Extreme Database programming with MUMPS Globals

Chapter 1

Globals: An alternative to the relational view

The really important heart of MUMPS is its data storage mechanism. This is based on what are known as Global Variables, more commonly known simply as Globals. Globals are an incredibly simple concept, and yet incredibly powerful.

Many people get turned off by Globals. They are a primitive structure. There are none of the controls, safety nets or value-added mechanisms that “proper” databases provide. As a result, MUMPS is often dismissed as irrelevant, insufficient or somehow, just plain wrong. To do so is to ignore a data storage mechanism that is lean, mean and totally malleable. In the right hands, that lack of baggage and overhead can be an incredibly liberating experience. In the wrong hands it can be a recipe for disaster. It’s a bit like a dangerous, extreme sport such as free-form mountain climbing. Few “proper” sportspeople will recommend it, but it’s the fastest, most efficient and most exhilarating way up that mountain if you can master the skills to climb without the safety-related paraphernalia. The climbing equipment companies won’t recommend it either, because it questions the need for their products!

So if you think you’re up to the challenge, and ready to accept that there may be another way to consider data storage than the entrenched relational and SQL view of the world, let’s dig deeper.

This first chapter will summarise the basics of Globals. The second chapter focuses on their use in terms that a RDBMS/SQL programmer will be familiar. If you wish, skip to Chapter 2 now and return to Chapter 1 to familiarise yourself with the basics later.

All MUMPS systems and their modern derivatives, such as Caché, use Globals as the basis of their storage mechanism. Many modern MUMPS systems and/or value-added packages layer more “conventional” views onto the core Global construct, eg a global-based system can be layered to present a logical Object database, relational database or native XML database characteristics. Indeed the same physical Global-based database could potentially be logically presented and viewed in more than one, if not all, of these ways. As a result, there are growing numbers of developers who are unaware that Globals are there under the covers, and unaware of what they are, how they work and how you can use them. This paper will help you discover this hidden and, we hope you’ll agree, very cool world.

So what are Globals?

Put simply, a Global is a persistent, sparse, dynamic, multi-dimensional array, containing a text value. Actually MUMPS allows the use of both persistent and non-persistent multi-dimensional arrays, the latter known as "local arrays".

Somewhat unusually, MUMPS allows the use of both numeric and textual subscripting. So in MUMPS, you can have an array such as:

```
Employee(company,country,office,employeeNumber) = employeeDetails
```

eg

```
Employee("MGW","UK","London",1) = "Rob Tweed`Director`020 8404 3054"
```

In this example, the data items that make up the employee details (name, position, telephone number) have been appended together with the back-apostrophe character as a delimiter. MUMPS does not impose any controls or rules over how you construct your data structures : there's no schema or data dictionary for describing your database records. This gives you incredible flexibility and speed of development. You can arbitrarily assign one or more data delimiters to break the text value of an array record into a number of "fields". The total string length that can be assigned to a single array record depends on the MUMPS implementation, but in Caché it is up to 32k. The string length that is actually stored is variable, and you can see that by using a delimiter character, individual fields are variable length also. This makes MUMPS globals a very efficient data storage mechanism : there's almost no disc space holding empty, unused spaces.

Now in the example above, the employee record will be held in what is known as a "Local Array". If you were to exit from your MUMPS session, the array would disappear, just like a PHP array once the page or session has gone.

Now here's the fun bit. To store this employee record permanently to disc, ie as a Global, just add a "^" in front of the array name:

```
^Employee(company,country,office,employeeNumber) = employeeDetails
```

eg

```
^Employee("MGW","UK","London",1) = "Rob Tweed`Director`020 8404 3054"
```

That's all there is to it!

So, to create such a global node, you would use the MUMPS command "set", ie:

```
set ^Employee(“MGW”,”UK”,”London”,1) = “Rob Tweed`Director`020 8404 3054”
```

Something that confuses and frightens people about MUMPS is that most commands can be abbreviated down to a single letter (in upper or lower case), so you’ll often see the following instead:

```
s ^Employee(“MGW”,”UK”,”London”,1) = “Rob Tweed`Director`020 8404 3054”
```

Now when you exit from MUMPS, the record will persist on disc permanently. When you come back at any time in the future, you can retrieve the record straight off the disc by using the global reference:

```
^Employee(“MGW”,”UK”,”London”,1)
```

To recover a global node within your programs, you typically use the “set” command to assign the value held in a global record (or “node”) to a local variable, eg:

```
 Set companyRecord= ^Employee(“MGW”,”UK”,”London”,1)
```

The variable companyRecord will now contain the string value:

```
“Rob Tweed`Director`020 8404 3054”
```

Globals can have as many subscripts as you like, and the subscripts can be any mixture of text and numbers (real or integers). String subscripts must be surrounded by double quotes, while numbers are not.

