

An Informal Introduction to MemCom

Table of Contents

1 The MemCom Database.....	2
1.1 Physical Layout.....	2
1.2 Database Exchange.....	2
1.3 Adding More Data.....	2
1.4 The Logical Layout.....	2
1.5 Inspecting Databases with the MemCom Browser.....	3
1.6 Querying Data Sets.....	3
1.7 Iterating Through the Index.....	4
2 Working with Datasets.....	4
2.1 Saving an Array of Doubles as a Dataset.....	4
2.2 Loading an Array of Doubles From a Dataset.....	4
3 Descriptive Data.....	5
3.1 Annotating a Data Set with a Descriptor.....	5
3.2 Loading a Data Set Descriptor.....	5
4 Wrap Up – What You Can Do Now.....	6
5 What We Did Not Yet Talk About.....	6

This document gives you an impression of what you can do with *MemCom* from the point of view of application programming. In the following examples we use the C programming language; Similar examples could be made with Fortran and Python. Note that the complete documentation of *MemCom* is available on-line. Have a look at

- The *MemCom* Database: Explains how to make use of *MemCom* to save array data to disk and to load array data from disk.
- We take a look how *MemCom* databases are physically and logically organized. Once a database contains more than just a few datasets, you would like to query dataset attributes and to obtain the list of datasets.
- Adding descriptive data is necessary to give data a meaning, that is, how to interpret it. *MemCom* descriptors are a straightforward way to get self-descriptive data sets.

- A wrap up summarizes this short introduction.
- Some of the advanced features, which are not always needed, but might be useful.

1 The MemCom Database

1.1 Physical Layout

MemCom stores data under a specific directory in the current file system. Example: The directory `/tmp/toto.mc` is a *MemCom* database. It contains the following files:

```
-rw-r--r-- 1 foo wheel 8000000 Dec 18 11:24 data.1
-rw-r--r-- 1 foo wheel          94 Dec 18 11:24 header
-rw-r--r-- 1 foo wheel 221184 Dec 18 11:24 index
```

The file `data.1` contains the saved data, while the `header` and `index` files are used to describe the contents of the database. Note that all data is saved in binary format, thus making *MemCom* a faster and more space-efficient solution than text-files.

1.2 Database Exchange

MemCom databases may be exchanged between different hardware platforms without any conversion; compressing a database before exchanging it can save time when copying the database.

1.3 Adding More Data

When working with *MemCom*, you can add new data sets any time to an open database. Suppose you have an array of integers

```
int array[12345]
```

then the call

```
mcDBputSet(handle, "new_data_set", "I", 12345, array);
```

creates a new data set. The data set name must be unique within the current *MemCom* database, since data sets are identified by their names.

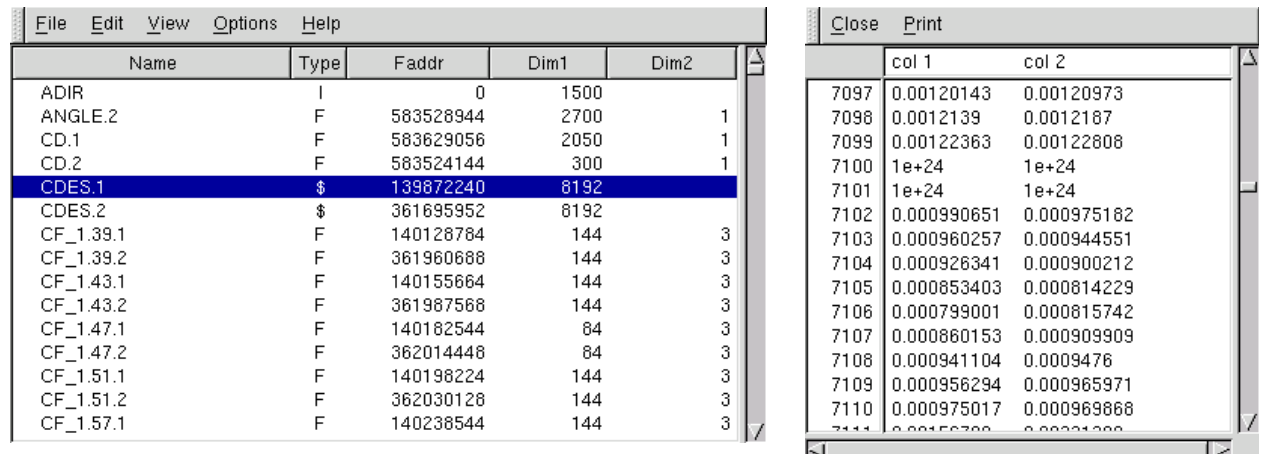
No matter how big this array is and no matter the number or the size of the previously created data sets, there is no performance penalty when adding new data sets. This is very important, because this way you can have hundreds of thousands of data sets in a single database, the data set and database sizes only being limited by your hardware's and the operating system's capabilities. This avoids the usual cluttering of directories with many files that depend on each other.

