FairCom White Paper

c-treeDB Virtual Tables

Contents

c-treeDB Multi-Record Virtual Tables

A common technique in many early applications was to combine multiple types of records into a single physical data file. Many of those applications have been very successful for many years and are still in service. This technique was a common practice when storage space was at a premium. However, this technique does not conform to the standard, single-schema, relational model necessary for SQL access.

Developers are looking at adding more complex features and interoperability over their data with new technologies such as web access and Windows .NET support. This has proven to be a challenge when the data is not of a consistent schema due to multiple record types in a single table.

c-treeACE allows a single table with multiple schemas (multiple record types) to appear to SQL as multiple virtual tables, each with a single schema.

For information about multi-record tables, see the *c-treeDB Developer's Guide*.

The next sections explain the techniques used to implement this feature.

Published 2/20/2018

1.1 c-treeDB Virtual Tables

c-treeDB Virtual Tables are a unique c-tree feature that allows such applications as multiple record types within a single c-tree data file. This situation is most likely to be found in earlier c-tree applications that are later ported to c-treeACE SQL with minimal application changes. The c-treeACE COBOL interface uses this support to allow flexible file creation from the COBOL application.

Introduction to Virtual Tables

As a proven high-performance database technology, c-treeACE has allowed application developers to craft their data handling needs precisely for their application needs for many years. Using only the necessary features required for a specific application task allows for very high-performance applications. Relational aspects of the database were maintained by the application greatly simplifying the persisted information required. As database technology has evolved, relations over the data are becoming more common and complex, and are now generally assumed to be persisted with the data. SQL is an excellent example of this, where tables are organized into databases, and columns comprise a data row within a table.

A common technique in many early c-treeACE applications was to combine multiple types of records into a single physical data file. With c-treeACE's direct record access and ISAM technology, this was very easy to implement and maintain. Many of these applications have been very successful for many years and are maintained yet today. As a successful application, developers are looking at adding more complex features and interoperability over their data with new technologies such as web access and Windows .NET support. This has proven to be a challenge when the data is not of a consistent schema due to multiple record types in a single table.

With extensible c-treeACE database technology this is no longer a challenge. c-treeDB offers a flexible and powerful relational architecture over your existing c-treeACE data files. A feature within c-treeDB called Virtual Tables allows an easy path to full relational support, including SQL, over these types of existing tables—all without changes to the underlying application and data.

Virtual Tables

A Virtual Table is operated upon as an ordinary c-treeDB table (with a few limitations, depending on the type of the Virtual Table, see *Unsupported Functions* (page 16) for details). However, it is not an actual physical table; it is an "interpretation" of the records (some or all) within the host (parent) table. Virtual Tables can be thought of as a generic concept. Actual implementations of this concept are the Virtual Table types, which are currently implemented through the *MultiRecordTable* type.

A Virtual Table is identified in the c-treeDB database dictionary as a particular marker indicating the type of Virtual Table. The dictionary handling function has been modified to differentiate Virtual Tables from regular tables and also make the c-treeDB interface view a Virtual Table as a regular table. For instance, when listing tables of a dictionary, Virtual Tables are listed together along with regular tables. Once created, a Virtual Table is operated upon as any other c-treeDB table.

Multiple Record Table

MultiRecordTable, *MRTTable*, support is the first implementation of a Virtual Table. This is a type of table in which the record structure varies from record to record depending on some criteria. This implementation requires that within a given record schema there is a rule identifying if a record can be properly represented with that record schema or not. This support requires common schema to be defined for all records (however, this may describe only the first part of the record) and a filter identifying records belonging to a particular record schema based on this common schema.

The actual table which contains all data records is required to have a standard c-treeACE *DODA* and *IFIL* resource (or provided through external callbacks). This table is called the host or parent. The

c-treeDB API provides the means to add definitions for the various record type schemas by defining an *MRTTable* per each record type as a Virtual Table for the host table.

The c-treeDB **ctdbCreateMRTTTable()** function behaves much like the ordinary **ctdbCreateTable()** function, however, it creates a *MRTTable* definition.

ctdbIsVTable(CTHANDLE table) can be used to check if a table is a Virtual Table or a regular table.

Virtual Table Callbacks

- CTDB_ON_TABLE_GET_VTABLE_INFO

  CTDB_ON_TABLE_GET_VTABLE_INFO is called when c-treeDB looks for the Virtual Table resource, typically at the time of a Virtual Table open or when calling **ctdbGetVTableInfoFromTable()**. This callback should be implemented when using alternative (to the resource as defined) methods of tracking Virtual Table definitions. (For example, c-treeACE COBOL support implements this callback.)

1.2 c-treeACE MRTTable API

This chapter describes the c-treeDB functions available to enable Virtual Table support.

For an example of how to use these functions to enable Virtual Table support, see *c-treeDB MRTTable Example* (page 17) in the section titled *Tutorials* (page 17).

ctdbAddMRTTable

ctdbAddMRTTable() adds an existing *MRTTable* definition to the database dictionary.

Declaration

```
CTDBRET ctdbAddMRTTable(pCTDBDATABASE pDatabase, pTEXT Name, pTEXT ParentName, UINT info);
```

Description

- *pDatabase*: the database handle
- *Name*: the *MRTTable* name
- *ParentName*: the host table name
- *info*: the *MRTTable* definition number

ctdbAddMRTTable() functions much as **ctdbAddTable()** in adding an existing table to the database dictionary.

Return Values

Value	Symbolic Constant	Explanation
0	NO_ERROR	Successful operation.

See c-tree Plus Error Codes (http://docs.faircom.com/doc/ctreeplus/#28320.htm) for a complete listing of valid c-tree Plus error values.

Example

Here is example pseudo code to add an existing *MRTTable* to a database dictionary. This example also demonstrates the **ctdbAllocVTableInfo() ctdbGetVTableInfoFromTable()**, and **ctdbAddMRTTable()**, **ctdbFreeVTableInfo()** functions.

For this example, assume that a *MRTTable* is already defined and you copy the host table to another machine. In this case, you simply need to add the tables to the new database dictionary.

```
pCTDBVTABLEINFO VTableInfo;
pCTDBTABLE hTable;

hTable = ctdbAllocTable(hDatabase);
if (hTable == NULL)
{
        /* error */
}

retval = ctdbOpenTable(hTable, "tutorial_host", CTOPEN_EXCLUSIVE);
if (retval != CTDBRET_OK)
{
        /* error */
}

VTableInfo = ctdbAllocVTableInfo(hTable, 200); /*retrieve max 200 definition (which is the maximum
supported) */
if (VTableInfo == NULL)
{
        /* error */
}

ctdbGetVTableInfoFromTable(VTableInfo);
ctdbCloseTable(hTable);

for (i = 0; i < VTableInfo->actual_elements; i++)
{
        retval = ctdbAddMRTTable(hDatabase, VTableInfo->data[i].name, "tutorial_host",
VTableInfo->data[i].id);
        if (retval != CTDBRET_OK)
        {
```

```
                /* error */
        }
}
ctdbFreeVTableInfo(VTableInfo );
```

See Also

ctdbAllocVTableInfo(), **ctdbCreateMRTTable()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableInfoFromTable()**, **ctdbGetVTableNumber()**, **ctdbIsVTable()**, **ctdbRemoveVTableResource()**, **ctdbSetMRTTableFilter()**

ctdbAddVTableResource

ctdbAddVTableResource() adds a virtual table resource.