Most MUMPS implementations have practical limits over the total subscript length, so you can’t go totally overboard, but you’ll find that it will cater for all of your requirements.

To delete a global node, you use the “kill” command, eg:

```
Kill ^Employee(“MGW”,”UK”,”London”,1)
```

That’s literally all there is to Globals. The real trick is how to make such a primitive data structure work for you. That’s what the rest of this paper will do. We’ll attempt to do this in such a way that the relational database programmer can understand the equivalent techniques and representations that a MUMPS database practitioner would use.

Remember that there’s nothing in MUMPS that will enforce a particular database design methodology, and its up to you to add the controls and checks that will ensure your database is logically consistent and error-free. That means you’ll be doing a lot of work that a more conventional DBMS would otherwise do for you, but you’ll soon find that you can automate the most repetitive tasks, and make light work of the management of a MUMPS-based database.

Creating a simple multi-level hierarchy

You may have multiple levels of subscripting simultaneously in your globals, eg:

```
^Employee(“MGW”,”UK”,”London”) = 2
^Employee(“MGW”,”UK”,”London”,1) = “Rob Tweed`Director`020 8404 3054”
^Employee(“MGW”,”UK”,”London”,2) = “Chris Munt`Director`01737 371457”
```

Here we’re specifying the number of employees at a given office at the third level of subscripting, and the actual employee record at the fourth level.

Links between different globals are for the programmer to define. Once again MUMPS provides no automatic indexing or cross-linking itself.

Hence, we could have a telephone number global that acts as an index to the employee global, eg:

```
^EmployeeTelephone(“020 8404 3054) = “MGW`UK`London`1”
^EmployeeTelephone(“01737 371457”) = “MGW`UK`London`2”
```

In this example, the data value stored against each telephone number holds the subscripts for the associated employee record, concatenated together. By knowing the telephone number, all we’d have to do is break the data value apart using the back apostrophe as a delimiter, and we could retrieve the main employee record.

For example:

```
S telNo=”020 8404 3054”
S indexData=^EmployeeTelephone(telNo)
S company=$piece(indexData,”`”,1)
S country=$piece(indexData,”`”,2)
S office=$piece(indexData,”`”,3)
S employeeNo=$piece(indexData,”`”,4)
S record=^Employee(company,country,office,employeeNo)
S employee=$piece(record,”`”,1)
```

Note the use of the $piece MUMPS function to separate out the individual “pieces” in the index data string and the employee record string.

One of the great things is that nothing in MUMPS has to be pre-declared. You decide when and how to create, modify or delete global records – it’s all automatically and dynamically handled for you. You can add further pieces to your global at any time without any need for declarations whatsoever. If you want to start using another global, just start using it and it will be created dynamically for you.

Setting, Getting and Deleting Globals

To summarise, in MUMPS, globals are created and retrieved using the “Set” command, and deleted using the “Kill” command.

1) Creating a global record:

```
Set ^Employee(“MGW”,”UK”,”London”,1)=“Rob Tweed`Director`020 8404 3054”
```

This creates the global reference, saving the record to disc.

2) Retrieving a global record

```
Set data=^Employee(“MGW”,”UK”,”London”,1)
```

This retrieves the specified global and places the data value into a local variable named “data”.

3) Deleting a global record:

```
kill ^Employee(“MGW”,”UK”,”London”,1)
```

This permanently and immediately deletes the specified global record from disc. *Be very careful with the Kill command* – it’s both extremely simple to use and incredibly dangerous. If you specify fewer subscripts, all lower-level subscripts will be deleted. If you specify no subscripts at all, the entire global will be deleted, eg:

```
Kill ^Employee(“MGW”,”UK”)
```

This will delete all records for all UK offices

```
Kill ^Employee
```

This will delete the entire ^Employee global, immediately, permanently and irretrievably, unless you’d backed it up.