1.4 The Logical Layout

Each data set is identified by its name. The data set names and attributes are stored in an index. For efficiency reasons there is only a global index. Given the data set's name, the data set attributes and data are looked up via the index. Hierarchical layouts can be achieved by organizing dataset names with specific fields, separated by a separator, such as the dot.

1.5 Inspecting Databases with the MemCom Browser

To display the contents of a *MemCom* database, launch the *mcBrowser* tool, which is shipped with *MemCom*. The example below shows the directory of a database. i.e. the browser main window (left), displaying the dataset names and set attributes in alphabetical order. Clicking with the middle mouse button on a data set will display the contents of the data set (right).



Name	Type	Faddr	Dim1	Dim2
ADIR	I	0	1500	
ANGLE.2	F	583528944	2700	1
CD.1	F	583629056	2050	1
CD.2	F	583524144	300	1
CDES.1	\$	139872240	8192	
CDES.2	\$	361695952	8192	
CF_1.39.1	F	140128784	144	3
CF_1.39.2	F	361960688	144	3
CF_1.43.1	F	140155664	144	3
CF_1.43.2	F	361987568	144	3
CF_1.47.1	F	140182544	84	3
CF_1.47.2	F	362014448	84	3
CF_1.51.1	F	140198224	144	3
CF_1.51.2	F	362030128	144	3
CF_1.57.1	F	140238544	144	3

	col 1	col 2
7097	0.00120143	0.00120973
7098	0.0012139	0.0012187
7099	0.00122363	0.00122808
7100	1e+24	1e+24
7101	1e+24	1e+24
7102	0.000990651	0.000975182
7103	0.000960257	0.000944551
7104	0.000926341	0.000900212
7105	0.000853403	0.000814229
7106	0.000799001	0.000815742
7107	0.000860153	0.000909909
7108	0.000941104	0.0009476
7109	0.000956294	0.000965971
7110	0.000975017	0.000969868

mcBrowser loads only the portion of data from the database which must be displayed, allowing for instant display of portions of large datasets, typically hundreds of millions of bytes.

1.6 Querying Data Sets

Often one needs to know if a data set exists from within a program. The function `mcDBinqSetAtt` returns data set attributes if the set exists:

```
if mcDBinqSetAtt(handle, "my_set", NULL) == 1)
    print("data set exists.\n");
```

To obtain the dimensions and element type of a data set, the same call is used, but with the third argument being a pointer to a `mcDBinqSetAtt` structure containing all data set attributes. The following example shows how to print out the contents of a data set of type double:

```
mcSetAttributes att;
double* data;
int i;

/* Inquire set and get set size */
mcDBinqSetAtt(handle, "my_set", &att);

/* Allocate memory */
data = malloc(sizeof(double) * att.size);

/* Read set from database */
mcDBgetSetArray(handle, "my_set", 0, data);
```

```

/ Print */
for (i = 0; i < att.size; i++)
    print("%f\n", data[i]);
free(data);

```

If we allocate the memory ourselves we can use the array version of the get function. It does not allocate memory, but it takes the pointer to the allocated memory area as additional argument.

1.7 Iterating Through the Index

To find out what data sets are stored in a *MemCom* database, the call to `mcDBgetSetNextIter` is used to iterate through the table of contents. The example below prints the names of all data sets:

```

mcSetAttributes att;
int i = 0; /* gets first data set */
while (mcDBgettNextIter(handle, &i, &att) == 1)
    printf("#%37s\n", att.name);

```

2 Working with Datasets

2.1 Saving an Array of Doubles as a Dataset

Suppose you have allocated the following array of double-precision floating-point numbers

```
double data[size];
```

which, after some calculations, contains now data that you would like to save to disk:

```

int handle;
char fname = "/path/to/the/memcom-db;
handle = mcDBopenFile>(fname, "rw");
mcDBputSet(handle, "my_name", "F", size, data);
mcDBcloseFile(handle);

```

We open the *MemCom* database in read and write mode. In this case, *MemCom* automatically creates a new database if the database does not yet exist. "F" indicates the data type (referring to the C data type `double`). `size` is then number of elements of the array.