Declaration

```
CTDBRET ctdbAddVTableResource(CTHANDLE Parent, CTHANDLE Child)
```

Description

- *Parent*: Host table to which the virtual table is assigned.
- *Child*: Virtual table handle.

Note: This function does not update the database dictionary and does not support callbacks. This function is used internally by c-treeDB and is documented only for completeness.

Return Values

Value	Symbolic Constant	Description
4130	**CTDBRET_NOMOREVTRES**	No more RESOURCE for vtable available on parent table.
4131	**CTDBRET_VTABLEEXIST**	The table has *VTable* defined in the dictionary.
4132	**CTDBRET_VTABLETYPE**	The *VTable* type in the dictionary mismatches the one in the resource.

See c-tree Plus Error Codes (http://docs.faircom.com/doc/ctreeplus/#28320.htm) for a complete listing of valid c-tree Plus error values.

See Also

ctdbAddMRTTable(), **ctdbAllocVTableInfo()**, **ctdbCreateMRTTable()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableInfoFromTable()**, **ctdbGetVTableNumber()**, **ctdbIsVTable()**, **ctdbRemoveVTableResource()**, **ctdbSetMRTTableFilter()**

ctdbAllocVTableInfo

ctdbAllocVTableInfo() allocates a new *CTDBVTABLEINFO* structure.

Declaration

```
CTHANDLE ctdbAllocVTableInfo(pCTDBTABLE pTable, UCOUNT size);
```

Description

- *pTable*: Table to allocate a *CTDBVTABLEINFO* object
- *size*: Number of elements allocated

Return Values

ctdbAllocVTableInfo() returns a c-treeDB handle to a *CTDBVTABLEINFO* structure.

See Also

ctdbAddMRTTable(), **ctdbAddVTableResource()**, **ctdbCreateMRTTable()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableInfoFromTable()**, **ctdbGetVTableNumber()**, **ctdbIsVTable()**, **ctdbRemoveVTableResource()**, **ctdbSetMRTTableFilter()**

ctdbCreateMRTTable

ctdbCreateMRTTable() creates a new *MRTTable* based on a host table.

Declaration

```
CTDBRET ctdbCreateMRTTable (CTHANDLE Handle, pTEXT VTableName, pTEXT ParentName, CTCREATE_MODE
CreateMode, pTEXT filter);
```

Description

- *Handle*: *MRTTable* handle
- *VtableName*: name of the *MRTTable*
- *ParentName*: name of the host (or parent) table
- *CreateMode*: virtual table create mode
- *filter*: the filter to apply to the host table to identify the record belonging to this *MRTTable*

Once created, an *MRTTable* behaves as a regular c-treeDB table.

Limitations: At this time it is not possible to use record filters and the **ctdbAlterTable()** function on a *MRTTable* Table.

Return Values

Value	Symbolic Constant	Description
4130	**CTDBRET_NOMOREVTRES**	No more RESOURCE for VTable available on parent table.
4131	**CTDBRET_VTABLEEXIST**	The table has *VTable* defined in the dictionary.
4132	**CTDBRET_VTABLETYPE**	*VTable* type mismatch between dictionary and resource.

See c-tree Plus Error Codes (http://docs.faircom.com/doc/ctreeplus/#28320.htm) for a complete listing of valid c-tree Plus error values.

Example

```
CTHANDLE pField0, pField1, pField2, pField3, pField4;
pField0 = ctdbAddField(hTableCustOrdr, "rectype", CT_FSTRING, 1);
pField1 = ctdbAddField(hTableCustOrdr, "co_ordrnumb", CT_FSTRING, 4);
pField2 = ctdbAddField(hTableCustOrdr, "co_ordrdate", CT_DATE, 4);
pField3 = ctdbAddField(hTableCustOrdr, "co_promdate", CT_DATE, 4);
pField4 = ctdbAddField(hTableCustOrdr, "co_custnumb", CT_FSTRING, 4);

if (!pField1 || !pField2 || !pField3 || !pField4)
    Handle_Error("Define(): ctdbAddField()");

/* create table */
if (ctdbCreateMRTTable(hTableCustOrdr, "custordr","tutorial_host",
CTCREATE_NORMAL|CTCREATE_NONULFLD,"rectype==\"O\""))
    Handle_Error("ctdbCreateMRTTable()");
```

See Also

ctdbAddMRTTable(), **ctdbAddVTableResource()**, **ctdbAllocVTableInfo()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableInfoFromTable()**, **ctdbGetVTableNumber()**, **ctdbIsVTable()**, **ctdbRemoveVTableResource()**, **ctdbSetMRTTableFilter()**

ctdbFreeVTableInfo

ctdbFreeVTableInfo() frees memory associated with a *CTDBVTABLEINFO* structure.

Declaration

```
VOID ctdbFreeVTableInfo(pCTDBVTABLEINFO Info);
```

Description

- *Info*: A *CTDBVTABLEINFO* object

Return Values

ctdbFreeVTableInfo() does not return a value.

See Also

ctdbAddMRTTable(), **ctdbAddVTableResource()**, **ctdbAllocVTableInfo()**, **ctdbCreateMRTTable()**, **ctdbGetVTableInfoFromTable()**, **ctdbGetVTableNumber()**, **ctdbIsVTable()**, **ctdbRemoveVTableResource()**, **ctdbSetMRTTableFilter()**

ctdbGetVTableInfoFromTable

ctdbGetVTableInfoFromTable() lists virtual tables defined in a host file.

Declaration

```
VOID ctdbGetVTableInfoFromTable(pCTDBVTABLEINFO VtableRes)
```

Description

There may be a need to list the virtual tables defined in a host file without looking at the c-treeDB database dictionary (for example, the dictionary is missing). This information can be reconstructed with this function.

- *pCTDBVTABLEINFO VtableRes*: a pre-allocated structure (using **ctdbAllocVTableInfo()**) to contain information retrieved from the table resources.
  - *VtableRes->hTable*: the host table handle (*pCTDBTABLE*)
  - *VtableRes->data*: an array of *tagVTABLEOBJ*
    - *data->id*: the virtual table number (*UCOUNT*)
    - *data->type*: the virtual table type (*VTABLE_TYPE*)
    - *data->name*: the virtual table name (*pTEXT*) (when applies)
  - *VtableRes->data_elements*: the maximum number of elements *data* could contain (*UCOUNT*)
  - *VtableRes->actual_elements*: the actual number of elements set in *data* (*UCOUNT*)

The *hTable* and *data_elements* fields are populated by **ctdbAllocVTableInfo()**. *data* and *actual_elements* are populated by **ctdbGetVTableInfoFromTable()**.

