have TRANDEP (transaction dependent control) turned on automatically.
This issue has been fixed and `ctdbUpdateCndxIndex` function now correctly operates on tables created with TRNLOG and TRANDEP (transaction dependent control).

16.3.35 **[c-treeDB] ctdbAddSegment Fail when Adding INTSEG and CT_BOOL**

The `ctdbAddSegment` function fail with error CTDBRET_INVSEGMODE when adding a segment of type INTSEG and field type CT_BOOL. The c-treeDB code was modified to allow the combination of REGSEG segment mode and CT_BOOL field type to be used when adding or inserting index segments.

16.3.36 **[c-treeDB] Segment Mode Change Failed to Trigger Index Rebuild**

The `ctdbSetSegmentMode` function can be used to modify the segment mode of an existing index segment. When the segment mode changes, a flag is set indicating that the segment mode was modified and CTDB alter table code uses this flag to identify changes for the segment. The CTDB alter table code did check the segment modification flag, which in turn failed to trigger an index rebuild.

16.3.37 **[c-treeDB] ctdbDatePack Fails to Pack Date 03/01/1700 Correctly**

An obscure issue was detected during internal tests when `ctdbDatePack` function was used to pack the date 03/01/1700 (March 1st 1700). The `ctdbDatePack` function takes a date in the form of year, month and day and converts it to the number of days since 03/01/1700 plus 1, thus, this particular date should be packed as 1. This issue affected no other dates. This issue has been fixed and `ctdbPackDate` now packs the date 03/01/1700 correctly.

16.3.38 **[c-treeDB] ctdbAlterTable Fails if Index File Name is Not Specified**

Internal testing revealed that when adding a new index to an existing table, without specifying an index file name, cause ctdbAlterTable function to fail with error code IINM_ERR (467) - Incremental Index: aidxnam NULL for 1st.

To add a new index to an existing table, you need to call ctdbAddIndex(), to add the new index to the table, and then call ctdbAddSegment() to add the segments to the new index. Once done adding indices and segments, you must call ctdbAlterTable() to perform the changes to the index.

When you add a new index to an existing table but specify an index file name, ctdbAlterTable mark the rebuild as CTDB_REBUILD_INDEX indicating that only this index need to be rebuilt since it will go into a separate file.

When you add a new index to an existing table but do not specify an index file name, ctdbAlterTable need to mark the rebuild as CTDB_REBUILD_ALL, indicating that all indices need to be rebuilt since this index must be added to an existing index superfile.
This issue was fixed by changing ctdbAlterTable to mark a new index add to an existing table and without an index file name as CTDB_REBUILD_ALL instead of CTDB_REBUILD_INDEX.

16.3.39 [c-treeDB] Usage of Invalid Index Key Length for Modified Index

c-treeDB alter table code may cause the creation of an index with an invalid key length when an existing index with no duplicate key is changed to allow duplicate keys. The c-treeDB alter table code used the existing index key length for the modified index instead of adding the recbyt size to the key length.

This issue has been fixed and the c-treeDB alter table code automatically accounts for the correct size of recbyt added to the existing key length. Please note that the recbyt size is typically 4 bytes for files without HUGEFILE support and 8 bytes for files with HUGEFILE support.

16.3.40 [c-treeDB] Rebuild Index Failure

When ctdbAlterTable is called with CTDB_ALTER_INDEX mode and the first IIDX structure has the aidxnam member set to NULL and c-tree function PRMIIDX return error 467 (IINM_ERR) - Incremental Index: aidxnam NULL for 1st. ctdbAlterTable has been fixed to provide the default index file name (the table name with the default index extension).

16.3.41 [c-tree ODBC] Non-Null-Terminated Fixed-Length Strings in Variable-Length Part of Data Record

A "Buffer too small" error was reported when a c-tree Plus ODBC Driver (not a c-treeSQL ODBC Driver) attempted to read data from a variable length c-tree data file. More specifically, the error condition was raised when a CT_FSTRING(fixed length string) field is located in the variable length portion of a record and the CT_FSTRING field is not null terminated (i.e., the field contains non-null bytes for the full field length as defined for the field in the DODA).

The ODBC Driver SDK function OT_FLDLEN calculates field lengths based on a field's data type in order to walk through c-tree's record buffer. The ODBC driver interpreted the field contents as having exceeded the defined field length, resulting in the "Buffer too small" error.

For all c-tree string data types other than CT_STRING, the ODBC driver now uses the field length specified in the DODA, if the field length is non-zero, as the actual field length. This change ensures that the ODBC driver stops scanning the data record buffer for the end of a given fixed-length string field when the defined field length limit is reached and whether or not the field contains a null terminator.

16.3.42 [Server] Set FILE_HANDLES Defaults to Maximum Number of Open Files the Server Supports

The c-tree Server now sets its FILE_HANDLES setting to a reasonable default value. If FILE_HANDLES is not specified, it now defaults to the maximum number of open files the server supports (based on the FILES setting plus some additional file handles for administra-
The server attempts to increase the file descriptor limit to this value at startup. If the server detects the hard limit on file descriptors is lower than the maximum number of open files the server supports, the server logs the following warning message to the server status log:

"Warning! The hard limit on file descriptors available to the c-tree Server process is lower than the server's FILES setting. This could lead to errors opening files."

16.3.43 [Server] Skip Flush of Updated Cache Pages at End of Dump Checkpoint

The end of dump (EDUMP) checkpoint flushed updated data and index cache pages to disk and synced the file system cache to disk. This was unnecessary, as this does not affect the state of the backup file. The end of dump checkpoint has been changed to skip the updated cache page flushing and file system syncing. This change is expected to reduce the total time required to complete a dynamic dump (especially if there are many updated cache pages at the end of the dump). The behavior of the start of dump (PDUMP) checkpoint is unchanged: it still flushes updated cache pages to disk as it has always done.

16.3.44 [Server] ctUSE_RLIMIT Support for Linux File Handles

ctUSE_RLIMIT support has been added to Linux such that the server configuration keyword HANDLES can be used to increase the number of descriptors available to a single process. This will make life a little easier if you use large numbers of files on this operating system. This previously required the use of ulimit prior to starting the server.

16.3.45 [Server] Cache Page Removed From List to Flush

After an apparently clean shutdown, the next Server start-up indicated a prior dirty shutdown. This scenario occurs when a file is deleted and is associated with a committed transaction, not yet flushed. When a transaction is committed it is possible for it or parts of it to remain in cache. When this is the case the cache page is marked as vulnerable. If the file associated with this transaction is deleted an error is flagged when the transaction is flushed. Removing the file's cache page from the vulnerable list when its file is deleted resolved this issue.

16.3.46 [Server] Windows RCHK_ERR After Apparent Clean Shutdown

An uninitialized transaction state variable related to internal thread management resulted in a Server startup error RCHK_ERR:

"Log file entry failed to find checkpoint"

This occurred after an apparent clean shutdown. Proper initialization of the transaction state variable is now performed which will prevent this behavior.

16.3.47 [Server] Advanced Encryption Master Password Over 9 Characters

A c-tree Server with advanced encryption enabled will prompt the user for a password at start-up. It was discovered that the Server only recognized the first nine characters of the password
entered. The Server was modified to accept a password length of 128 characters. This is consistent with the utility that generates encrypted passwords, ctcpvf.c

16.3.48 [Server] Default Unix Server Compile Switches Changed

In a Unix environment the following compile time switches are now enabled by default.

#define SYNC_LOG
#define ctFeatUSE_FDATASYNC

16.3.49 [Server] Add User Termination Check to Rebuild Loop With Memory File

During the rebuild of an index from a memory data file, it was not checked to see if the user was terminated. A call to ctUserAlive() was implemented, such that the loop can exit when the user is terminated. Because memory files were introduced with the Version 8, this would only impact pre-release customers using memory files.

16.3.50 [Server] LOKDMP Displays Incorrect File Disk I/O Figures

When the c-tree Server is compiled with the snapshot feature enabled (#define ctFeatSNAPSHOT), performance monitoring snapshots will be taken. The per-file disk I/O figures are 8-byte values but were inadvertently treated as 4-byte values. The format specifier used to display these fields was changed to display the correct Disk I/O values. Because the SNAPSHOT feature was introduced in Version 8, this would only impact pre-release customers using this feature.

16.3.51 [Server] Restoring the Top-Level Exception Handler

It was discovered that the server keyword DIAGNOSTICS NO_EXCEPTION_HANDLER, designed to restore the top-level exception handler for the c-tree Server process, had no effect. At startup, the c-tree Server overrode the default top-level exception handler for the c-tree Server process with the server's own exception handler function. The server did this such that in the event of an unhandled exception, the server could perform some minimal shutdown operations before terminating. This keyword was intended to restore the top-level exception handler function to the default exception handler. The reason this keyword had no effect was that the test for this keyword occurred before the configuration file was read. This problem has been corrected.

16.3.52 [Server] Auto Transaction Commits Improved

A new SETOPS function option, OPS_KEEPLOK_TRN has been added. This new option causes the locks involved in automatic transactions on record adds and updates to be kept after the automatic commit. Record delete operations will still have the locks freed.

This change improves the performance of automatic transactions by avoiding the network overhead of the TRANBEG / TRANEND calls.
Please remember that you can enable automatic transaction for ISAM record add, update and delete operations by calling the SETOPS function with option OPS_AUTOISAM_TRN.

16.3.53 [Server] Permit Time-outs to be Interrupted

A blocking SYSMON call with a long timeout period can now be interrupted when the client-side application terminates. This modification permits a thread waiting (even indefinitely) on a SYSMON message queue (by calling SYSMON) to detect that the thread has been marked for termination or that the server is shutting down. This modification extends the capability of the SYSMON routine to cancel a pending call without terminating the client.

A user belonging to the ADMIN group can call SYSMON using SYSMON CANCEL_MAIN (and/or in V8 SYSMON CANCEL_PERF) to cancel a pending SYSMON call for another thread. The thread waiting on the pending SYSMON receives a XMON_ERR (760). If no such pending thread is waiting on a pending SYSMON, the SYSMON cancel call receives a NMON_ERR (761).

A thread that has an active SYSMON call may be canceled by using SYSMON OFF option. Error NMON_ERR (761) is returned if SYSMON OFF finds no active SYSMON for the caller.

16.3.54 [Server] Superfile Member Create Causes Server Core Dump

An obscure condition that could lead to the c-tree Server abnormally terminating has been resolved. An attempt to create a superfile member that already exists and is open at the time of the create results in an error (such as DOPN_ERR or KOPN_ERR), but then calls code designed to delete a file after a failed create. While the delete code is smart enough to know whether or not a file should be deleted, it de-references a c-tree FCB pointer that is garbage under this particular situation. This situation has been easily rectified by initializing the FCB pointer to NULL and testing the FCB pointer prior to calling the delete file code.