Traversing a Global

One of the most frequent things you need to do is traverse some or all of a global. For example, let's say you want to manipulate all the employee records, perhaps to display a list so that the user can select one of them, or to count them. To do this, you use the $order MUMPS function. The $order function is one of the "crown jewels" of MUMPS, allowing, with incredible simplicity, access to any of the data you store in your globals. It's not functionality that will be intuitive to a "traditional" database programmer, but it's worth understanding because it is so powerful and yet so simple.

The $order function operates on a single subscript level within a global, and returns the next subscript value in collating sequence that exists at that level in the global. You can specify some starting value, and the $order function will find the next value that exists in collating sequence. To find the first subscript at the specified level, use a starting value of null (""). To find the first subscript at the specified level that starts with "C", use a starting value that collates just before "C", for example "B~"

So, to find the first company in our Employee global :

```
S company=""
S company=$order(^Employee(company))
```

The variable company now contains the value of the company subscript in the first record in the ^Employee global.

When the last value is found, the next time the $order function is run, it will return a null value. So, for example, if there was only one company in the global, if we repeat that $order, ie:

```
S company=$order(^Employee(company))
```

then the variable company will now contain a null value (ie "")

To get and process all companies in the Employee global, we would create a loop:

```
S company=""
For  s company=$order(^Employee(company)) quit:company=""  do
. ; do something with the company
```

This demonstrates some of the interesting features of the brevity of MUMPS coding:

- The For command followed by two spaces sets up an infinite loop.
- The Quit:company="" provides the termination condition for the loop, and uses a construct known as the "post-conditional". What this construct is saying is "if the value of company is null, then quit the For loop". The Quit must be followed by two spaces if any other command is written after it.

- The “do” at the end of the line means execute an code that follows at the next “dot” level. The “do” will get executed on every iteration of the For loop where company is non-null
- The code that is to be executed on every loop is written after a single dot. In essence, any lines starting with a dot constitute a subroutine that is executed by that “do” at the end of the second line in the example.

So, we seed the $order with a null to make sure that it starts by finding the first subscript value that is stored in the global. We loop through each value until we exhaust the values that are saved, in which case we get a null value returned by the $order. We then detect this null value and terminate the loop.

You’ll find that this kind of loop is one of the most common things you’ll find in a MUMPS routine.

We can extend this to traverse the entire global. To do this we loop through each subscript level, starting at the first, and progressively moving to the next one:

```
s company=””
for  s company=$order(^Employee(company)) quit:company=””  do
. s country=””
.  for  s country=$order(^Employee(company,country)) quit:country=””  do
. . s office=””
. . for  s office=$order(^Employee(company,country,office)) quit:office=””  do
. . . s employeeNo=””
. . . for  s employeeNo=$order(^Employee(company,country,office,employeeNo))
          quit:employeeNo=””  do
. . . . s record=^Employee(company,country,office,employeeNo)
. . . . ; do something with the record
```

Note here the way we’ve nested the “dot” levels to create a hierarchy of nested subroutines. Also note how the $order is being used to ultimately provide values for the four subscripts of the global, so that at the inside of the loop hierarchy, we can process each and every record on file.

If we wanted to find and process only those employees in companies starting with “C”, then one common method is as follows:

```
s company=”B~”
f  s company=$o(^Employee(company)) quit:$e(company,1)’=”C”  do
. ; do something with the record
```

Note the use of the $extract (or $e) function. This extracts the character from the specified position in the string value : in this case the first character of the company value. Also note way that "not equals C" is specified : this uses the MUMPS *NOT* operator which is a single quote (').

So read this loop as:

- Seed the $order with a value that just preceeds "C" in collating sequence
- Set up an infinite loop to find all company records in collating sequence
- If the first character of the company name no longer starts with a "C", then exit the loop
- Otherwise, process the company record.

This ability to start and stop traversing or parsing a global at any place in the collating sequence, and at any level in the hierarchy of subscripts is pretty much unique to MUMPS globals.

Checking to see if a Global node exists

You'll often want to know if a particular global node exists. You can use the MUMPS $data function for this, eg:

```
        if $data(^Employee(company)) do xxx
```

Read this as "if ^Employee(company) exists, then execute the xxx subroutine. The $data function can be abbreviated to $d.

$data will actually return a number of different values.

- If data exists at the specified level of subscripting, and there are no sub-nodes, a value of 1 is returned
- If data exists at the specified level of subscripting, and there are sub-nodes, a value of 11 is returned
- If no data exists at the specified level of subscripting, but there are sub-nodes, a value of 10 is returned
- If no data and no subnodes exist at the specified level of subscripting, a value of 0 is returned.