Note: ctdbFreeVTableInfo() should be used in order to properly free memory allocated for *CTDBVTABLEINFO*.

Return Values

ctdbGetVTableInfoFromTable() does not return a value.

See Also

ctdbAddMRTTable(), **ctdbAddVTableResource()**, **ctdbAllocVTableInfo()**, **ctdbCreateMRTTable()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableNumber()**, **ctdbIsVTable()**, **ctdbRemoveVTableResource()**, **ctdbSetMRTTableFilter()**

ctdbGetVTableNumber

ctdbGetVTableNumber() retrieves the number assigned to a virtual table.

Declaration

```
UINT ctdbGetVTableNumber(CTHANDLE table)
```

Description

- *table*: A handle to any c-treeDB table.

Upon a virtual table open, c-treeDB internally opens the virtual table host and loads the definition of the virtual table from a resource contained in the host table. To identify among multiple possible virtual tables created on a single host table, a definition number is used which is automatically assigned by c-treeDB when defining the virtual table and is stored in the database dictionary.

The dictionary entry for a virtual table contains (in addition to existing table information) links to the host table and the virtual table number.

The resource containing the virtual table information has a resource type set to *FC_CTDB_VTABLES* (2). Resource numbers range from *FCRES_CTDB_VTABLES* (0) to *FCRES_CTDB_VTABLES_END* (199). Its content is as follows (packed, no alignment...)

- First byte: endian-ness
- Next 4 bytes: virtual table type

The remainder of the resource is variable and depends on the virtual table type.

Return Values

ctdbGetVTableNumber() returns the number assigned to a virtual table.

See Also

ctdbAddMRTTable(), **ctdbAddVTableResource()**, **ctdbAllocVTableInfo()**, **ctdbCreateMRTTable()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableInfoFromTable()**, **ctdbIsVTable()**, **ctdbRemoveVTableResource()**, **ctdbSetMRTTableFilter()**

ctdbIsVTable

ctdbIsVTable() can be used to check if a table is a virtual table or a regular table.

Declaration

```
CTBOOL ctdbIsVTable(CTHANDLE table)
```

Description

- *table*: A handle to any c-treeDB table.

Return Values

ctdbIsVTable() returns *YES* if the table is a Virtual Table, *NO* otherwise.

See Also

ctdbAddMRTTable(), **ctdbAddVTableResource()**, **ctdbAllocVTableInfo()**, **ctdbCreateMRTTable()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableInfoFromTable()**, **ctdbGetVTableNumber()**, **ctdbRemoveVTableResource()**, **ctdbSetMRTTableFilter()**

ctdbRemoveVTableResource

ctdbRemoveVTableResource() removes a virtual table resource.

Declaration

```
CTDBRET ctdbRemoveVTableResource(CTHANDLE Parent, NINT number)
```

Description

- *Parent*: Host table to which the virtual table is assigned.
- *number*: Number of the virtual table.

Note: This function does not update the database dictionary and does not support callbacks. This function is used internally by c-treeDB and is documented only for completeness.

Return Values

Value	Symbolic Constant	Description
4130	**CTDBRET_NOMOREVTRES**	No more RESOURCE for VTable available on parent table.
4131	**CTDBRET_VTABLEEXIST**	The table has *VTable* defined in the dictionary.
4132	**CTDBRET_VTABLETYPE**	The *VTable* type in the dictionary mismatches the one in the resource.

See c-tree Plus Error Codes (http://docs.faircom.com/doc/ctreeplus/#28320.htm) for a complete listing of valid c-tree Plus error values.

See Also

ctdbAddMRTTable(), **ctdbAddVTableResource()**, **ctdbAllocVTableInfo()**, **ctdbCreateMRTTable()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableInfoFromTable()**, **ctdbGetVTableNumber()**, **ctdbIsVTable()**, **ctdbSetMRTTableFilter()**

ctdbSetMRTTableFilter

ctdbSetMRTTableFilter() sets filter information in the virtual table handle.

Declaration

```
CTDBRET ctdbSetMRTTableFilter(CTHANDLE Handle, pTEXT condition)
```

Description

When implementing virtual table callbacks the **ctdbSetMRTTableFilter()** function may be of use.

This function sets the filter information in the *MRTTable Handle* such that at the end of the open procedure c-treeDB knows which filter to use.

Return Values

Value	Symbolic Constant	Description
4130	**CTDBRET_NOMOREVTRES**	No more RESOURCE for vtable available on parent table.
4131	**CTDBRET_VTABLEEXIST**	The table has *VTable* defined in the dictionary.
4132	**CTDBRET_VTABLETYPE**	The *VTable* type in the dictionary mismatches the one in the resource.

See c-tree Plus Error Codes (http://docs.faircom.com/doc/ctreeplus/#28320.htm) for a complete listing of valid c-tree Plus error values.

See Also

ctdbAddMRTTable(), **ctdbAddVTableResource()**, **ctdbAllocVTableInfo()**, **ctdbCreateMRTTable()**, **ctdbFreeVTableInfo()**, **ctdbGetVTableInfoFromTable()**, **ctdbGetVTableNumber()**, **ctdbIsVTable()**, **ctdbRemoveVTableResource()**

Virtual Table Errors

- **CTDBRET_NOMOREVTRES** (4130) No more resources for *VTable* available on parent table.
- **CTDBRET_VTABLEEXIST** (4131) The table has *VTable* defined in the dictionary.
- **CTDBRET_VTABLETYPE** (4132) The *VTable* type in the dictionary mismatches the one in the resource.

Unsupported Functions

The following functions are not supported in the MRTTable API:

- **ctdbGetRecordCount()** - Because it is not possible to efficiently count records in an MRTTable, an error is returned if this function is used.
- **ctdbAlterTable()** - The alter-table function is not allowed on an MRTTable.

1.3 Multi-Record Table Tutorial

To demonstrate the utility of the c-treeDB Virtual Table support, this section contains a full working example based on the standard c-treeDB tutorial 2 exercise.