16.3.55 [Server] Application Server Crash During Shutdown with Active Client

It was found that shutting down the server while a client was active might lead to an access violation error. The access violation was caused by the remaining client(s) attempting to use memory that had been freed by the server at shutdown. The server checked for active clients but this did not take into account clients that were completing a shutdown process. This issue is fixed.

16.3.56 [Server] User Name and Password Not Maintained by Application Server

An application server that established a direct connection to the c-tree Server engine did not have its user name and password set to the specified values when calling INTISAMX(). This omission prevented the calling thread from using ADMIN-only features. This issue was resolved by setting the username and password for bound threads to the specified values in INTISAMX calls. Please note that a thread that establishes a direct connection to the c-tree
Server engine can likely specify any user name and password it wants, as the user validation logic is not implemented in the c-tree initialization logic for bound threads.

16.3.57 [Server] ctstat Utility Crashes When Using Function Option

Two cases were observed in which early versions of the ctstat utility crashed when using the function (-func) option:

1. When running ctstat with the -func option, ctstat crashed if the snapshot function returned an entry for a system queue function.

2. When <Ctrl>-<C> is used to terminate the ctstat process on Windows, ctstat crashes because Windows handled the signal by creating a separate thread. This thread had not established a c-tree connection, and the c-tree client code made a reference to a NULL ctWNGV.

Both issues are fixed.

16.3.58 [Server] Automatic Recovery Failure with LOPN_ERR

FairCom identified an uncommon situation in which the c-tree Server deleted a transaction log that was required for automatic recovery, causing automatic recovery to fail with error LOPN_ERR(96). This situation occurs when the server creates a new transaction log and the earliest transaction for which unwritten updated cache pages exist is the first transaction in a log. In this case, the server did not detect that it needed to preserve the log. The server now properly detects this situation and preserves the log.

16.3.59 [SQL] HAVING Clause is fixed

It was possible for a SQL statement containing the HAVING clause could appear make the query hang. It was found that a bug caused the optimizer to take a different path when the HAVING clause was present. This bug has been quashed.

16.3.60 [SQL] SQL/ODBC Error when CT_DATE Fields were zero

Existing data and index files, when imported into c-treeSQL, usually do not have the necessary internal support to handle NULL fields. The NULL field support for data files is automatic only for tables created with the c-treeDB API or with the c-treeSQL Server.

c-treeSQL reported a date conversion error when using imported tables without NULL field support and with CT_DATE fields with a value of zero. This error prevented, in most cases, c-treeSQL or ODBC from using imported files without NULL field support and with CT_DATE fields with a value of zero.

This issue was fixed by checking the value of CT_DATE fields. If the value is zero, set the NULL field flag, instead of converting the value of zero to a valid SQL date.
16.3.61 [SQL] c-treeSQL Server Java Edition Error

When running c-treeSQL Server Java Edition as a service under Windows, two problems occurred, both of which have been fixed:

- A system signal was causing the server to terminate after a user logoff.
- The Java JVM engine needed to be instantiated with a "-Xrs" switch to indicate that signals are to be ignored.

16.3.62 [SQL] Hide Columns Within a Table in the SQL Import Utility

The FairCom ODBC driver supports a feature that allows a user to hide columns within a table by prefixing a column name with an underline character. The "hidden" fields do not appear on SQL statements executed by the FairCom ODBC driver.

The CTSQLIMP utility now prompts the user for action upon detecting a field name with a leading underline, '_'. The user has the option of importing the field, making it visible to the SQL Server. Alternatively the user can decline to import, in which case it will not appear in the SQL data dictionary, making it invisible to the SQL Server. A further check is made during index processing to make sure hidden fields are not segments of an index.

16.3.63 [SQL] Error 21041 When Selecting Rows with CT_TIMES Columns

Performing a select * from an imported table containing a CT_TIMES field may cause an error 21041 - Can't perform type conversion - in the case where the CT_TIMES field contained a value of zero (0). c-treeDB does not consider a CT_TIMES value of 0 (zero) as a valid value and the CTDATETIME conversion routines failed with an invalid timestamp value. The issue was fixed by detecting when CT_TIMES fields contain a zero value and setting its NULL bit flag.

16.3.64 [SQL] SQL Wrong Results When Selecting Inexistent rowed

The c-treeSQL server returned error -17101 "FSS ctree error:CT - Could not find isam keyno request" when a select query specified a non-existent rowid value, instead of the expected "0 records selected" message.

This issue is now fixed and select queries such as the one shown below return '0 records selected' if no rows in table have this rowid value.

```
select * from table where rowid = '1234567'
```

16.3.65 [SQL] Implemented Record Locking in CTSQLIMP

During internal tests conducted on the CTSQLIMP utility, it was found that a possible vulnerability existed when both the c-treeSQL Server and the CTSQLIMP utility updated the same SQL system catalog table at the same time.

Changing CTSQLIMP to lock any records in the system catalog tables that are modified when tables are imported into c-treeSQL resolved this issue.
16.3.66 [SQL ODBC] SQL Server Connection Error in Remote Machine

The c-treeSQL ODBC driver returned an error when connecting to a c-treeSQL Server on a remote machine when the address of the remote server was presented in the form of "port-number@ipaddress". The '@' character was not recognized by the ODBC driver.

The ODBC driver was modified to allow the use of the '@' character in the specification of remote c-treeSQL Server names.

16.3.67 [SQL ODBC] Better performance for ctdbGetIndexByUID

ctdbGetIndexByUID was modified to cache the UID of indices. This change greatly increases the performance of the SQL ODBC driver, as it only needs to search the database dictionary file once per index UID over the lifetime of the index object.

16.3.68 [SQL ESQL] Prevent eSQL From Closing Standard Output on Disconnect

During internal FairCom testing of the embedded SQL interface, it was found that when an embedded SQL client disconnects from the c-treeSQL Server, the embedded SQL code disconnect logic also closes a message log which by default is directed to standard output. With the closure of the standard output stream, subsequent calls to printf() do not output anything.

FairCom adjusted the message log close logic to avoid closing the message log stream if the message log stream is directed to standard output.

16.3.69 [SQL ESQL] ESQL Limitation of 20 Columns

It was discovered that an ESQL statement could not process more than 20 columns. This was found to be a hard coded limit in the internal code. This restriction has been lifted.

16.3.70 [r-tree] External DODA with Alignment Restrictions Cause Violation Error

A segmentation violation error is generated when using an external DODA on a system with a processor that enforces alignment restrictions (e.g., SOLARIS). The issue is caused when external DODA fields passed to an r-tree enabled server were not aligned. The issue has been resolved by forcing the external DODA fields to be aligned. Both the client and server must be updated.

16.3.71 [Utility] ctBTRV - Btrieve Utilities Updated

The utility CTBUTIL, when invoked using the -nolog command line switch, would inadvertently leave the TRNLOG flag in the data file turned off. This could have led to data integrity problems and has been corrected.
16.3.72 [Utility] Compact IFIL Utility Multi-Threaded Client Error

Users that attempted to compile the Compact IFIL Utility (ctcmpcif) as a multi-threaded client application experienced a MINT_ERR(738) error. The MINT_ERR error resulted from the fact that the utility did not properly call the Thread Initialization function `ctThrdInit()` required for multi-threaded applications. This has been fixed.

16.3.73 [VCL] Freezing Issue with TCtDatabase and TCtTable Components

An issue was fixed that caused the Delphi 7, C++ Builder 6, Kylix 2 and Kylix 3 IDE to freeze when TCtDatabase and TCtTable components were dropped on a form and the IDE File | Close All menu option was selected.

16.3.74 [VCL] Internal Post Method Trapping Exceptions

An issue was fixed that caused an internal post method to trap exceptions to implement automatic transaction processing. The internal post method re-raised the exception after dealing with automatic transaction processing to enable applications using TCtTable component to deal with eventual errors during record add and record updates. The same fix was applied in the case of automatic transaction processing for record deletion.

16.3.75 [VCL] Modified Record Buffer Not Indicated

An issue was fixed where the TCtTable component was not calling the TDataSet method SetModified to indicate that the record buffer was modified. This change fixed an issue where TCtTable did not call post automatically when the current record pointer moved away from a modified record by calling First, Last, Next, Prior, etc.

16.3.76 [VCL] TCtTable Component Properties Not Set at Runtime

An issue was fixed which caused the VCL component to fail when calling the appropriate c-treeDB functions while setting the Directory, GroupId, Permission, and Password properties of TCtTable at runtime. This problem was manifested by the application using values of these properties set at design time. Initializing the underlying c-treeDB with the new property values set at runtime eliminated this inadvertent behavior.

16.3.77 [VCL] GotoBookmark Method Failed to Retrieve Bookmarked Record

It was found in c-treeVCL and c-treeCLX that calls to method GotoBookmark of TCtTable component would fail with error "Can't field records.". A change was applied to method `InternalGotoBookmark()` of TcDataSet class to correctly fetch a record that was previously bookmarked with a call to GetBookmark method.
17.1 c-tree Plus Low-Level API Functions

This section provides detailed function descriptions for c-tree Plus API functions added or updated with this release.

The following functions have been added:

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReplaceSavePoint Function</td>
<td>Maintain a moving savepoint within a transaction.</td>
</tr>
</tbody>
</table>
17.1.1 ReplaceSavePoint Function

Maintain a moving savepoint within a transaction.

SHORTNAME
SPCLSAV

TYPE
Low-level data file function.

DECLARATION
LONG ctDECL SPCLSAV(VOID)

DESCRIPTION
Call ReplaceSavePoint to establish a savepoint while at the same time clearing the previous savepoint. ReplaceSavePoint can be used to maintain a single "moving" savepoint within a transaction, which is useful for applications that may need to undo the work since the last savepoint but that never need to restore back to a point prior to the most recent savepoint. ReplaceSavePoint provides this ability in the form of a single savepoint rather than multiple savepoints. Because ReplaceSavePoint clears the previous savepoint, only the most recently established savepoint can be restored to. To restore to this savepoint, call TRANRST(-1).

RETURN
ReplaceSavePoint returns a non-zero value to indicate success and a value of zero to indicate failure. If the return value is zero, uerr_cod contains the c-tree error code. If a client supports ReplaceSavePoint but the server does not, the message "Bad raw function #. fn: 231" will be placed in CTSTATUS.FCS and the c-tree error SFUN_ERR(170) will be returned. A RestoreSavePoint cannot go back beyond a special save point set with SPCLSAV. Further, ClearSavePoint cannot clear a special save point. Either of these situations returns a SPCL_ERR(753).

EXAMPLE
An example of the use of SPCLSAV is with the SQL layer that must be able to undo the last update, and may involve a very large number of updates within a single transaction.

SEE ALSO
SetSavePoint and RestoreSavePoint.