In MUMPS, any non-zero value is evaluated as "true" when a logical operator such as *if* is used. Hence, the first three values returned by $data (ie 1, 10 and 11) will evaluate as true. The last situation (no data and no subnodes) will evaluate as false.

For example, consider the following global:

```
^test=3
^test(“a”)=1
^test(“a”,”b”,”c”)=2
^test(“a”,”b”,”d”)=3
```

```
$data(^test) = 11
$data(^test(“a”,”b”)=10
$data(^test(“a”,”b”,”c”)=1
$data(^test(“b”)=0
```

Preventing Undefined errors

If you try to retrieve a global node that doesn’t exist, MUMPS will produce a run-time error, eg <UNDEF>. To prevent this happening, you can either use the $data function to first test for the existence of the node, or, more conveniently, you can use the $get function. This will return the value of the global node if it exists, or a null value if it doesn’t exist. The $get function can be abbreviated to $g

So, based on the example we used in the previous section on $data:

```
$get(^test) = 3
$get(^test(“a”,”b”)=”” ; because no data exists at this level of subscripting
$get(^test(“a”,”b”,”c”)=1
$get(^test(“b”)=””
$get(^nonExistentGlobal)=””
```

Examining Globals

Caché and all other MUMPS systems include utilities for examining globals. The simplest, command line utility is ^%G which you'll find in Caché and some other MUMPS implementations. Run this from a terminal session:

USER> ***D ^%G***

Global ^ ***Employee***

Typing just the Global name will list the entire global. However, you can restrict the listing to specific subscripts, eg:

> ***Employee()***
>
> This will list all values for the first subscript only
>
> ***Employee("MGW"***
>
> This will list all Employee records with a first subscript value of MGW
>
> ***Employee("MGW",)***
>
> This will list all second subscripts for Employee records with a first subscript of "MGW"

Alternatively, Caché provides a graphical user interface tool called Caché Explorer that allows you to view and edit globals.

Conclusions

We've now covered the core basics of what globals are and how they can be created and manipulated. The next chapter will look at globals from the perspective of someone familiar with a relational database.

You've probably already realised that MUMPS globals impose very few controls or limitations over what you do. That's both a great thing – it allows you to very rapidly and flexibly design, implement and redesign your database structures – and a dangerous thing – in the wrong hands it can be a recipe for a completely uncontrolled mess. MUMPS leaves the discipline to you, the programmer. There are no safety nets, but on the other hand, there's almost no limit to what you can achieve or how you achieve it. You'll find that the efficiency of coding and execution is what really makes MUMPS an exciting and exhilarating environment to work in.

Once you try using MUMPS globals persistent storage, you'll probably begin to wonder why all databases couldn't work this way! Its simple, intuitive, flexible and the performance outstrips any relational database. Its also available for pretty much every platform, and will scale to enormous systems – some of the biggest interactive systems in the world run in MUMPS, some with tens of thousands of concurrent users.

However, if you think you need those controls and safety nets and the features that the relational world believe are essential "must-have" features, then MUMPS is definitely not for you. If you're still determined to go free-form mountain climbing, move on to the next chapter !

Chapter 2

RDBMS/SQL vs. MUMPS

This chapter will set out the salient differences between a standard Relational Database Management Systems (RDBMS) managed through SQL and the MUMPS data repository. Refer back to to Chapter 1 if you need to understand more about the basics of Globals and their manipulation.

Defining the data

Let us start with the basics – defining the data.

As an example, we shall use a simple database consisting of three tables:

1. A table of customers (CUSTOMER).
2. A table of orders (ORDER).
3. A table indicating the items making up an individual order (ITEM).

```
CUSTOMER  custNo  name  address  totalOrders
|
|
+------< ORDER  orderNo  custNo  orderDate  invoiceDate  totalValue
         |
         |
         +------< ITEM  orderNo  itemNo  price
```

Table names are shown in **bold**. The primary key of each table is shown underlined.

CUSTOMER

custNo	The Customer's (unique) number.
name	The Customer's name.
address	The Customer's address.
totalOrders	The number of orders placed by the customer to date.

ORDER

<u>orderNo</u>	The order number.
custNo	The relevant customer number (a foreign key from CUSTOMER).
orderDate	The Date of the Order.
invoiceDate	The Date of the Invoice.
totalValue	The Value of the order.

ITEM

<u>orderNo</u>	The order number (corresponding key from ORDER).
<u>itemNo</u>	The item (or part) number.
price	The price of the item to the customer (including any discount).