This tutorial will introduce the most basic c-treeACE API to accomplish creating and manipulating a table through c-treeACE. You will create and populate four tables and then it will create the MRT HOST table, which will host your Virtual Table. Functionally, the application you will build in this application will perform the following:

1. Create a database.
2. Create four tables each with an index.
3. Populate each table with a few records.
4. Build a query utilizing the advantage of indexes.
5. Output the results of the query.

c-treeDB MRTTable Example

```
/*
 * MRTTable_tutorial.c
 *
 * Public domain c-treeACE C example
 *
 * FairCom Corporation, 6300 West Sugar Creek Drive, Columbia, MO 65203 USA
 * FairCom Corporation, 6300 West Sugar Creek Drive, Columbia, MO 65203
 *
 * The goal of this tutorial is to introduce the most basic c-treeACE API
 * to accomplish creating and manipulating a table through c-treeACE
 *
 * Functionally, this application will perform the following:
 *  1.  Create a database
 *  2.  Create 4 tables each with an index
 *  3.  Populate each table with a few records
 *  4.  Build a query utilizing the advantage of indexes
 *  5.  Output the results of the query
 */

#ifdef _WIN32_WCE
#undef UNICODE
#undef _UNICODE
#define main my_main
#endif

/* Preprocessor definitions and includes */
#include "ctdbsdk.h" /* c-tree headers */
#define END_OF_FILE INOT_ERR  /* INOT_ERR is ctree's 101 error. See cterrc.h */

/* Global declarations */

/* Session handle */
CTHANDLE hSession;

/* Database handle */
CTHANDLE hDatabase;

/* Table handles */
CTHANDLE hTableCustMast;
CTHANDLE hTableCustOrdr;
CTHANDLE hTableOrdrItem;
CTHANDLE hTableItemMast;

/* Record handles */
CTHANDLE hRecordCustMast;
CTHANDLE hRecordCustOrdr;
```

```
CTHANDLE hRecordOrdrItem;
CTHANDLE hRecordItemMast;

/* Function declarations */

#ifdef PROTOTYPE
VOID Initialize(VOID), Define(VOID), Manage(VOID), Done(VOID);

VOID Create_HOST_Table(VOID);
VOID Create_CustomerMaster_Table(VOID), Create_ItemMaster_Table(VOID);
VOID Create_OrderItems_Table(VOID), Create_CustomerOrders_Table(VOID);

VOID Add_CustomerMaster_Records(VOID), Add_ItemMaster_Records(VOID);
VOID Add_CustomerOrders_Records(VOID), Add_OrderItems_Records(VOID);
VOID Delete_Records(CTHANDLE hRecord);
VOID Check_Table_Mode(CTHANDLE hTable);

VOID Handle_Error(CTSTRING);
#else
VOID Initialize(), Define(), Manage(), Done();

VOID Create_HOST_Table();
VOID Create_CustomerMaster_Table(), Create_ItemMaster_Table();
VOID Create_OrderItems_Table(), Create_CustomerOrders_Table();

VOID Add_CustomerMaster_Records(), Add_ItemMaster_Records();
VOID Add_CustomerOrders_Records(), Add_OrderItems_Records();
VOID Delete_Records();
VOID Check_Table_Mode();

VOID Handle_Error();
#endif

/*
 * main()
 *
 * The main() function implements the concept of "init, define, manage
 * and you're done..."
 */

#ifdef PROTOTYPE
NINT main (NINT argc, pTEXT argv[])
#else
NINT main (argc, argv)
NINT argc;
TEXT argv[];
#endif
{
   Initialize();

   Define();

   Manage();

   Done();

   printf("\nPress <ENTER> key to exit . . .\n");
   getchar();

   return(0);
}

/*
 * Initialize()
 *
 * Perform the minimum requirement of logging onto the c-tree Server
 */

#ifdef PROTOTYPE
VOID Initialize(VOID)
#else
VOID Initialize()
#endif
```

```
{
   CTDBRET  retval;

   printf("INIT\n");

   if ((retval = ctdbStartDatabaseEngine())) /* This function is required when you are using the
Server DLL model to start the underlying Server. */
       Handle_Error("Initialize(): ctdbStartDatabaseEngine()");    /* It does nothing in all
other c-tree models */

   /* allocate session handle */
   if ((hSession = ctdbAllocSession(CTSESSION_CTDB)) == NULL)
      Handle_Error("Initialize(): ctdbAllocSession()");

   if ((hDatabase = ctdbAllocDatabase(hSession)) == NULL)
      Handle_Error("Initialize(): ctdbAllocDatabase()");

   /* connect to server */
   printf("\tLogon to server...\n");
   if (ctdbLogon(hSession, "FAIRCOMS", "", ""))
      Handle_Error("Initialize(): ctdbLogon()");
   if (ctdbConnect(hDatabase, "ctreeSQL"))
      Handle_Error("Initialize(): ctdbConnect()");
}

/*
 * Define()
 *
 * Open the tables, if they exist. Otherwise create and open the tables
 */

#ifdef PROTOTYPE
VOID Define(VOID)
#else
VOID Define()
#endif
{
   printf("DEFINE\n");

   Create_HOST_Table();
   Create_CustomerMaster_Table();
   Create_CustomerOrders_Table();
   Create_OrderItems_Table();
   Create_ItemMaster_Table();

   if (ctdbOpenTable(hTableCustMast, "custmast", CTOPEN_NORMAL))
         Handle_Error("Create_CustomerMaster_Table(): ctdbOpenTable()");
   if (ctdbOpenTable(hTableCustOrdr, "custordr", CTOPEN_NORMAL))
         Handle_Error("Create_CustomerOrders_Table(): ctdbOpenTable()");
   if (ctdbOpenTable(hTableOrdrItem, "ordritem", CTOPEN_NORMAL))
         Handle_Error("Create_OrderItems_Table(): ctdbOpenTable()");
   if (ctdbOpenTable(hTableItemMast, "itemmast", CTOPEN_NORMAL))
         Handle_Error("Create_ItemMaster_Table(): ctdbOpenTable()");
}

/*
 * Manage()
 *
 * Populates table and perform a simple query
 *
 */

#ifdef PROTOTYPE
VOID Manage(VOID)
#else
VOID Manage()
#endif
{
   CTDBRET  retval;
   CTSIGNED quantity;
   CTFLOAT  price, total;
   TEXT     itemnumb[5+1], custnumb[4+1], ordrnumb[6+1], custname[47+1];
   CTBOOL   isOrderFound, isItemFound;
```

```
printf("MANAGE\n");

/* populate the tables with data */
Add_CustomerMaster_Records();
Add_CustomerOrders_Records();
Add_OrderItems_Records();
Add_ItemMaster_Records();

/* perform a query:
   list customer name and total amount per order

   name               total
   @@@@@@@@@@@@@      $xx.xx