17.2 c-tree Plus ISAM API Functions

This section provides detailed function descriptions for c-tree Plus API functions added or updated with this release.

The following functions have been added:

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocateRange</td>
<td>Allocate buffer space and establish a new range.</td>
</tr>
</tbody>
</table>
## New Function Descriptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlockingISAMRead</td>
<td>Add support for a blocking record read.</td>
</tr>
<tr>
<td>CloseCtFileByName</td>
<td>Close and remove from memory a memory file.</td>
</tr>
<tr>
<td>ctCallback</td>
<td>Establish a notification callback.</td>
</tr>
<tr>
<td>ctNotify</td>
<td>Send notification messages to a queue.</td>
</tr>
<tr>
<td>ctSysQueueClose</td>
<td>Closes an existing queue.</td>
</tr>
<tr>
<td>ctSysQueueCount</td>
<td>Return the number of messages waiting in the queue.</td>
</tr>
<tr>
<td>ctSysQueueLIFOWrite</td>
<td>Add new data at the beginning of a queue.</td>
</tr>
<tr>
<td>ctSysQueueMlen</td>
<td>Retrieves the length of the next available message in the queue.</td>
</tr>
<tr>
<td>ctSysQueueOpen</td>
<td>Open or create a new queue.</td>
</tr>
<tr>
<td>ctSysQueueRead</td>
<td>Read data from the queue.</td>
</tr>
<tr>
<td>ctSysQueueWrite</td>
<td>Add new data at the end of a queue.</td>
</tr>
<tr>
<td>FirstInRange</td>
<td>Read the first data record in a range.</td>
</tr>
<tr>
<td>FreeRange</td>
<td>Reset and free allocated buffers for range operation.</td>
</tr>
<tr>
<td>LastInRange</td>
<td>Read the last data record in a range.</td>
</tr>
<tr>
<td>NextInRange</td>
<td>Read the next data record in a range.</td>
</tr>
<tr>
<td>PreviousInRange</td>
<td>Read the previous record in a range.</td>
</tr>
</tbody>
</table>
17.2.1 AllocateRange

Allocate buffer space and establish a new range.

SHORT NAME
ALCRNG

TYPE
ISAM function

DECLARATION
COUNT AllocateRange(COUNT keyno, COUNT segcount, pVOID lRange, pVOID uRange, pCOUNT operators)

DESCRIPTION
AllocateRange allocate buffer space and establish a new range based on the segment values passed on lower, upper and the operators for each segment. keyno identify the index file that should be used to establish the range. segcount indicates the number of index segments values that should be used for setting the range, and the number of operators. lRange is a buffer with the lower range segment values, while uRange is a buffer with the upper range segment values. operators is an array of operators for each segment value. Currently the operators must be the same for all segments and only the first operator is used.

The operators CTIX_EQ, CTIX_NE, CTIX_GT, CTIX_GE, CTIX_LE, CTIX_LT are open ended and use only the lRange buffer for range values and uRange is ignored and maybe set to NULL. For example,

The operators CTIX_BET, CTIX_BET_IE, CTIX_BET_EI, CTIX_BET_EE and CTIX_NOTBET use both lRange and uRange buffers to establish the lower and upper bound values. The range is set for all index entries that are situated between the lower bounds and upper bounds values.

The segment values are stored in lRange and uRange buffers in the same order and type of the index segment definition.

If a previous range exists for this index, the previous range is released and the new range is established.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>No error occurred.</td>
</tr>
</tbody>
</table>


EXAMPLE
COUNT operators[] = {CTIX_BET, CTIX_BET};

if (AllocateRange(1, 2, buffer1, buffer2, operators) != NO_ERROR)
{  
    printf("AllocateRange failed\n");  
}

LIMITATIONS

Only the first operator in the array of operators is considered.

SEE ALSO

FreeRange, FirstInRange, LastInRange, NextInRange and PreviousInRange
17.2.2 **BlockingISAMRead**

c-tree ISAM **BlockingISAMRead** function add support for a blocking record read.

**SHORT NAME**
BLKIREC

**TYPE**
ISAM function

**DECLARATION**
```c
COUNT BlockingISAMRead(COUNT filno, NINT opcode, LONG timeoutsec,
                        pTEXT blockcond, pVOID target, pVOID recptr, pVRLEN plen)
```

**DESCRIPTION**

A blocking record read permits an application to attempt to read a record, and if no existing record satisfy the read request, **BlockingISAMRead**() blocks until such a record exists or the times out period expires.

**BlockingISAMRead** supports blocking ISAM reads on fixed or variable length records subject to an optional blocking condition that may be a logical expression or a function callback, just as in a set filter call (SETFLTR).

*opcode* is one of ctBLKIREC_FIRST, ctBLKIREC_NEXT, ctBLKIREC_PREV, ctBLKIREC_LAST, ctBLKIREC_GT, ctBLKIREC_GTE, ctBLKIREC_EQL, ctBLKIREC_LTE, ctBLKIREC_LT; and *timeoutsec* specifies the number of seconds the blocking read will wait before returning if no record satisfies the optional *blockcond* argument. *target* should be non-NULL only if *filno* specifies an index and the *opcode* requires a target value (e.g., ctBLKIREC_GT). The record is read into the buffer pointed to by *recptr*. *plen* should be NULL for a fixed length read, and should point to the length of the output buffer (recptr) for a variable length read. *plen* is updated to the actual record length upon successful return.

The function operates by performing the requested ISAM operation (FIRST, NEXT, etc.). If the requested ISAM operation successful, **BlockingISAMRead** then sees if the optional blocking condition is satisfied. If no record was found or if the record does not satisfy the blocking condition, and *timeoutsec* is non-zero, then the read blocks (sleeps) for *timeoutsec*. The sleep is interrupted if the target file is updated, and this process repeats until a record is found that satisfies the condition, or the block times out. A time out condition is indicated by returning NTIM_ERR(156). **BlockingISAMRead** requires server notification support to operate correctly.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>No error occurred.</td>
</tr>
<tr>
<td>156</td>
<td>NTIM_ERR</td>
<td>Timeout error</td>
</tr>
</tbody>
</table>

**EXAMPLE**

```c
struct {
    TEXT    job_name[256];
    .......
    LONG    start_time;
    .......
} JOBMAN;

TEXT    blockcond[128];

/*
** setting the first character to zero means no
** blocking condition
*/
blockcond[0] = '\0';

start:
    rc = BLKIREC( keyno, ctBLKIREC_FIRST, WAITTIME, blockcond, NULL, &JOBMAN, NULL);
    if (rc == NTIM_ERR) {
        /* time out occurred. check whether to continue
           ** job management processing
        */
        if (**continue** {
            blockcond[0]= '\0';
            goto start;
        } else
            goto end;
    } else if (rc) {
        /* unexpected error */
        handle possible error conditions
        [examin BLKIREC source for possible error cases]
    } else {
        /* found job record. check job start time */
        if (**JOBMAN.start_time <= current_time** {
            launch job
            update record (so no longer at front of file)
            blockcond[0] = '\0';
            goto start;
        } else {
            /*
              ** job not ready to run yet. create a
              ** blocking condition so that if a job with
              ** an earlier time is added to the job file
              ** the BLKIREC will return
            */
            ctrt_sprintf(blockcond, "start_time < %ld",JOBMAN.start_time);
            goto start;
        }
    }
end:
```
17.2.3 CloseCtFileByName

Close a memory file and remove it from memory.

**SHORT NAME**
CLSNAM

**TYPE**
ISAM function

**DECLARATION**
```
NINT  CloseCtFileByName(pTEXT filnam, pTEXT fileword);
```

**DESCRIPTION**

CloseCtFileByName closes a memory file and remove its data from memory.

Parameter *filnam* is the file name.

*fileword* is an optional file password. If *fileword* is null then there will be no password for this file. If a password is established, every user will need to use the password to be able to open the file.

In order to be able to remove the file from memory, all users must have the file closed. To close the file and reopen it, use CloseIFile.

**RETURN**

CloseCtFileByName returns NO_ERROR in case of successfully closing and removing the file from memory.

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>FNOP_ERR</td>
<td>Could not open file.</td>
</tr>
</tbody>
</table>

See the *c-tree Plus Programmer's Reference Guide* for a complete listing of valid *c-tree Plus* error values.

**LIMITATIONS**

Client/Server mode only.
**17.2.4 ctCallback**

Establish a notification callback.

**SHORT NAME**

ctCallback

**TYPE**

ISAM function.

**DECLARATION**

```c
NINT ctCallback(NINT opcode, NINT objhandle, ctCallbackPtr cbptr,
                NINT contents, NINT controls)
```

**DESCRIPTION**

Instead of passing the notification information to a queue, a server-side SDK can make calls to ctCallback to associate a callback function to a notification.

**ctCallback** is used in the same manner as ctNotify except that the third parameter is a pointer to a callback function instead of a queue handle.

A notification setup with **ctCallback** causes the function pointed to by **cbptr** to be called (instead of a message written to a queue). This capability is only available with the server SDK, and **ctCallback** can only be called from code compiled into the server itself (using the server SDK).

The prototype for the callback function pointer is:

```c
typedef NINT (*ctCallbackPtr)(pVOID msg,NINT msglen,pVOID aux,NINT auxlen);
```

The callback function returns NO_ERROR on success and a non zero value on failure. Parameter **msg** and the optional parameter **aux** are input parameters. If both **msg** and **aux** are passed in, then they should be conceptually pasted together to form one long message.

It is important to note that as currently coded, the target file's header semaphore is held while the callback function is executed. Therefore the callback function cannot introduce pauses or delays or attempt to lock the header of the target file.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>Successful close of the queue.</td>
</tr>
<tr>
<td>446</td>
<td>BMOD_ERR</td>
<td>Bad mode: parameter out of range.</td>
</tr>
<tr>
<td>458</td>
<td>SWRT_ERR</td>
<td>Write permission not granted.</td>
</tr>
</tbody>
</table>

See the **c-tree Plus Programmer’s Reference Guide** for a complete listing of valid c-tree Plus error values.
EXAMPLE

NINT MyCallBack(pVOID msg, NINT msglen, pVOID aux, NINT auxlen)
{
    /* save callback message to disk */
    fwrite(msglen, sizeof(msglen), 1, fd);
    fwrite(msg, msglen, 1, fd);
}

if (ctCallback(ctNT_ADDREC, datno, MyCallBack, 0, 0) != NO_ERROR)
    printf("ctCallback failed\n");

LIMITATIONS

Server-side SDK only.

SEE ALSO

ctNotify
17.2.5 ctNotify

Send notification messages to a queue.

SHORT NAME
crNotify

TYPE
ISAM function.