The one-to-many relationships are shown in the diagram. Each customer can have many orders and each order can be made up of any number of items (or parts).

The number of orders for a particular customer (CUSTOMER.totalOrders) is the total number of orders placed by the customer as identified by ORDER.orderNo. The value of an order (ORDER.totalValue) is the sum of the cost of each item in the order (as identified by ITEM.price).

CUSTOMER.totalOrders and ORDER.totalValue are not directly entered by a user of the application - they are derived fields.

For an RDBMS, these table definitions must be entered into the data definition part of the database (or schema) before SQL can be used to add, modify and retrieve records.

MUMPS does not enforce the use of a data-definition scheme and, as such, records can be written directly to the MUMPS data store without the need to formally defined the data. However, it is important to note that a relational schema can easily be layered on top of an MUMPS data repository for the purpose of accessing the MUMPS data via SQL-based reporting tools. A relational schema can be retrospectively added to an existing MUMPS data store provided the records are modeled to (approximately) third normal form.

The tables can be represented in MUMPS using the following globals:

CUSTOMER

```
^CUSTOMER(custNo)=name|address|totalOrders
```

ORDER

^ORDER(orderNo)=custNo|orderDate|invoiceDate|totalValue

ITEM

^ITEM(orderNo,itemNo)=price

The relationship between the CUSTOMER and ORDER tables could be represented by another global:

^ORDERX1(custNo,orderNo)=""

This would provide pointers to all Order Numbers for a specified Customer Number.

In MUMPS, its up to you what global names you use. Also you have a choice of either using different globals for each table (as we've shown above), or using the same global for some or all the tables and indices. For example, we could pack everything together using the following global structure:

```
^OrderDatabase("customer",custNo)= name_"~"_address_"~"_totalOrders
^OrderDatabase("order",orderNo)= custNo_"~"_orderDate_"~"_invoiceDate_"~"_totalValue
^OrderDatabase("item",orderNo,itemNo)=price
^OrderDatabase("index1",custNo,orderNo)=""
```

For the purposes of the examples that follow, we've chosen to use separate globals for each table.

We've also decided, arbitrarily, to use a tilde (~) character as the data delimiter in our example globals.

Adding new records to the Database

Let's start with a really simple example. Let's add a new customer record to the customer table.

SQL

```
INSERT INTO CUSTOMER (CustNo, Name, Address)
VALUES (100, 'Chris Munt', 'Oxford')
```

MUMPS

```
Set ^CUSTOMER(100)= "Chris Munt"_"~"_"Oxford"
```

We are using the tilde character (~) as the field delimiter. We can, of course, use any character we like for this purpose (including non-printable characters). For example, we could use:

```
Set ^CUSTOMER(100)= "Chris Munt"_$c(1)_"Oxford"
```

Read the $c(1) (aka $char) function as meaning "the character whose ASCII value is 1". This second alternative therefore uses ASCII 1 as a field delimiter.

Of course, in a real situation, the data values would be passed to the insert query (or MUMPS command) by means of a program as values of variables:

SQL

```
INSERT INTO CUSTOMER (custNo, :name, :address)
VALUES (:custNo, :name, :address)
```

MUMPS

```
Set ^CUSTOMER(custNo)=name_"~"_address
```

Retrieving records from the database

SQL

```
SELECT A.name, A.address
FROM CUSTOMER A
WHERE A.custNo = :custNo
```

MUMPS

```
Set record=$get(^CUSTOMER(custNo))
Set name=$piece(record,"~",1)
Set address=$piece(record,"~",1)
```

Note the use of the $get(). This is a convenient way of safely retrieving a global node. If the node doesn't actually exist, the $get() function returns a null (""). If you don't use the $get() function, ie:

```
Set record=^CUSTOMER(custNo)
```

Then if the specified global node doesn't exist, MUMPS will return a run-time <UNDEF> (ie undefined) error. Like most commands and functions in MUMPS, the $get() function can be abbreviated to just $g(), eg:

```
Set record=$g(^CUSTOMER(custNo))
```

Removing records from the database

SQL

```
DELETE FROM CUSTOMER A
WHERE A.custNo = :custNo
```

MUMPS

```
kill ^CUSTOMER(custNo)
```

Note that this simple example doesn't take referential integrity into account. You'll see later how we'll deal with this.