   for each order in the CustomerOrders table
      fetch order number
      fetch customer number
      fetch name from CustomerMaster table based on customer number
      for each order item in OrderItems table
         fetch item quantity
         fetch item number
         fetch item price from ItemMaster table based on item number
      next
   next
*/
printf("\n\tQuery Results\n");

/* get the first order */
if (ctdbFirstRecord(hRecordCustOrdr))
   Handle_Error("Manage(): ctdbFirstRecord()");

isOrderFound = YES;
while (isOrderFound)   /* for each order in the CustomerOrders table */
{
   /* fetch order number */
   retval = ctdbGetFieldAsString(hRecordCustOrdr, 1, ordrnumb, sizeof(ordrnumb));
   /* fetch customer number */
   retval |= ctdbGetFieldAsString(hRecordCustOrdr, 4, custnumb, sizeof(custnumb));
   if (retval)
      Handle_Error("Manage(): ctdbGetFieldAsString()");

   /* fetch name from CustomerMaster table based on customer number */
   if (ctdbClearRecord(hRecordCustMast))
      Handle_Error("Manage(): ctdbClearRecord()");
   if (ctdbSetFieldAsString(hRecordCustMast, 1, custnumb))
      Handle_Error("Manage(): ctdbSetFieldAsString()");
   if (ctdbFindRecord(hRecordCustMast, CTFIND_EQ))
      Handle_Error("Manage(): ctdbFindRecord()");
   if (ctdbGetFieldAsString(hRecordCustMast, 5, custname, sizeof(custname)))
      Handle_Error("Manage(): ctdbGetFieldAsString()");

   /* fetch item price from OrderItems table */
   if (ctdbClearRecord(hRecordOrdrItem))
      Handle_Error("Manage(): ctdbClearRecord()");
   if (ctdbSetFieldAsString(hRecordOrdrItem, 1, ordrnumb))
      Handle_Error("Manage(): ctdbSetFieldAsString()");
   /* define a recordset to scan only items applicable to this order */
   if (ctdbRecordSetOn(hRecordOrdrItem, 4))
      Handle_Error("Manage(): ctdbRecordSetOn()");
   if (ctdbFirstRecord(hRecordOrdrItem))
      Handle_Error("Manage(): ctdbFirstRecord()");
   isItemFound = YES;

   total = 0;
   while (isItemFound)   /* for each order item in OrderItems table */
   {
      /* fetch item quantity */
      if (ctdbGetFieldAsSigned(hRecordOrdrItem, 3, &quantity))
         Handle_Error("Manage(): ctdbGetFieldAsSigned()");
      /* fetch item number */
      if (ctdbGetFieldAsString(hRecordOrdrItem, 4, itemnumb, sizeof(itemnumb)))
         Handle_Error("Manage(): ctdbGetFieldAsString()");
```

```
            /* fetch item price from ItemMaster table based on item number */
            if (ctdbClearRecord(hRecordItemMast))
               Handle_Error("Manage(): ctdbClearRecord()");
            if (ctdbSetFieldAsString(hRecordItemMast, 1, itemnumb))
               Handle_Error("Manage(): ctdbSetFieldAsString()");
            if (ctdbFindRecord(hRecordItemMast, CTFIND_EQ))
               Handle_Error("Manage(): ctdbFindRecord()");
            if (ctdbGetFieldAsFloat(hRecordItemMast, 3, &price))
               Handle_Error("Manage(): ctdbGetFieldAsFloat()");

            /* calculate order total */
            total += (price * quantity);

            /* read next record */
            retval = ctdbNextRecord(hRecordOrdrItem);
            if (retval != CTDBRET_OK)
            {
               if (retval == END_OF_FILE)
                  isItemFound = NO;
               else
                    Handle_Error("Manage(): ctdbNextRecord()");
            }
         }

         ctdbRecordSetOff(hRecordOrdrItem);

         /* output data to stdout */
         printf("\t\t%-20s %.2f\n", custname, total);

         /* read next order */
         retval = ctdbNextRecord(hRecordCustOrdr);
         if (retval != CTDBRET_OK)
         {
            if (retval == END_OF_FILE)
               isOrderFound = NO;
            else
               Handle_Error("Manage(): ctdbNextRecord()");
         }
      }
   }

   /*
    * Done()
    *
    * This function handles the housekeeping of closing tables and
    * freeing of associated memory
    */

   #ifdef PROTOTYPE
   VOID Done(VOID)
   #else
   VOID Done()
   #endif
   {
      printf("DONE\n");

      /* close tables */
      printf("\tClose tables...\n");
      if (ctdbCloseTable(hTableCustMast))
         Handle_Error("Done(): ctdbCloseTable()");
      if (ctdbCloseTable(hTableOrdrItem))
         Handle_Error("Done(): ctdbCloseTable()");
      if (ctdbCloseTable(hTableCustOrdr))
         Handle_Error("Done(): ctdbCloseTable()");
      if (ctdbCloseTable(hTableItemMast))
         Handle_Error("Done(): ctdbCloseTable()");

      /* logout */
      printf("\tLogout...\n");
      if (ctdbLogout(hSession))
         Handle_Error("Done(): ctdbLogout()");
```

```
   /* free handles */
   ctdbFreeRecord(hRecordCustMast);
   ctdbFreeRecord(hRecordItemMast);
   ctdbFreeRecord(hRecordOrdrItem);
   ctdbFreeRecord(hRecordCustOrdr);

   ctdbFreeTable(hTableCustMast);
   ctdbFreeTable(hTableItemMast);
   ctdbFreeTable(hTableOrdrItem);
   ctdbFreeTable(hTableCustOrdr);

   ctdbFreeSession(hSession);

   /* If you are linked to the Server DLL, then we should stop our Server at the end of the program.
*/
   /* It does nothing in all other c-tree models */
   ctdbStopDatabaseEngine();
}

/*
 * Create_HOST_Table()
 *
 * Open the MRT HOST table, if it exists. Otherwise create it
 */

#ifdef PROTOTYPE
VOID Create_HOST_Table(VOID)
#else
VOID Create_HOST_Table()
#endif
{
   CTHANDLE pField1, pField2, pField3;
   CTHANDLE pIndex;
   CTHANDLE pIseg;
   CTHANDLE hosttable;

   /* define table CustomerMaster */
   printf("\ttable HOST\n");

   /* allocate a table handle */
   if ((hosttable = ctdbAllocTable(hDatabase)) == NULL)
      Handle_Error("Create_HOST_Table(): ctdbAllocTable()");

   /* open table */
   if (ctdbOpenTable(hosttable, "tutorial_host", CTOPEN_NORMAL))