DECLARATION
NINT ctNotify(NINT opcode, NINT objhandle, NINT qhandle, NINT contents, NINT controls)

DESCRIPTION
Using the facilities of c-tree system queues, client applications can direct c-tree server to monitor the use of a resource such as a data file and place notification messages on a queue. To use the notification process the user must open a queue using ctSysQueueOpen, use ctNotify to write the notification, ctSysQueueRead to read the notification messages and ctSysQueueClose to close the notification queue.

Parameter opcode specify which actions on the resource should be notified. The following values are available:

<table>
<thead>
<tr>
<th>Symbolic Constants</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctNT_ADDREC</td>
<td>notify that a new record was added to the data file</td>
</tr>
<tr>
<td>ctNT_DELREC</td>
<td>notify that a record was deleted</td>
</tr>
<tr>
<td>ctNT_RWTREC</td>
<td>notify that a record was modified</td>
</tr>
<tr>
<td>ctNT_CLSOBJ</td>
<td>notify that an object was closed</td>
</tr>
<tr>
<td>ctNT_ISMUPP</td>
<td>defined as</td>
</tr>
<tr>
<td></td>
<td>#define ctNT_ISMUPP (ctNT_ADDREC</td>
</tr>
</tbody>
</table>

Parameter objhandle is the datno of an ISAM data file and qhandle is a system queue handle returned by a call to ctSysQueueOpen. Parameters contents and controls are currently reserved for future use and must be set to zero.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>Successful close of the queue.</td>
</tr>
<tr>
<td>10</td>
<td>SPAC_ERR</td>
<td>INTREE parameters require too much space.</td>
</tr>
</tbody>
</table>
See the *c-tree Plus Programmer’s Reference Guide* for a complete listing of valid c-tree Plus error values.

**EXAMPLE**

```c
NINT eRet;
NINT qhandle = ctSysQueueOpen("myqueue", 0);

if (qhandle < 0)
    printf("ctSysQueueOpen failed with code %d\n", -qhandle);
else
{
    eRet = ctNotify(ctNT_ADDREC, datno, qhandle, 0, 0);
    if (eRet)
        printf("ctNotify failed with code %d\n", eRet);
}
```

**LIMITATIONS**

Client/Server mode only.

**SEE ALSO**

ctSysQueueOpen, ctSysQueueClose, ctSysQueueRead

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>FMOD_ERR</td>
<td>Operation incompatible with type of file.</td>
</tr>
<tr>
<td>90</td>
<td>NQUE_ERR</td>
<td>Could not create queue.</td>
</tr>
<tr>
<td>446</td>
<td>BMOD_ERR</td>
<td>Bad mode: parameter out of range.</td>
</tr>
<tr>
<td>458</td>
<td>SWRT_ERR</td>
<td>Write permission not granted.</td>
</tr>
<tr>
<td>514</td>
<td>CQUE_ERR</td>
<td>Queue has already been closed.</td>
</tr>
<tr>
<td>754</td>
<td>QOWN_ERR</td>
<td>Only queue creator can perform operation.</td>
</tr>
<tr>
<td>755</td>
<td>SQUE_ERR</td>
<td>A system queue is required.</td>
</tr>
</tbody>
</table>
17.2.6  **ctSysQueueClose**

Closes an existing queue.

**SHORT NAME**

ctSysQueueClose

**TYPE**

ISAM function.

**DECLARATION**

NINT ctSysQueueClose(NINT qhandle)

**DESCRIPTION**

Closes an existing queue when the queue is no longer needed.

Parameter *qhandle* is a system queue handle returned by a call to the `ctSysQueueOpen` function.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>Successful close of the queue.</td>
</tr>
<tr>
<td>90</td>
<td>NQUE_ERR</td>
<td>Could not create queue.</td>
</tr>
<tr>
<td>454</td>
<td>NSUP_ERR</td>
<td>Service not supported.</td>
</tr>
<tr>
<td>514</td>
<td>CQUE_ERR</td>
<td>Queue has already been closed.</td>
</tr>
</tbody>
</table>

See the *c-tree Plus Programmer’s Reference Guide* for a complete listing of valid c-tree Plus error values.

**EXAMPLE**

```c
NINT qhandle = ctSysQueueOpen("myqueue", 0);

if (qhandle > 0)
   ctSysQueueClose(qhandle);
```

**LIMITATIONS**

Client/Server mode only.

**SEE ALSO**

ctSysQueueOpen
17.2.7 ctSysQueueCount

Return the number of messages waiting in the queue.

SHORT NAME
ctSysQueueCount

TYPE
ISAM function.

DECLARATION

NINT ctSysQueueCount(NINT qhandle)

DESCRIPTION

ctSysQueueCount can be used to obtain the number of messages waiting in the system queue.
Parameter qhandle is a system queue handle returned by a call to ctSysQueueOpen.

RETURN

ctSysQueueCount returns the number of messages in the queue, or a negative number on error.

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-90</td>
<td>NQUE_ERR</td>
<td>Could not create queue.</td>
</tr>
<tr>
<td>-514</td>
<td>CQUE_ERR</td>
<td>Queue has already been closed.</td>
</tr>
</tbody>
</table>

See the c-tree Plus Programmer’s Reference Guide for a complete listing of valid c-tree Plus error values.

EXAMPLE

/* check the number of messages on a queue */
NINT count = ctSysQueueCount(qhandle);
if (count >= 0)
   printf("There are %d messages on my queue\n", count);
else
   printf("ctSysQueueCount failed with code %d\n", -count);

LIMITATIONS

Client/Server mode only.

SEE ALSO
tcSysQueueRead
17.2.8  **ctSysQueueLIFOWrite**

Add new data at the beginning of a queue.

**SHORT NAME**

ctSysQueueLIFOWrite

**TYPE**

ISAM function.

**DECLARATION**

```
NINT ctSysQueueLIFOWrite(NINT qhandle, pVOID message, NINT msglen)
```

**DESCRIPTION**

Data is placed at the beginning of the queue by invoking `ctSysQueueLIFOWrite`. This is similar to a stack operation.

Parameter `qhandle` is a system queue handle returned by a call to `ctSysQueueOpen`. Parameter `message` is a pointer to a block of memory containing arbitrary data to be placed on the queue and `msglen` indicates how many bytes are pointed to by `message`.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>Successful write to the queue.</td>
</tr>
<tr>
<td>90</td>
<td>NQUE_ERR</td>
<td>Could not create queue.</td>
</tr>
<tr>
<td>92</td>
<td>QMRT_ERR</td>
<td>Queue memory error during write.</td>
</tr>
<tr>
<td>446</td>
<td>BMOD_ERR</td>
<td>Bad mode: parameter out of range.</td>
</tr>
<tr>
<td>454</td>
<td>NSUP_ERR</td>
<td>Service not supported.</td>
</tr>
<tr>
<td>514</td>
<td>CQUE_ERR</td>
<td>Queue has already been closed.</td>
</tr>
<tr>
<td>758</td>
<td>QNOT_ERR</td>
<td>Only notifications to queue.</td>
</tr>
<tr>
<td>759</td>
<td>QUIN_ERR</td>
<td>Wrong queue instance.</td>
</tr>
</tbody>
</table>

See the *c-tree Plus Programmer’s Reference Guide* for a complete listing of valid c-tree Plus error values.

**EXAMPLE**

```
NINT eRet = ctSysQueueLIFOWrite(qhandle, "My message", 10);

if (eRet)
    printf("ctSysQueueLIFOWrite failed with code %d\n", eRet);
```
LIMITATIONS

Client/Server mode only.

SEE ALSO

cSysQueueWrite
17.2.9  **ctSysQueueMlen**

Retrieves the length of the next available message in the queue.

**SHORT NAME**  
ctSysQueueMlen

**TYPE**  
ISAM function.

**DECLARATION**  
NINT  ctSysQueueMlen(NINT qhandle, LONG timeout)

**DESCRIPTION**  
ctSysQueueMlen returns the length of the next available message in the specified queue.

Parameter *qhandle* is a system queue handle returned by a call to ctSysQueueOpen. Parameter *timeout* specifies a millisecond time that ctSysQueueMlen will block if the queue is empty. A timeout value of ctWAITDFOREVER will cause ctSysQueueMlen to block until data is available in the queue.

Use ctSysQueueMlen to find out the appropriate size of a buffer before calling ctSysQueueRead.

**RETURN**  
ctSysQueueMlen returns the length of the next available message, or a negative value on error.

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-7</td>
<td>TUSR_ERR</td>
<td>Terminate user.</td>
</tr>
<tr>
<td>-90</td>
<td>NQUE_ERR</td>
<td>Could not create queue.</td>
</tr>
<tr>
<td>-514</td>
<td>CQUE_ERR</td>
<td>Queue has already been closed.</td>
</tr>
</tbody>
</table>

See the c-tree Plus Programmer’s Reference Guide for a complete listing of valid c-tree Plus error values.

**EXAMPLE**  
NINT len = ctSysQueueMlen(qhandle, ctWAITFOREVER);

    if (len >= 0)
        printf("The next message length is %d bytes\n", len);
    else
        printf("ctSysQueueMlen failed with code %d\n", -len);

**LIMITATIONS**  
Client/Server mode only.

**SEE ALSO**  
ctSysQueueRead
17.2.10 ctSysQueueOpen

Open or create a new queue.

**SHORT NAME**
ctSysQueueOpen

**TYPE**
ISAM function

**DECLARATION**

```c
NINT  ctSysQueueOpen(pTEXT qname, NINT qmode)
```

**DESCRIPTION**

*ctSysQueueOpen* opens an existing queue or creates a new queue.

Parameter *qname* identify the queue to be opened. If the queue specified by *qname* does not exist, a new queue is created. Parameter *qmode* is currently reserved for future use and should be set to zero.

When the queue is no longer needed, it must be closed by calling the close function *ctSysQueueClose*.

**RETURN**

*ctSysQueueOpen* returns a queue handle on success, or a negative value on error.

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-454</td>
<td>NSUP_ERR</td>
<td>Service not supported.</td>
</tr>
</tbody>
</table>

See the *c-tree Plus Programmer's Reference Guide* for a complete listing of valid c-tree Plus error values.

**EXAMPLE**

```c
NINT qhandle = ctSysQueueOpen("myqueue", 0);

if (qhandle < 0)
    printf("ctSysQueueOpen failed with code %d\n", -qhandle);
```

**LIMITATIONS**

Client/Server mode only.

**SEE ALSO**

*ctSysQueueClose*
17.2.11 \texttt{ctSysQueueRead}

Read data from the queue.

**SHORT NAME**

\texttt{ctSysQueueRead}

**TYPE**

ISAM function.

**DECLARATION**

\begin{verbatim}
NINT  ctSysQueueRead(NINT qhandle, pVOID buffer, NINT buflen, LONG timeout)
\end{verbatim}

**DESCRIPTION**

Data is read from the queue by calling \texttt{ctSysQueueRead}.