2.2 Loading an Array of Doubles From a Dataset

At a later stage in the current program or in a different program, you might want to retrieve the saved data:

```

int handle;
double* data;
char fname = "/path/to/the/memcom-db;
handle = mcDBopenFile(fname, "r");
data = mcDBgetSet(handle, "my_name", 0);
mcDBcloseFile(handle);

```

The array `data` now contains the data that were previously saved. In this case, the memory needed to hold the data is allocated by *MemCom* during the call to `mcDBgetSet`.

The size argument 0 tells *MemCom* to load the whole array. When the array data is no longer needed, its memory can be freed:

```
mcDBfree(data);
```

In these examples we always opened and closed the *MemCom* database. If you perform more than a single *MemCom* operation in a program, you should leave the database open and work with the same database handle. Subsequently, we just assume that `handle` designates a *MemCom* database opened for reading and writing.

3 Descriptive Data

So far we have worked with so-called positional data sets, that is, array tables of basic data types. Now we introduce *MemCom*'s concept of relational tables, also referred to as dictionaries.

3.1 Annotating a Data Set with a Descriptor

Frequently, descriptive data need to be stored together with data sets. Since this data is generally only small, it is grouped together in descriptors. Each data set can have a descriptor, which is a dictionary (or a set of key-value pairs).

```
/* Create a table object */
mcRTable* rt;

/* Create an empty relational table object in main memory */
rt = mcTBcreateDesc();

/* Insert the key-value pair "UNIT"="mm" in the relational
table. The key is always a string, while the value may be of
type string or a numerical.*/
mcTBinsK("UNIT", "mm", rt);

/* Store the relational table as descriptor of the data set */
mcTBstoreDesc(handle, "my_set", rt);

/* Free the relational table object */
mcTBfree(rt);
```

Note that only the `mcTBstoreDesc` function operates on the database, all other functions operate in main memory, thus avoiding latency problems: It is generally not desirable to access the database (that is, the file system, and eventually the disk) with very small portions of data. The `mcTBins` functions manipulate the data in main memory, while `mcTBstoreDesc` copies the whole relational table to disk, minimizing performance losses due to I/O latencies. Finally, `mcTBfree` releases the table object in memory, freeing the occupied space.

3.2 Loading a Data Set Descriptor

The following example loads a data set descriptor and prints the key, type, and size of all items:

```
mcRTable* rt;
if ((rt = mcTBloadDesc(handle, "my_set")) != NULL) {
```

```

/* descriptor exists, print contents */
char name[64];
char type[4];
int size;
int i = 0; /* gets first key */
while (mcTBnextObjIter(i, name, type, &size;, rt))
    print("name=%s type=%s size=%d\n", name, type, size);
}

```

4 Wrap Up – What You Can Do Now

By making use of the *MemCom* calls described earlier, you should now already be able to use *MemCom* for

- Opening and closing MemCom databases
- Saving an array to a data set
- Loading a data set into an array
- Adding more datasets to a database
- Inquiring data set attributes
- Iterating through the list of data sets in a database
- Exchanging MemCom databases with others working on different platforms
- Browsing databases with the MemCom browser
- Adding descriptive data to data sets
- Reading data set descriptors

5 What We Did Not Yet Talk About

Of course, this introduction covers only some - although important- aspects of *MemCom*. Among the many other features of *MemCom* we can mention

- Customized error handling: *MemCom* error messages are descriptive and help you to point out the problem that caused the error. The `mcErrTermLevel` and `mcErrSetOutput` functions permit customisation of *MemCom*'s error handling.
- Read and write only parts of a data set: Sometimes only a part of a large data set needs to be read or written, see the `mcDBgetSetpos` and `mcDBputSetpos` functions.
- Multi-dimensional data sets: Matrices are the most frequent case of two-dimensional data structures. The `mcDBgetSubset` and `mcDBputSubset` calls permit I/O on the sub-set level. *MemCom* supports up to five dimensions. The `mcDBgetSetslice` and `mcDBputSetslice` functions implement the concept of slices for *MemCom* datasets.
- Relational table data sets: Descriptive data can be stored in the dataset itself. See the `mcTBload` and `mcTBstore` functions.
- Minimizing I/O latency: When doing I/O on very small datasets or small portions of

datasets, the operating system's overhead becomes a bottle-neck. *MemCom*'s paging mode, initiated by the function `mcDBopenFileBuffered`, implements an efficient cache mechanism, a feature not present in other data managers. In paging mode, the system I/O is minimized.

-
- Remote access: The client/server mode permits access to remote *MemCom* databases over a TCP/IP network. Whether a database is accessed locally or remotely, is transparent to the application. The `mcDBopenFile` function allows for opening a database in stand-alone or client mode.
- Concurrent access: A simple but efficient transaction mechanism for the client/server mode helps to maintain database integrity when multiple clients access a database concurrently. See `mcBegin` and `mcEnd`.