   {
      /* define table fields */
      pField1 = ctdbAddField(hosttable, "rectype", CT_FSTRING, 1);
      pField2 = ctdbAddField(hosttable, "rec_numb", CT_FSTRING, 4);
      pField3 = ctdbAddField(hosttable, "variablepart", CT_STRING, 1);

      if (!pField1 || !pField2 || !pField3 )
         Handle_Error("Create_HOST_Table(): ctdbAddField()");

      /* define index */
      pIndex = ctdbAddIndex(hosttable, "rec_numb", CTINDEX_FIXED, YES, NO);
      pIseg = ctdbAddSegment(pIndex, pField2, CTSEG_SCHSEG);

      if (!pIndex || !pIseg)
         Handle_Error("Create_HOST_Table(): ctdbAddIndex()|ctdbAddSegment()");

      /* create table */
      if (ctdbCreateTable(hosttable, "tutorial_host", CTCREATE_NORMAL|CTCREATE_NONULFLD))
/*CTCREATE_NONULFLD because NULFLD may case troubles in case different schemas have different
number of fields */
         Handle_Error("Create_HOST_Table(): ctdbCreateTable()");
   }
   else
   {
      Check_Table_Mode(hosttable);
      ctdbCloseTable(hosttable);
   }
}
```

```
/*
 * Create_CustomerMaster_Table()
 *
 * Open table CustomerMaster, if it exists. Otherwise create it
 * along with its indices and open it
 */

#ifdef PROTOTYPE
VOID Create_CustomerMaster_Table(VOID)
#else
VOID Create_CustomerMaster_Table()
#endif
{
   CTHANDLE pField0, pField1, pField2, pField3, pField4;
   CTHANDLE pField5, pField6, pField7;

   /* define table CustomerMaster */
   printf("\ttable CustomerMaster\n");

   /* allocate a table handle */
   if ((hTableCustMast = ctdbAllocTable(hDatabase)) == NULL)
      Handle_Error("Create_CustomerMaster_Table(): ctdbAllocTable()");

   /* open table */
   if (ctdbOpenTable(hTableCustMast, "custmast", CTOPEN_NORMAL))
   {
      /* define table fields */
      pField0 = ctdbAddField(hTableCustMast, "rectype", CT_FSTRING, 1);
      pField1 = ctdbAddField(hTableCustMast, "cm_custnumb", CT_FSTRING, 4);
      pField2 = ctdbAddField(hTableCustMast, "cm_custzipc", CT_FSTRING, 9);
      pField3 = ctdbAddField(hTableCustMast, "cm_custstat", CT_FSTRING, 2);
      pField4 = ctdbAddField(hTableCustMast, "cm_custratg", CT_FSTRING, 1);
      pField5 = ctdbAddField(hTableCustMast, "cm_custname", CT_STRING, 47);
      pField6 = ctdbAddField(hTableCustMast, "cm_custaddr", CT_STRING, 47);
      pField7 = ctdbAddField(hTableCustMast, "cm_custcity", CT_STRING, 47);

      if (!pField1 || !pField2 || !pField3 || !pField4 ||
          !pField5 || !pField6|| !pField7)
         Handle_Error("Create_CustomerMaster_Table(): ctdbAddField()");

      /* create table */
      if (ctdbCreateMRTTable(hTableCustMast, "custmast","tutorial_host",
CTCREATE_NORMAL|CTCREATE_NONULFLD,"rectype==\"C\""))
         Handle_Error("Create_CustomerMaster_Table(): ctdbCreateTable()");

   }
   else
               ctdbCloseTable(hTableCustMast);
}

/*
 * Create_CustomerOrders_Table()
 *
 * Open table CustomerOrders, if it exists. Otherwise create it
 * along with its indices and open it
 */

#ifdef PROTOTYPE
VOID Create_CustomerOrders_Table(VOID)
#else
VOID Create_CustomerOrders_Table()
#endif
{
   CTHANDLE pField0, pField1, pField2, pField3, pField4;

   /* define table CustomerOrders */
   printf("\ttable CustomerOrders\n");

   /* allocate a table handle */
   if ((hTableCustOrdr = ctdbAllocTable(hDatabase)) == NULL)
      Handle_Error("Create_CustomerOrders_Table(): ctdbAllocTable()");
```

```
   /* open table */
   if (ctdbOpenTable(hTableCustOrdr, "custordr", CTOPEN_NORMAL))
   {
      /* define table fields */
      pField0 = ctdbAddField(hTableCustOrdr, "rectype", CT_FSTRING, 1);
      pField1 = ctdbAddField(hTableCustOrdr, "co_ordrnumb", CT_FSTRING, 4);
      pField2 = ctdbAddField(hTableCustOrdr, "co_ordrdate", CT_DATE, 4);
      pField3 = ctdbAddField(hTableCustOrdr, "co_promdate", CT_DATE, 4);
      pField4 = ctdbAddField(hTableCustOrdr, "co_custnumb", CT_FSTRING, 4);

      if (!pField1 || !pField2 || !pField3 || !pField4)
         Handle_Error("Define(): ctdbAddField()");

      /* create table */
      if (ctdbCreateMRTTable(hTableCustOrdr, "custordr","tutorial_host",
CTCREATE_NORMAL|CTCREATE_NONULFLD,"rectype==\"O\""))
         Handle_Error("Create_CustomerOrders_Table(): ctdbCreateTable()");
   }
   else
         ctdbCloseTable(hTableCustOrdr);
}

/*
 * Create_OrderItems_Table()
 *
 * Open table OrderItems, if it exists. Otherwise create it
 * along with its indices and open it
 */

#ifdef PROTOTYPE
VOID Create_OrderItems_Table(VOID)
#else
VOID Create_OrderItems_Table()
#endif
{
   CTHANDLE pField0, pField1, pField2, pField3, pField4;

   /* define table OrderItems */
   printf("\ttable OrderItems\n");

   /* allocate a table handle */
   if ((hTableOrdrItem = ctdbAllocTable(hDatabase)) == NULL)
      Handle_Error("Create_OrderItems_Table(): ctdbAllocTable()");

   if (ctdbOpenTable(hTableOrdrItem, "ordritem", CTOPEN_NORMAL))
   {
      /* define table fields */
      pField0 = ctdbAddField(hTableOrdrItem,"rectype", CT_FSTRING, 1);
      pField1 = ctdbAddField(hTableOrdrItem,"oi_ordrnumb", CT_FSTRING, 4);
      pField2 = ctdbAddField(hTableOrdrItem,"oi_sequnumb", CT_INT2, 2);
      pField3 = ctdbAddField(hTableOrdrItem,"oi_quantity", CT_INT2, 2);
      pField4 = ctdbAddField(hTableOrdrItem,"oi_itemnumb", CT_FSTRING, 5);
      if (!pField1 || !pField2 || !pField3 || !pField4)
         Handle_Error("Create_OrderItems_Table(): ctdbAddField()");

      /* create table */
      if (ctdbCreateMRTTable(hTableOrdrItem, "ordritem","tutorial_host",
CTCREATE_NORMAL|CTCREATE_NONULFLD,"rectype==\"I\""))
         Handle_Error("Create_OrderItems_Table(): ctdbCreateTable()");
   }
   else
         ctdbCloseTable(hTableOrdrItem);
}

/*