Parameter \texttt{qhandle} is a system queue handle returned by a call to \texttt{ctSysQueueOpen}, while \texttt{timeout} specifies a millisecond time that \texttt{ctSysQueueRead} will block waiting for data. A \textit{timeout} of \texttt{ctWAITFOREVER} will cause \texttt{ctSysQueueRead} to block until data is available in the queue. Parameter \texttt{buffer} is a pointer to a memory block large enough to hold the next message in the queue, and \texttt{buflen} indicates the size in bytes of \texttt{buffer}.

\texttt{ctSysQueueMlen} may be used to retrieve the next message length, while \texttt{ctSysQueueCount} may be used to retrieve the number of messages waiting in the queue.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>Successful function.</td>
</tr>
<tr>
<td>90</td>
<td>NQUE_ERR</td>
<td>Could not create queue.</td>
</tr>
<tr>
<td>514</td>
<td>CQUE_ERR</td>
<td>Queue has already been closed.</td>
</tr>
<tr>
<td>638</td>
<td>TQUE_ERR</td>
<td>Queue message truncated to fit.</td>
</tr>
</tbody>
</table>

See the \textit{c-tree Plus Programmer’s Reference Guide} for a complete listing of valid c-tree Plus error values.

**EXAMPLE**

\begin{verbatim}
TEXT msg[128];
NINT eRet = ctSysQueueRead(qhandle, msg, sizeof(msg), ctWAITFOREVER);

if (eRet)
    printf("ctSysQueueRead failed with code %d\n", eRet);
\end{verbatim}

**LIMITATIONS**

Client/Server mode only.

**SEE ALSO**

\texttt{ctSysQueueMlen}, \texttt{ctSysQueueCount}
17.2.12 ctSysQueueWrite

Add new data at the end of a queue.

**SHORT NAME**

ctSysQueueWrite

**TYPE**

ISAM function.

**DECLARATION**

NINT ctSysQueueWrite(NINT qhandle, PVOID message, NINT msglen)

**DESCRIPTION**

Data is placed at the end of the queue by invoking `ctSysQueueWrite`.

Parameter `qhandle` is a system queue handle returned by a call to `ctSysQueueOpen`. Parameter `message` is a pointer to a block of memory containing arbitrary data to be placed on the queue and `msglen` indicates how many bytes are pointed to by `message`.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>Successful write to the queue.</td>
</tr>
<tr>
<td>90</td>
<td>NQUE_ERR</td>
<td>Could not create queue.</td>
</tr>
<tr>
<td>92</td>
<td>QMRT_ERR</td>
<td>Queue memory error during write.</td>
</tr>
<tr>
<td>446</td>
<td>BMOD_ERR</td>
<td>Bad mode: parameter out of range.</td>
</tr>
<tr>
<td>454</td>
<td>NSUP_ERR</td>
<td>Service not supported.</td>
</tr>
<tr>
<td>514</td>
<td>CQUE_ERR</td>
<td>Queue has already been closed.</td>
</tr>
<tr>
<td>758</td>
<td>QNOT_ERR</td>
<td>Only notifications to queue.</td>
</tr>
<tr>
<td>759</td>
<td>QUIN_ERR</td>
<td>Wrong queue instance.</td>
</tr>
</tbody>
</table>

See the *c-tree Plus Programmer’s Reference Guide* for a complete listing of valid c-tree Plus error values.

**EXAMPLE**

```c
NINT eRet = ctSysQueueWrite(qhandle, "My message", 10);
if (eRet)
    printf("ctSysQueueWrite failed with error %d\n", eRet);
```

**LIMITATIONS**

Client/Server mode only.

**SEE ALSO**

ctSysQueueLIFOWrite
17.2.13  FirstInRange

Read the first data record in a range

SHORT NAME  
FRSRNG

TYPE  
ISAM function

DECLARATION  
COUNT FirstInRange(COUNT keyno, pVOID recptr);

DESCRIPTION  
Read the first data record in a range established by a call to AllocateRange. If successful, the record becomes the current ISAM record for the associated data file. A successful FirstInRange defines a current key value set, and subsequent calls to NextInRange or PreviousInRange will read the other records in the range.

If the data file has variable-length records, only the fixed-length portion of the record is actually read. You can use ReReadVRecord to retrieve the whole record, including the variable length portion. You can use FirstInVRange to read the whole variable length record with one function call.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>No error occurred.</td>
</tr>
</tbody>
</table>


SEE ALSO  
AllocateRange, FreeRange, LastInRange, NextInRange and PreviousInRange
17.2.14 FreeRange

Reset and free allocated buffers for range operation

SHORT NAME
FRERNG

TYPE
ISAM function

DECLARATION
COUNT FreeRange(COUNT keyno)

DESCRIPTION
FreeRange should be called when a range associated with an index is no longer needed.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>No error occurred.</td>
</tr>
</tbody>
</table>


EXAMPLE

```c
if (FreeRange(1))
    printf("\nFreeRange error = %d", uerr_cod);
```

SEE ALSO
AllocateRange, FirstInRange, LastInRange, NextInRange and PreviousInRange
17.2.15 LastInRange

Read the last data record in a range

SHORT NAME
LSTRNG

TYPE
ISAM function

DECLARATION
COUNT LastInRange(COUNT keyno, pVOID recptr)

DESCRIPTION
Read the last data record in a range established by a call to AllocateRange. If successful, the record becomes the current ISAM record for the associated data file. A successful LastInRange defines a current key value set, and subsequent calls to NextInRange or PreviousInRange will read the other records in the range.

If the data file has variable-length records, only the fixed-length portion of the record is actually read. You can use ReReadVRecord to retrieve the whole record, including the variable length portion. You can use LastInVRange to read the whole variable length record with one function call.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>No error occurred.</td>
</tr>
</tbody>
</table>


SEE ALSO
AllocateRange, FreeRange, FirstInRange, NextInRange and PreviousInRange
17.2.16  NextInRange

Read the next data record in a range

SHORT NAME

NXTRNG

TYPE

ISAM function

DECLARATION

COUNT NextInRange(COUNT keyval, pVOID recptr)

DESCRIPTION

Read the next data record in a range established by a call to AllocateRange. If successful, the record becomes the current ISAM record for the associated data file. A successful NextInRange defines a current key value set, and subsequent calls to NextInRange or PreviousInRange will read the other records in the range.

If the data file has variable-length records, only the fixed-length portion of the record is actually read. You can use ReReadVRecord to retrieve the whole record, including the variable length portion. You can use NextInVRange to read the whole variable length record with one function call.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>No error occurred.</td>
</tr>
</tbody>
</table>


SEE ALSO

AllocateRange, FreeRange, FirstInRange, LastInRange and PreviousInRange
17.2.17  PreviousInRange

Read the previous record in a range

SHORT NAME
PRVRNG

TYPE
ISAM function

DECLARATION
COUNT PreviousInRange(COUNT keyval, pVOID recptr)

DESCRIPTION
Read the previous data record in a range established by a call to AllocateRange. If successful, the record becomes the current ISAM record for the associated data file. A successful PreviousInRange defines a current key value set, and subsequent calls to NextInRange or PreviousInRange will read the other records in the range.

If the data file has variable-length records, only the fixed-length portion of the record is actually read. You can use ReReadVRecord to retrieve the whole record, including the variable length portion. You can use PreviousInVRange to read the whole variable length record with one function call.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO_ERROR</td>
<td>No error occurred.</td>
</tr>
</tbody>
</table>


SEE ALSO
AllocateRange, FreeRange, FirstInRange, LastInRange and NextInRange
### 17.3 c-treeDB C API Functions

This section provides detailed function descriptions for c-treeDB C API functions added or updated with this release.

The following functions have been added:

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ctdbEstimateSpan</code></td>
<td>Estimate an approximate number of records between two key target values.</td>
</tr>
<tr>
<td><code>ctdbGetDatno</code></td>
<td>Retrieves the table data file number.</td>
</tr>
<tr>
<td><code>ctdbGetIdxno</code></td>
<td>Retrieves the index file number from an index handle.</td>
</tr>
<tr>
<td><code>ctdbGetIdxnoByName</code></td>
<td>Retrieves the index file number from a table handle given the index name.</td>
</tr>
<tr>
<td><code>ctdbGetIdxnoByNumber</code></td>
<td>Retrieves the index file number from a table handle given the c-treeDB index number.</td>
</tr>
<tr>
<td><code>ctdbGetPathPrefix</code></td>
<td>Get the current path prefix.</td>
</tr>
<tr>
<td><code>ctdbIsRecordRangeOn</code></td>
<td>Indicate if an index range operation is active for this record handle.</td>
</tr>
<tr>
<td><code>ctdbRecordAtPercentile</code></td>
<td>Find a record located at about the given percentile value.</td>
</tr>
<tr>
<td><code>ctdbRecordRangeOff</code></td>
<td>Terminate a record index range operation established by <code>ctdbRecordRangeOn</code>.</td>
</tr>
<tr>
<td><code>ctdbRecordRangeOn</code></td>
<td>Establish a new index range on a record handle.</td>
</tr>
<tr>
<td><code>ctdbRemoveTable</code></td>
<td>Removes a table from the database.</td>
</tr>
<tr>
<td><code>ctdbSetCurrentNodeName</code></td>
<td>Set the client-side node name.</td>
</tr>
<tr>
<td><code>ctdbSetPathPrefix</code></td>
<td>Set a new path prefix.</td>
</tr>
<tr>
<td><code>ctdbSwitchContext</code></td>
<td>Force an ISAM context switch.</td>
</tr>
<tr>
<td><code>ctdbSwitchInstance</code></td>
<td>Force a c-tree Plus instance switch.</td>
</tr>
<tr>
<td><code>CTBase::SwitchInstance</code></td>
<td>Force a c-tree Plus instance switch.</td>
</tr>
<tr>
<td><code>CTIndex::GetIdxno</code></td>
<td>Retrieves the table index file number from an index object.</td>
</tr>
<tr>
<td><code>CTRecord::SwitchContext</code></td>
<td>Force an ISAM context switch.</td>
</tr>
<tr>
<td><code>CTSession::GetPathPrefix</code></td>
<td>Get the current path prefix.</td>
</tr>
</tbody>
</table>
### New Function Descriptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTSession::SetPathPrefix</td>
<td>Set a new path prefix.</td>
</tr>
<tr>
<td>CTTable::GetDatno</td>
<td>Retrieves the table data file number.</td>
</tr>
<tr>
<td>CTTable::GetIdxno</td>
<td>Retrieves the table index file number given an index name, or c-treeDB index number, from a table object.</td>
</tr>
<tr>
<td>CTTable::Remove</td>
<td>Removes a table from the database.</td>
</tr>
</tbody>
</table>
17.3.1  **ctdbEstimateSpan**

Estimate an approximate number of records between two key target values.

**DECLARATION**

```
LONG ctdbEstimateSpan(CTHANDLE Handle, pVOID key1, pVOID key2);
```

**DESCRIPTION**

*Handle* is a record handle and *key1* and *key2* are two key target values used to obtain the estimated number of records. If *ctdbEstimateSpan* return 0, use *ctdbGetError* function to retrieve the error code. If *ctdbEstimateSpan* return 0 and *ctdbGetError* return CTDBRET_OK then there are no records between the two key values supplied.

The estimation is based on the record handle current index. The current index may be changed by calling *ctdbSetDefaultIndex*. The table must have at least one index to be able to use this function.

*ctdbEstimateSpan*, which is based on c-tree low level function *EstimateKeySpan*, does not traverse the index to compute the values. Instead, it makes about ten calls to c-tree low level function *KeyAtPercentile* to determine the relative location of the target values.

The key target values used by *ctdbEstimateSpan* can be created using *ctdbBuildTargetKey*.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td>ctdbRecordAtPercentile return CTDBRET_OK on success or c-treeDB SDK error code on failure.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer’s Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

Lets suppose that a student table has two fields (*Name* - CT_FSTRING and *Age* - CT_INT2) and one index on field *Age*. The sample function estimate the number of students with ages between 10 and 12:

```c
LONG EstimateStudents(CTHANDLE hRecord)
{
    TEXT key1[32];
    TEXT key2[32];

    /* build the first key for age = 10 years */
    ctdbClearRecord(hRecord);
    ctdbSetFieldAsSigned(hRecord, 1, 10);
    if (ctdbBuildTargetKey(hRecord, CTFIND_EQ, key1, 32))
        return 0;
    /* build the second key for age = 12 years */
    ctdbClearRecord(hRecord);
    ctdbSetFieldAsSigned(hRecord, 1, 12);
```
if (ctdbBuildTargetKey(hRecord, CTFIND_EQ, key2, 32))
  return 0;
/* estimate the number of students */
return ctdbEstimateSpan(hRecord, key1, key2);
}

SEE ALSO
ctdbSetDefaultIndex, ctdbBuildTargetKey.
17.3.2 ctdbGetDatno

Retrieve the table data file number.

**DECLARATION**

NINT ctdbGetDatno(CTHANDLE Handle)

**DESCRIPTION**

Retrieve the table data file number. Handle must be a table handle, or a handle that can be converted into a table handle such as a record, segment, index or field handle. Return the table datno on success or -1 on failure. If ctdbGetDatno function return -1, the error code can be retrieved by calling ctdbGetError function.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td>ctdbGetDatno failed. You can retrieve the error code by calling ctdbGetError function.</td>
</tr>
</tbody>
</table>


**EXAMPLE**

```c
CTDBRET DeleteTable(CTHANDLE hSession, pTEXT tablename) {
    CTDBRET Retval = CTDBRET_OK;
    CTHANDLE hTable = ctdbAllocTable(hSession);

    if (hTable)
    {
        /* open the table exclusive */
        if ((Retval = ctdbOpenTable(hTable, tablename, CTOPEN_EXCLUSIVE)) != CTDBRET_OK)
        {
            return Retval;
        }
        /* delete a file */
        if ((Retval = (CTDBRET)DeleteRFile((COUNT)ctdbGetDatno(hTable)) != CTDBRET_OK)
        {
            return Retval;
        }
    }
    else
    {
        Retval = CTDBRET_NOMEMORY;
        return Retval;
    }
}
```

**SEE ALSO**

cmdbSwitchInstance, ctdbSwitchContext, ctdbGetIdxno, ctdbGetIdxnoByName, ctdbGetIdxno-ByNumber, ctdbGetError
17.3.3  ctdbGetIdxno

Retrieve the index file number from a index handle.

DECLARATION

NINT ctdbGetIdxno(CTHANDLE Handle)

DESCRIPTION

Retrieve the index file number to be used with c-tree ISAM or low-level index functions.

Handle must be an index or segment handle.

Return the index number on success or -1 on failure. If ctdbGetIdxno function return -1, the error code can be retrieved by calling ctdbGetError function.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td>ctdbGetIdxno failed. You can retrieve the error code by calling ctdbGetError.</td>
</tr>
</tbody>
</table>


EXAMPLE

/* retrieve the first key of first index */
TEXT keyval[256];
COUNT idxno = (COUNT)ctdbGetIdxno(ctdbGetIdnex(hTable, 0));
if (FirstKey(idxno, 0), keyval)
    printf("FirstKey failed\n");

SEE ALSO

ctdbSwitchInstance, ctdbSwitchContext, ctdbGetDatno, ctdbGetIdxnoByName, ctdbGetIdxnoByNumber, ctdbGetError
17.3.4  **ctdbGetIdxnoByName**

Retrieve the index file number from a table handle.

**DECLARATION**

\[
\text{NINT ctdbGetIdxnoByName(CTHANDLE Handle, pTEXT indexname)}
\]

**DESCRIPTION**

Retrieve the index file number to be used with c-tree ISAM or low-level index functions.

Handle is a table handle. indexname is a string containing the index name.

Return the index number on success or -1 on failure. If ctdbGetIdxnoByNumber return -1, the error code can be retrieved by calling ctdbGetError function.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td>ctdbGetIdxnoByNumber failed. You can retrieve the error code by calling ctdbGetError function.</td>
</tr>
</tbody>
</table>


**EXAMPLE**

/* retrieve the first key of first index */
TEXT keyval[256];
COUNT idxno = (COUNT)ctdbGetIdxnoByName(hTable, "indexname");
if (FirstKey(idxno, 0), keyval)
    printf("FirstKey failed\n");

**SEE ALSO**

ctdbSwitchInstance, ctdbSwitchContext, ctdbGetDatno, ctdbGetIdxno, ctdbGetIdxnoByNumber, ctdbGetError
17.3.5  **ctdbGetIdxnoByNumber**

Retrieve the index file number from a table handle.

**DECLARATION**

```
NINT ctdbGetIdxnoByNumber(CTHANDLE Handle, NINT index)
```

**DESCRIPTION**

Retrieve the index file number to be used with c-tree ISAM and low-level index functions.

Handle is a table handle. index is a c-treeDB index number. The first c-treeDB index number is zero.

Return the index number on success or -1 on failure. If ctdbGetIdxnoByNumber return -1, the error code can be retrieved by calling ctdbGetError function.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td>ctdbGetIdxnoByNumber failed. You can retrieve the error code by calling ctdbGetError function.</td>
</tr>
</tbody>
</table>


**EXAMPLE**

```
/* retrieve the first key of first index */
TEXT keyval[256];
COUNT idxno = (COUNT)ctdbGetIdxnoByNumber(hTable, 0);
if (FirstKey(idxno, 0), keyval)
  printf("FirstKey failed\n");
```

**SEE ALSO**

ctdbSwitchInstance, ctdbSwitchContext, ctdbGetDatno, ctdbGetIdxno, ctdbGetIdxnoByName, ctdbGetError
### 17.3.6 ctdbGetPathPrefix

Returns the client-side path prefix.

**DECLARATION**

```c
pTEXT ctdbGetPathPrefix(CTHANDLE hSession);
```

**DESCRIPTION**

A path prefix can be set anytime after the session handle is allocated. If a path prefix is set before a session logon, the new path prefix will affect the location of the session dictionary file. If a path prefix is set after a session logon, but before a database connect, then the path prefix affects only the database dictionary and any tables that are manipulated during that session.

A path prefix can be removed at any time by setting a NULL value for the path prefix. You can use ctdbGetPathPrefix to check if a path prefix is set or not. If ctdbGetPathPrefix returns NULL, then no path prefix is set.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td>ctdbGetPathPrefix return CTDBRET_OK on success or c-treeDB SDK error code on failure.</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>ctdbGetPathPrefix returns NULL when no path prefix is set.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer's Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

```c
/* get the current path prefix */
YourPathPrefix = ctdbGetPathPrefix(AnyHandle);
if (CurrentPathPrefix == NULL)
    printf("ctdbGetPathPrefix() returned no path.\n");
```

**SEE ALSO**

ctdbSetPathPrefix.
17.3.7 **ctdbIsRecordRangeOn**

Indicate if an index range operation is active for this record handle.

**DECLARATION**

CTBOOL ctdbIsRecordRangeOn(CTHANDLE Handle);

**DESCRIPTION**

**ctdbIsRecordRangeOn** return YES if a index range operation is active for this record handle, or NO is no index range is active.

- *Handle* [IN] is a record handle.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO</td>
<td>No index range operation is active</td>
</tr>
<tr>
<td>1</td>
<td>YES</td>
<td>Index range operation is active</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer’s Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

```c
/* display all records where age is greater than 65 */
void DisplayAll(CTHANDLE hRecord)
{
    UTEXT lRange[32];
    NINT op[1] = {CTIX_GT};
    NINT fldno = ctdbGetFieldNumberByName(hHandle, "age");
    CTDBRET eRet;

    ctdbClearRecord(hRecord);
    ctdbSetFieldAsSigned(hRecord, fldno, 65);
    ctdbSetDefaultIndex(hRecord, 0);
    ctdbBuildTargetKey(hRecord, CTFIND_EQ, lRange, 32);
    eRet = ctdbRecordRangeOn(hRecord, 1, lRange, NULL, op);
    if (eRet == CTDBRET_OK)
    {
        eRet = ctdbFirstRecord(hRecord);
        while (eRet == CTDBRET_OK)
        {
            TEXT str[128];
            ctdbGetFieldAsString(hRecord, 0, str, sizeof(str));
            printf("%s\n", str);
            eRet = ctdbNextRecord(hRecord);
        }
    }
    if (ctdbIsRecordRangeOn(hRecord))
        ctdbRecordRangeOff(hRecord);
}
```

**SEE ALSO**

ctdbRecordRangeOn, ctdbRecordRangeOff
17.3.8  ctdbRecordAtPercentile

Find a record located at about the given percentile value.

DECLARATION

CTDBRET ctdbRecordAtPercentile(CTHANDLE Handle, NINT percent);

DESCRIPTION

ctdbRecordAtPercentile read the record located at, approximately, the given percentile value.

Handle is a record handle and percent indicate the percentile value. The valid values for percent are from 0 to 100, indicating 0% to 100%. ctdbRecordAtPercentile return CTDBRET_OK on success.

The record is located using the record handle current index. You may select a new current index by calling ctdbSetDefaultIndex function. The table must have at least one index to be able to use this function.