 * Create_ItemMaster_Table()
 *
 * Open table ItemMaster, if it exists. Otherwise create it
 * along with its indices and open it
 */

#ifdef PROTOTYPE
VOID Create_ItemMaster_Table(VOID)
```

```
#else
VOID Create_ItemMaster_Table()
#endif
{
   CTHANDLE pField0, pField1, pField2, pField3, pField4;

   /* define table ItemMaster */
   printf("\ttable ItemMaster\n");

   /* allocate a table handle */
   if ((hTableItemMast = ctdbAllocTable(hDatabase)) == NULL)
      Handle_Error("Create_ItemMaster_Table(): ctdbAllocTable()");

   /* open table */
   if (ctdbOpenTable(hTableItemMast, "itemmast", CTOPEN_NORMAL))
   {
      /* define table fields */
      pField0 = ctdbAddField(hTableItemMast,"rectype", CT_FSTRING, 1);
      pField1 = ctdbAddField(hTableItemMast, "im_itemnumb", CT_FSTRING, 4);
      pField2 = ctdbAddField(hTableItemMast, "im_itemwght", CT_INT4, 4);
      pField3 = ctdbAddField(hTableItemMast, "im_itempric", CT_MONEY, 4);
      pField4 = ctdbAddField(hTableItemMast, "im_itemdesc", CT_STRING, 47);
      if (!pField1 || !pField2 || !pField3 || !pField4)
         Handle_Error("Create_ItemMaster_Table(): ctdbAddField()");

      /* create table */
      if (ctdbCreateMRTTable(hTableItemMast, "itemmast","tutorial_host",
CTCREATE_NORMAL|CTCREATE_NONULFLD,"rectype==\"M\""))
         Handle_Error("Create_ItemMaster_Table(); ctdbCreateTable()");
   }
   else
         ctdbCloseTable(hTableItemMast);
}

/*
 * Check_Table_Mode()
 *
 * Check if existing table has transaction processing flag enabled.
 * If a table is under transaction processing control, modify the
 * table mode to disable transaction processing
 */

#ifdef PROTOTYPE
VOID Check_Table_Mode(CTHANDLE hTable)
#else
VOID Check_Table_Mode(hTable)
CTHANDLE hTable;
#endif
{
   CTCREATE_MODE mode;

   /* get table create mode */
   mode = ctdbGetTableCreateMode(hTable);

   /* check if table is under transaction processing control */
   if ((mode & CTCREATE_TRNLOG))
   {
      /* change file mode to disable transaction processing */
      mode ^= CTCREATE_TRNLOG;
      if (ctdbUpdateCreateMode(hTable, mode) != CTDBRET_OK)
         Handle_Error("Check_Table_Mode(); ctdbUpdateCreateMode");
   }
}

/*
 * Add_CustomerMaster_Records()
 *
 * This function adds records to table CustomerMaster from an
 * array of strings
 */

typedef struct {
   CTSTRING type, number, zipcode, state, rating, name, address, city;
```

```
} CUSTOMER_DATA;

CUSTOMER_DATA data4[] = {
   "C", "1000", "92867", "CA", "1", "Bryan Williams", "2999 Regency",      "Orange",
   "C","1001", "61434", "CT", "1", "Michael Jordan", "13 Main",           "Harford",
   "C","1002", "73677", "GA", "1", "Joshua Brown",   "4356 Cambridge",    "Atlanta",
   "C","1003", "10034", "MO", "1", "Keyon Dooling",  "19771 Park Avenue", "Columbia"
};

#ifdef PROTOTYPE
VOID Add_CustomerMaster_Records(VOID)
#else
VOID Add_CustomerMaster_Records()
#endif
{
   CTDBRET  retval;
   CTSIGNED i;
   CTSIGNED nRecords = sizeof(data4) / sizeof(CUSTOMER_DATA);

   if ((hRecordCustMast = ctdbAllocRecord(hTableCustMast)) == NULL)
      Handle_Error("Add_Customer_Records(): ctdbAllocRecord()");

   Delete_Records(hRecordCustMast);

   printf("\tAdd records in table CustomerMaster...\n");

   /* add records to table */
   for (i = 0; i < nRecords; i++)
   {
      /* clear record buffer */
      ctdbClearRecord(hRecordCustMast);

      retval = 0;
      /* populate record buffer with data */
      retval |= ctdbSetFieldAsString(hRecordCustMast, 0, data4[i].type);
      retval |= ctdbSetFieldAsString(hRecordCustMast, 1, data4[i].number);
      retval |= ctdbSetFieldAsString(hRecordCustMast, 2, data4[i].zipcode);
      retval |= ctdbSetFieldAsString(hRecordCustMast, 3, data4[i].state);
      retval |= ctdbSetFieldAsString(hRecordCustMast, 4, data4[i].rating);
      retval |= ctdbSetFieldAsString(hRecordCustMast, 5, data4[i].name);
      retval |= ctdbSetFieldAsString(hRecordCustMast, 6, data4[i].address);
      retval |= ctdbSetFieldAsString(hRecordCustMast, 7, data4[i].city);

      if(retval)
         Handle_Error("Add_Customer_Records(): ctdbSetFieldAsString()");

      /* add record */
      if (ctdbWriteRecord(hRecordCustMast))
         Handle_Error("Add_Customer_Records(): ctdbWriteRecord()");
   }
}

/*
 * Add_CustomerOrders_Records()
 *
 * This function adds records to table CustomerOrders from an
 * array of strings
 */

typedef struct {
   CTSTRING type, orderdate, promisedate, ordernum, customernum;
} ORDER_DATA;

ORDER_DATA data1[] = {
   {"O", "09/01/2002", "09/05/2002",  "1",  "1001"},
   {"O", "09/02/2002", "09/06/2002",  "2",  "1002"}
};

#ifdef PROTOTYPE
VOID Add_CustomerOrders_Records(VOID)
#else
VOID Add_CustomerOrders_Records()
#endif
```

```
{
   CTDBRET  retval;
   CTSIGNED i;
   CTSIGNED nRecords = sizeof(data1) / sizeof(ORDER_DATA);
   CTDATE   orderdate;
   CTDATE   promisedate;

   if ((hRecordCustOrdr = ctdbAllocRecord(hTableCustOrdr)) == NULL)
      Handle_Error("Add_CustomerOrders_Records(): ctdbAllocRecord()");

   Delete_Records(hRecordCustOrdr);

   printf("\tAdd records in table CustomerOrders...\n");

   /* add records to table */
   for (i = 0; i < nRecords; i++)
   {
      /* clear record buffer */
      ctdbClearRecord(hRecordCustOrdr);

      retval = 0;
      retval |= ctdbStringToDate(data1[i].orderdate, CTDATE_MDCY ,&orderdate);
      retval |= ctdbStringToDate(data1[i].promisedate, CTDATE_MDCY, &promisedate);
      /* populate record buffer with data */
      retval |= ctdbSetFieldAsString(hRecordCustOrdr, 0, data1[i].type);
      retval |= ctdbSetFieldAsString(hRecordCustOrdr, 1, data1[i].ordernum);
      retval |= ctdbSetFieldAsDate(hRecordCustOrdr, 2, orderdate);