The record returned is an approximation location indicated by the percentual value passed to ctdbRecordAtPercentile.

ctdbRecordAtPercentile, which is based on c-tree low level function KeyAtPercentile, and it is very efficient since it does not traverse all of the key values in order to determine the record located at the specified percentile. However, ctdbRecordAtPercentile is only an approximation since it assumes that key values are uniformly distributed among all of the b-tree leaf nodes.

ctdbRecordAtPercentile may be used to support scroll bar positioning, found in many GUI windowing environments, in the cases when the position must be maintained in key sequential order.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td>ctdbRecordAtPercentile return CTDBRET_OK on success or c-treeDB SDK error code on failure.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the c-treeDB C API Developer's Guide for a complete listing of valid c-treeDB error codes and return values.

EXAMPLE

/* display the record at 50% of table */
if (ctdbRecordAtPercentile(hRecord, 50) == CTDBRET_OK)
{
    DisplayTheRecord(hRecord);
}
else
{
    printf("Failed with error %d\n", ctdbGetError(hRecord));
}

SEE ALSO

ctdbSetDefaultIndex
17.3.9  

**ctdbRecordRangeOff**

Terminate a record index range operation established by `ctdbRecordRangeOn`.

**DECLARATION**

```c
CTDBRET ctdbRecordRangeOff(CTHANDLE Handle);
```

**DESCRIPTION**

- `ctdbRecordRangeOff` terminate a range operation.
  - `Handle [IN]` is a record handle.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td><code>ctdbRecordRangeOff</code> return <code>CTDBRET_OK</code> on success or c-treeDB SDK error code on failure.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer’s Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

```c
/* display all records where age is greater than 65 */
void DisplayAll(CTHANDLE hRecord)
{
    UTEXT lRange[32];
    NINT op[1] = {CTIX_GT};
    NINT fldno = ctdbGetFieldNumberByName(hHandle, "age");
    CTDBRET eRet;

    ctdbClearRecord(hRecord);
    ctdbSetFieldAsSigned(hRecord, fldno, 65);
    ctdbSetDefaultIndex(hRecord, 0);
    ctdbBuildTargetKey(hRecord, CTFIND_EQ, lRange, 32);
    eRet = ctdbRecordRangeOn(hRecord, 1, lRange, NULL, op);
    if (eRet == CTDBRET_OK)
    {
        eRet = ctdbFirstRecord(hRecord);
        while (eRet == CTDBRET_OK)
        {
            TEXT str[128];
            ctdbGetFieldAsString(hRecord, 0, str, sizeof(str));
            printf("%s\n", str);
            eRet = ctdbNextRecord(hRecord);
        }
    }
    if (ctdbIsRecordRangeOn(hRecord))
    ctdbRecordRangeOff(hRecord);
}
```

**SEE ALSO**

`ctdbRecordRangeOn`, `ctdbIsRecordRangeOn`
17.3.10 **ctdbRecordRangeOn**

Establish a new index range on a record handle.

**DECLARATION**

```c
CTDBRET ctdbRecordRangeOn(CTHANDLE Handle, NINT SegCount,
                          pVOID lRange, pVOID uRange, pNINT operators);
```

**DESCRIPTION**

`ctdbRecordRangeOn` establishes a new range based on the key segment values passed on `lRange` and `uRange` buffers, and the operators for each segment. Once the range is set, use `ctdbFirstRecord`, `ctdbNextRecord`, `ctdbPrevRecord` and `ctdbLastRecord` to navigate the records in the specified range. The range is set for all index entries that are situated between the lower bounds and upper bounds values. The segment values are stored in `lRange` and `uRange` buffers in the same order and type of the index segment definition. If a previous range exists for this index, the previous range is released and the new range is established. Ranges take precedence over sets. If a record handle has a set established, record from a range will fetched instead of records from a range. Once the range is terminated, the records from a set is established.

- **Handle** [IN] is the record handle.
- **SegCount** [IN] indicates the number of index segments values that should be used for setting the range, and the number of operators, since there must be one operator for each key segment in `lRange` and/or `uRange`.
- **lRange** [IN] is a buffer with the lower range segment values. Use the function `ctdbBuildTargetKey` to build the `lRange` buffer.
- **uRange** [IN] is a buffer with the upper range segment values. Use the function `ctdbBuildTargetKey` to build the `uRange` buffer.
- **operators** [IN] operators is an array of operators. There must be one operator for each key segment in `lRange` and/or `uRange`. The operators `CTIX_EQ`, `CTIX_NE`, `CTIX_GT`, `CTIX_GE`, `CTIX_LE`, `CTIX_LT` are open ended and use only the `lRange` buffer for range values and the equivalent key segment in `uRange` is ignored and maybe set to null (ascii `\0` values). The operators `CTIX_BET`, `CTIX_BET_IE`, `CTIX_BET_EI`, `CTIX_BET_EE` and `CTIX_NOTBET` use both `lRange` and `uRange` buffers to establish the lower and upper bound values.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td><code>ctdbRecordRangeOn</code> return CTDBRET_OK on success or c-treeDB SDK error code on failure.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer’s Guide* for a complete listing of valid c-treeDB error codes and return values.
EXAMPLE

/* display all records where age is greater than 65 */
void DisplayAll(CTHANDLE hRecord)
{
    UTEXT lRange[32];
    NINT op[1] = {CTIX_GT};
    NINT fldno = ctdbGetFieldNumberByName(hHandle, "age");
    CTDBRET eRet;

    ctdbClearRecord(hRecord);
    ctdbSetFieldAsSigned(hRecord, fldno, 65);
    ctdbSetDefaultIndex(hRecord, 0);
    ctdbBuildTargetKey(hRecord, CTFIND_EQ, lRange, 32);
    eRet = ctdbRecordRangeOn(hRecord, 1, lRange, NULL, op);
    if (eRet == CTDBRET_OK)
    {
        eRet = ctdbFirstRecord(hRecord);
        while (eRet == CTDBRET_OK)
        {
            TEXT str[128];
            ctdbGetFieldAsString(hRecord, 0, str, sizeof(str));
            printf("%s\n", str);
            eRet = ctdbNextRecord(hRecord);
        }
    }
    if (ctdbIsRecordRangeOn(hRecord))
        ctdbRecordRangeOff(hRecord);
}

SEE ALSO

ctdbRecordRangeOff, ctdbIsRecordRangeOn
17.3.11  ctdbRemoveTable

Removes any c-treeDB table.

DECLARATION

CTDBRET ctdbDECL ctdbRemoveTable(CTHANDLE Handle);

DESCRIPTION

The ctdbRemoveTable function allows any table to be deleted, including tables that are not members of a database.

Handle is a table handle. The ctdbRemoveTable function deletes a c-tree data file and the associated index files from disk. If the table was opened under a database handle, the table is closed and ctdbDeleteTable is called. If the handle is not active, the table is opened exclusive and then deleted. If the table handle is not active, you must set the path, file extension and password for the handle before calling ctdbRemoveTable.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td>ctdbRemoveTable returns CTDBRET_OK on success or c-treeDB SDK error code on failure.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the c-treeDB C API Developer's Guide for a complete listing of valid c-treeDB error codes and return values.

EXAMPLE

/* delete a c-treeDB table */
if (ctdbRemoveTable(MyHandle) != CTDBRET_OK)
    printf("ctdbRemoveTable Failed!\n");

SEE ALSO

ctdbDeleteTable.
17.3.12  **ctdbSetCurrentNodeName**

Set the client-side node name.

**DECLARATION**

```
CTDBRET ctdbSetCurrentNodeName(CTHANDLE Handle, pTEXT NodeName);
```

**DESCRIPTION**

When monitoring c-tree Server attached users, well written c-treeDB client applications should "register" their workstation (node) name using the standard c-tree **SETNODE**() (**SetNode**-Name()) function call.

**ctdbSetCurrentNodeName** set a client-side node name. **Handle** is a session handle and **NodeName** is an string specifying the node name. The specified node name appears in the CTADMN utility under the option for "list clients". **ctdbSetCurrentNodeName** must be called after a successful logon.

This functionality is available only with the c-tree Server. A call to **ctdbSetCurrentNodeName** on non-server environment will always return CTDBRET_OK and the node name is ignored.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td>ctdbRecordAtPercentile return CTDBRET_OK on success or c-treeDB SDK error code on failure.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer’s Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

```c
/* set the c-tree Server node name for this workstation */
if (ctdbSetCurrentNodeName(AnyHandle, "This is my node") != CTDBRET_OK)
    printf("ctdbSetCurrentNodeName() failed\n");
```

**SEE ALSO**

ctdbSetDefaultIndex, ctdbBuildTargetKey.
17.3.13  `ctdbSetPathPrefix`

Set the client-side path prefix.

**DECLARATION**

```c
CTDBRET ctdbSetPathPrefix(CTHANDLE hSession, pTEXT pathPrefix);
```

**DESCRIPTION**

A path prefix can be set anytime after the session handle is allocated. If a path prefix is set before a session logon, the new path prefix will affect the location of the session dictionary file. If a path prefix is set after a session logon, but before a database connect, then the path prefix affects only the database dictionary and any tables that are manipulated during that session.

A path prefix can be removed at any time by setting a NULL value for the path prefix. You can use `ctdbGetPathPrefix` to check if a path prefix is set or not. If `ctdbGetPathPrefix` returns NULL, then no path prefix is set.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td><code>ctdbRecordAtPercentile</code> return CTDBRET_OK on success or c-treeDB SDK error code on failure.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer's Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

```c
/* set the current path prefix */
if (ctdbSetPathPrefix(AnyHandle, "c:\myDirectory") != CTDBRET_OK)
    printf("ctdbSetPathPrefix() failed\n");

/* clear the current path prefix */
if (ctdbSetPathPrefix(AnyHandle, NULL) != CTDBRET_OK)
    printf("ctdbSetPathPrefix() failed\n");
```

**SEE ALSO**

`ctdbGetPathPrefix`. 
17.3.14  ctdbSwitchContext

Force a ISAM context switch.

DECLARATION

CTDBRET ctdbSwitchContext(CTHANDLE Handle)

DESCRIPTION

Force a switch to the c-tree Plus ISAM context indicated by the record handle. Each record handle has its own c-tree ISAM context id.

When most c-treeDB record handling functions are called, they automatically perform a c-tree ISAM context switch. ctdbSwitchContext may be useful before calling specific c-tree ISAM or low level calls to make sure the correct ISAM context is active before making those calls.

Handle is a record handle. The handle must be a record handle. No other type of c-treeDB handle will be acceptable.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td>Successful function.</td>
</tr>
</tbody>
</table>


EXAMPLE

```c
/* force a context switch */
if (ctdbSwitchContext(hRecord) != CTDBRET_OK)
    printf("ctdbSwitchContext failed\n");
/* call ResetRecord */
if (ResetRecord((COUNT)ctdbGetDatno(hRecord), SWTCURI))
    printf("ResetRecord failed\n");
```

SEE ALSO

ctdbSwitchInstance, ctdbGetDatno, ctdbGetIdxno, ctdbGetIdxnoByName, ctdbGetIdxno-ByNumber
17.3.15  ctdbSwitchInstance

Force a c-tree Plus instance switch.