      retval |= ctdbSetFieldAsDate(hRecordCustOrdr, 3, promisedate);
      retval |= ctdbSetFieldAsString(hRecordCustOrdr, 4, data1[i].customernum);

      if (retval)
         Handle_Error("Add_CustomerOrders_Records():
ctdbSetFieldAsString()|ctdbSetFieldAsDate()");

      /* add record */
      if (ctdbWriteRecord(hRecordCustOrdr))
         Handle_Error("Add_CustomerOrders_Records(): ctdbWriteRecord()");
   }
}

/*
 * Add_OrderItems_Records()
 *
 * This function adds records to table OrderItems from an
 * array of strings
 */

typedef struct {
   CTSTRING type;
   COUNT sequencenum, quantity;
   CTSTRING ordernum, itemnum;
} ORDERITEM_DATA;

ORDERITEM_DATA data2[] = {
   {"I", 1,   2, "1",  "1"},
   {"I", 2,   1, "1",  "2"},
   {"I", 3,   1, "1",  "3"},
   {"I", 1,   3, "2",  "3"}
};

#ifdef PROTOTYPE
VOID Add_OrderItems_Records(VOID)
#else
VOID Add_OrderItems_Records()
#endif
{
   CTDBRET  retval;
   CTSIGNED i;
   CTSIGNED nRecords = sizeof(data2) / sizeof(ORDERITEM_DATA);

   if ((hRecordOrdrItem = ctdbAllocRecord(hTableOrdrItem)) == NULL)
      Handle_Error("Add_OrderItems_Records(): ctdbAllocRecord()");
```

```
   Delete_Records(hRecordOrdrItem);

   printf("\tAdd records in table OrderItems...\n");

   /* add records to table */
   for (i = 0; i < nRecords; i++)
   {
      /* clear record buffer */
      ctdbClearRecord(hRecordOrdrItem);

      retval = 0;
      /* populate record buffer with data */
      retval |= ctdbSetFieldAsString(hRecordOrdrItem, 0, data2[i].type);
      retval |= ctdbSetFieldAsString(hRecordOrdrItem, 1, data2[i].ordernum);
      retval |= ctdbSetFieldAsSigned(hRecordOrdrItem, 2, data2[i].sequencenum);
      retval |= ctdbSetFieldAsSigned(hRecordOrdrItem, 3, data2[i].quantity);
      retval |= ctdbSetFieldAsString(hRecordOrdrItem, 4, data2[i].itemnum);

      if(retval)
         Handle_Error("Add_OrderItems_Records(): ctdbSetFieldAsString()");

      /* add record */
      if (ctdbWriteRecord(hRecordOrdrItem))
         Handle_Error("Add_OrderItems_Records(): ctdbWriteRecord()");
   }
}

/*
 * Add_ItemMaster_Records()
 *
 * This function adds records to table ItemMaster from an
 * array of strings
 */

typedef struct {
   CTSTRING type;
   CTSIGNED weight;
   CTMONEY  price;
   CTSTRING itemnum, description;
} ITEM_DATA;

ITEM_DATA data3[] = {
   {"M", 10,  1995, "1",  "Hammer"},
   {"M", 3,   999,  "2",  "Wrench"},
   {"M", 4,   1659, "3",  "Saw"},
   {"M", 1,   398,  "4",  "Pliers"}
};

#ifdef PROTOTYPE
VOID Add_ItemMaster_Records(VOID)
#else
VOID Add_ItemMaster_Records()
#endif
{
   CTDBRET  retval;
   CTSIGNED i;
   CTSIGNED nRecords = sizeof(data3) / sizeof(ITEM_DATA);

   if ((hRecordItemMast = ctdbAllocRecord(hTableItemMast)) == NULL)
      Handle_Error("Add_ItemMaster_Records(): ctdbAllocRecord()");

   Delete_Records(hRecordItemMast);

   printf("\tAdd records in table ItemMaster...\n");

   /* add records to table */
   for (i = 0; i < nRecords; i++)
   {
      /* clear record buffer */
      ctdbClearRecord(hRecordItemMast);

      retval = 0;
      /* populate record buffer with data */
```

```
      retval |= ctdbSetFieldAsString(hRecordItemMast, 0, data3[i].type);
      retval |= ctdbSetFieldAsString(hRecordItemMast, 1, data3[i].itemnum);
      retval |= ctdbSetFieldAsSigned(hRecordItemMast, 2, data3[i].weight);
      retval |= ctdbSetFieldAsMoney(hRecordItemMast, 3, data3[i].price);
      retval |= ctdbSetFieldAsString(hRecordItemMast, 4, data3[i].description);

      if(retval)
         Handle_Error("Add_ItemMaster_Records():
ctdbSetFieldAsString()|ctdbSetFieldAsSigned()");

      /* add record */
      if (ctdbWriteRecord(hRecordItemMast))
         Handle_Error("Add_ItemMaster_Records(): ctdbWriteRecord()");
   }
}


/*
 * Delete_Records()
 *
 * This function deletes all the records in the table based
 * on the input parameter
 */

#ifdef PROTOTYPE
VOID Delete_Records(CTHANDLE hRecord)
#else
VOID Delete_Records(hRecord)
CTHANDLE hRecord;
#endif
{
   CTDBRET  retval;
   CTBOOL   empty;

   printf("\tDelete records...\n");

   if (ctdbClearRecord(hRecord))
      Handle_Error("Delete_Records(): ctdbClearRecord()");

   empty = NO;
   retval = ctdbFirstRecord(hRecord);
   if (retval != CTDBRET_OK)
   {
      if (retval == END_OF_FILE)
         empty = YES;
      else
         Handle_Error("Delete_Records(): ctdbFirstRecord()");
   }

   while (empty == NO) /* while table is not empty */
   {
      /* delete record */
      if (ctdbDeleteRecord(hRecord))
         Handle_Error("Delete_Records(): ctdbDeleteRecord()");

      /* read next record */
      retval = ctdbNextRecord(hRecord);
      if (retval != CTDBRET_OK)
      {
         if (retval == END_OF_FILE)
            empty = YES;
         else
            Handle_Error("Delete_Records(): ctdbNextRecord()");
      }
   }
}

/*
 * Handle_Error()
 *
 * This function is a common bailout routine. It displays an error message
 * allowing the user to acknowledge before terminating the application
 */
```

```
#ifdef PROTOTYPE
VOID Handle_Error(CTSTRING errmsg)
#else
VOID Handle_Error(errmsg)
CTSTRING errmsg;
#endif
{
   printf("\nERROR: [%d] - %s \n", ctdbGetError(hSession), errmsg);
   printf("*** Execution aborted *** \nPress <ENTER> key to exit...");

   ctdbLogout(hSession);

   ctdbFreeRecord(hRecordCustMast);
   ctdbFreeRecord(hRecordItemMast);
   ctdbFreeRecord(hRecordOrdrItem);
   ctdbFreeRecord(hRecordCustOrdr);

   ctdbFreeTable(hTableCustMast);
   ctdbFreeTable(hTableItemMast);
   ctdbFreeTable(hTableOrdrItem);
   ctdbFreeTable(hTableCustOrdr);

   ctdbFreeSession(hSession);

   getchar();

   exit(1);
}
/* end of ctdb_tutorial2.c */
```

Index