DECLARATION

CTDBRET ctdbSwitchInstance(CTHANDLE Handle)

DESCRIPTION

Force a switch to the c-tree Plus instance indicated by the Session handle. Each session handle has its unique c-tree instance id.

When most c-treeDB C functions are called, they automatically perform a c-tree instance switch. ctdbSwitchInstance may be useful before calling specific c-tree ISAM or low level calls to make sure the correct instance is active before making those calls.

Handle is a Session handle. You can pass any c-treeDB handle to ctdbSwitchInstance.

RETURN

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTDBRET_OK</td>
<td>Successful function.</td>
</tr>
</tbody>
</table>


EXAMPLE

/* declare and allocate the remote and local session handles */
CTHANDLE hRemote = ctdbAllocSession(CTSESSION_CTREE);
CTHANDLE hLocal = ctdbAllocSession(CTSESSION_CTREE);

/* logon to c-tree server using the remote session handle */
if (ctdbLogon(hRemote, "FAIRCOM", "ADMIN", "ADMIN") != CTDBRET_OK)
    printf("Remote ctdbLogon failed\n");
/* logon to local session using the local session handle */
if (ctdbLogon(hLocal, "local", "ADMIN", "ADMIN") != CTDBRET_OK)
    printf("Local ctdbLogon failed\n");
/* perform a c-tree instance switch and call CtreeUserOperation function */
if (ctdbSwitchInstance(hRemote) != CTDBRET_OK)
    printf("ctdbSwitchInstance failed\n");
CtreeUserOperation("!mkdir faircom", buffer, sizeof(buffer);

SEE ALSO

ctdbSwitchContext, ctdbGetDatno, ctdbGetIdxno, ctdbGetIdxnoByName, ctdbGetIdxnoByName
17.3.16 **CTBase::SwitchInstance**

Force a c-tree Plus instance switch.

**DECLARATION**

```cpp
void CTBase::SwitchInstance()
```

**DESCRIPTION**

Force a switch to the c-tree Plus instance indicated by the current session object. Each session object has its unique c-tree instance id.

When most c-treeDB C++ methods are called, they automatically perform a c-tree instance switch. SwitchInstance may be useful before calling specific c-tree ISAM or low level functions to make sure the correct instance is active before making those calls.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td>SwitchInstance return no value.</td>
</tr>
</tbody>
</table>


**EXAMPLE**

```cpp
// declare a remote and local session objects
CTSession hRemove(CTSESSION_CTREE);
CTSession hLocal(CTSESSION_CTREE);

// logon to c-tree server using the remote session object
hRemote.Logon("FAIRCOMS", "ADMIN", "ADMIN");
// logon to local session using the local session object
hLocal.Logon("local", "ADMIN", "ADMIN");
// perform a c-tree instance switch and call //CtreeUserOperation function
hRemote.SwitchInstance();
CtreeUserOperation("!mkdir faircom", buffer, sizeof(buffer));
```

**SEE ALSO**

CTRecord::SwitchContext, CTTable::GetDatno, CTTable::GetIdxno, CTIndex::GetIdxno
17.3.17  CTIndex::GetIdxno

Retrieve a table index file number from an index object.

**DECLARATION**

```c
NINT CTIndex::GetIdxno()
```

**DESCRIPTION**

Retrieve a table index file number from an index object.

**RETURN**

GetIdxno return a table index file number.

**EXAMPLE**

```c
// retrieve the first key of first index
TEXT keyval[256];
CTIndex hIndex = hTable.GetIndex(0);

if (FirstKey(hIndex.GetIdxno(), keyval)
    printf("FirstKey failed\n");
```

**SEE ALSO**

CTBase::SwitchInstance, CTRecord::SwitchContext, CTTable::GetDatno, CTTable::GetIdxno
17.3.18 **CTRrecord::SwitchContext**

Force a c-tree Plus ISAM context switch.

**DECLARATION**

```c
void CTRrecord::SwitchContext();
```

**DESCRIPTION**

Force a switch to the c-tree Plus ISAM context indicated by the record object. Each record object has its own c-tree ISAM context id.

When most c-treeDB record handling functions are called, they automatically perform a c-tree ISAM context switch. SwitchContext may be useful before calling specific c-tree ISAM or low level calls to make sure the correct ISAM context is active before making those calls.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td></td>
<td>SwitchContext return no value.</td>
</tr>
</tbody>
</table>


**EXAMPLE**

```c
// force a context switch
hRecord.SwitchContext();
// call ResetRecord
if (ResetRecord((COUNT)hTable.GetDatno(), SWTCURI))
  printf("ResetRecord failed\n");
```

**SEE ALSO**

CTBase::SwitchInstance, CTTtable::GetDatno, CTTtable::GetIdxno CTIndex::GetIdxno
17.3.19  **CTSession::GetPathPrefix**

Get the client-side path prefix.

**DECLARATION**

```cpp
CTBOOL CTSession::GetPathPrefix(CTString& pathPrefix);
```

**DESCRIPTION**

A path prefix can be set anytime after the session handle is allocated. If a path prefix is set before a session logon, the new path prefix will affect the location of the session dictionary file. If a path prefix is set after a session logon, but before a database connect, then the path prefix affects only the database dictionary and any tables that are manipulated during that session.

If a path prefix is set, GetPathPrefix returns YES and the path is set to pathPrefix. If no path prefix is set, GetPathPrefix returns NO and pathPrefix is cleared.

A CTException is thrown when an error occurs. Use CTException::GetErrorCode and CTException::GetErrMsg to retrieve the error code and descriptive message.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NO</td>
<td>CTSession::GetPathPrefix returns NO when no path prefix has been set.</td>
</tr>
<tr>
<td>1</td>
<td>YES</td>
<td>CTSession::GetPathPrefix returns YES when a path prefix has been returned.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer's Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

```c
/* get the current path prefix */
try {
    if (MySession.GetPathPrefix(MyCTStringPath))
        printf("Current path prefix is : %s\n", MyCTStringPath);
    else
        printf("No current path prefix is set.\n");
}
catch(CTException &E) {
    printf("ERROR: (%d) - %s\n", E.GetErrorCode(), E.GetErrMsg());
}
```

**SEE ALSO**

CTSession::SetPathPrefix.
### 17.3.20 CTSession::SetPathPrefix

Set the client-side path prefix.

**DEVELOPMENT**

```cpp
void CTSession::SetPathPrefix(const CTString& pathPrefix);
void CTSession::SetPathPrefix();
```

**DESCRIPTION**

A path prefix can be set anytime after the session handle is allocated. If a path prefix is set before a session logon, the new path prefix will affect the location of the session dictionary file. If a path prefix is set after a session logon, but before a database connect, then the path prefix affects only the database dictionary and any tables that are manipulated during that session.

A CTException is thrown when an error occurs. Use CTException::GetErrorCode and CTException::GetErrorMsg to retrieve the error code and descriptive message.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOID</td>
<td>VOID</td>
<td>No return value.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer's Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

```cpp
/* set a path prefix */
try {
    MySession.SetPathPrefix("C:\FairCom\V8.14\ctreeSDK\ctreeAPI");
} catch(CTException &E) {
    printf("SetPathPrefix error: (%d) - %s\n", E.GetErrorCode(),
           E.GetErrorMsg());
}

/* clear a path prefix */
try {
    MySession.SetPathPrefix("");
} catch(CTException &E) {
    printf("SetPathPrefix error: (%d) - %s\n", E.GetErrorCode(),
           E.GetErrorMsg());
}
```

**SEE ALSO**

CTSession::GetPathPrefix
17.3.21  **CTTable::GetDatno**

Retrieve the table data file number.

**DECLARATION**

NINT CTTable::GetDatno()

**DESCRIPTION**

Retrieve the table data file number from a table object. If an error is detected, GetDatno throw a CTException.

**RETURN**

GetDatno return the table data file number.

**EXAMPLE**

```cpp
void DeleteTable(CTSession& hSession, pTEXT tablename)
{
    CTDBRET eRet;
    CTTable hTable(hSession);

    // open the table exclusive
    hTable.Open(tablename, CTOPEN_EXCLUSIVE);
    // delete a file
    eRet = (CTDBRET)DeleteRFile((COUNT)hTable.GetDatno());
    if (eRet != CTDBRET_OK)
        throw CTException(eRet, "DeleteTable", __FILE__, __LINE__);
}
```

**SEE ALSO**

CTBase::SwitchInstance, CTRecord::SwitchContext, CTTable::GetIdxno, CTIndex::GetIdxno
17.3.22  CTTable::GetIdxno

Retrieve a table index file number given the index name or the c-treeDB index number.

DECLARATION

NINT CTTable::GetIdxno(const CTString& IndexName)
NINT CTTable::GetIdxno(NINT index)

DESCRIPTION

Retrieve the index file number given the index name or the c-treeDB index number.

IndexName contains the symbolic name for the index. Please note the difference between the
index symbolic name and the index file name.

index is a c-treeDB index number. The first c-treeDB index number is zero.

Return the index file number.

RETURN

GetIdxno return a table index file number.

EXAMPLE

// retrieve the first key of first index
TEXT keyval[256];

if (FirstKey(hTable.GetIdxno("IndexName"), keyval)
    printf("FirstKey failed\n");

if (FirstKey(hTable.GetIdxno(0), keyval)
    printf("FirstKey failed\n");

SEE ALSO

CTBase::SwitchInstance, CTRecord::SwitchContext, CTTable::GetDatno, CTIndex::GetIdxno
### 17.3.23 CTTable::Remove

Removes any c-treeDB table.

**DECLARATION**

```c
void CTTable::Remove();
```

**DESCRIPTION**

The CTTable::Remove function allows any table to be deleted, including tables that are not members of a database.

Handle is a table handle. The CTTable::Remove function deletes a c-tree data file and the associated index files from disk. If the table was opened under a database handle, the table is closed and CTDatabase::DeleteTable is called. If the handle is not active, the table is opened exclusive and then deleted. If the table handle is not active, you must set the path, file extension and password for the handle before calling ctdbRemoveTable.

A CTException is thrown when an error occurs. Use CTException::GetErrorCode and CTException::GetErrorMsg to retrieve the error code and descriptive message.

**RETURN**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Constant</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOID</td>
<td>VOID</td>
<td>No return value.</td>
</tr>
</tbody>
</table>

See Appendix C "c-treeDB Errors and Return Values" in the *c-treeDB C API Developer's Guide* for a complete listing of valid c-treeDB error codes and return values.

**EXAMPLE**

```c
/* delete a c-treeDB Table */
try {
    MyTable.Remove();
}
catch(CTException &E) {
    printf("MyTable.Remove error: (%d) - %s\n", E.GetErrorCode(), E.GetErrorMsg());
}
```

**SEE ALSO**

CTDatabase::DeleteTable.