Parsing the database

SQL

```
SELECT A.custNo, A.name, A.address
FROM CUSTOMER A
```

MUMPS

```
s custNo="" f  s custNo=$order(^CUSTOMER(custNo)) Quit:custNo= "" do
      . Set record=$get(^CUSTOMER(custNo))
      . Set name=$piece(record,"~",1)
      . Set address=$piece(record,"~",2)
      . Set totalOrders=$piece(record,"~",3)
      . ; add code here to process the "selected row"
```

Note the use of the so-called "dot syntax". The lines preceded with a dot (".") represent a subroutine, called by the "do" command that terminated the first line. Typically you would do whatever you need to do with each "selected row" within the subroutine, as indicated by the comment (the last line that starts with the semi-colon (";").

The $order function in MUMPS is one of the keys to the power and flexibility of globals. Its functionality is not likely to be intuitive to someone familiar with RDBMS and SQL, so it's word reading the more detailed description in Chapter 1.

With $order and globals, you have total access to step through any of the keys in a table, starting and finishing on any value we wish. The important thing is to understand that Globals represent a hierarchical data storage structure. The key fields in the tables we're emulating are represented as the subscripts in a global, so we no longer have to access rows from a table in strict sequence: the $order function can be applied to each subscript (key) independently.

Using MUMPS functions to encapsulate database access

In practice, and in order to maximise code reuse, the MUMPS queries shown previously would usually be implemented as functions. Examples of such functions are shown below.

Adding new records to the database:

```
setCustomer(custNo,data) ;
      If custNo="" Quit 0
      Set ^CUSTOMER(custNo)=data("name")_"~"_data("address")
      Quit 1
```

This function could be now called repeatedly in the following way:

```
 kill data ; clear down data local array
 set data("name")="Rob Tweed"
 set data("address")="London"
 set custNo=101
 set ok=$$setCustomer(custNo,.data)
```

Note the period (".") in front of the *data* parameter. This is a "call by reference". *data* is a local array, not a simple scalar variable, so we must pass the name of the array by reference into the function.

The example above assumes that the setCustomer function is contained in the same MUMPS routine as our example run-time code. setCustomer might be held in a different routine (eg ^myFunctions), in which case our run-time code would look like the following:

```
 kill data ; clear down data local array
 set data("name")="Rob Tweed"
 set data("address")="London"
 set custNo=101
 set ok=$$setCustomer^myFunctions(custNo,.data)
```

The $$setCustomer() function is known as an *extrinsic function*. Think of extrinsic functions as *accessor methods* and you won't go far wrong. The $$setCustomer function returns 0 (ie false) if a null value of customer number is specified. Otherwise the record is saved and it returns 1 (ie true). You may want to test the value of the variable *ok* to check that the function worked.

Because we have a derived field (totalOrders) within the CUSTOMER table, the MUMPS code would more properly be:

```
setCustomer(custNo,data) ;
 if custNo="" Quit 0
 ; Derive the number of orders
 Set data("totalOrders")=0
 Set orderNo=""
 for  set orderNo=$order(^ORDERX1(custNo,orderNo)) Quit:orderNo=""  do
 . set data("totalOrders")=data("totalOrders")+1
 set ^CUSTOMER(custNo)=data("name")_"~"_data("address")_"~"_data("totalOrders")
 Quit 1
```

We will discuss these derived fields later in the 'triggers' section.

Retrieving records from the database:

The following extrinsic function will retrieve a row from the CUSTOMER table.

```
getCustomer(custNo,data) ;
      new record
      kill data ; clear down data array
      if custNo="" Quit 0
      set record=$get(^CUSTOMER(custNo))
      set data("name")=$piece(record,"~",1)
      set data("address")=$piece(record,"~",2)
      set data("totalOrders")=$piece(record,"~",3)
      Quit 1
```

This function can be used as follows:

```
 S custNo=101
 set ok=$$getCustomer(custNo,.data)
```

This will return the local array data (note the call by reference) containing the three data fields for the specified customer number:

```
data("name)
data("address")
data("totalOrders")
```

Removing records from the database:

The following extrinsic function will delete a row from the CUSTOMER table:

```
deleteCustomer(custNo) ;
      if custNo="" Quit 0
      kill ^CUSTOMER(custNo)
      Quit 1
```

This function can be used as follows:

```
 S custNo=101
 S ok=$$deleteCustomer(custNo)
```

Secondary Indices

In an RDBMS, secondary indices are maintained within the data-definition layer. Having defined them there, the contents of the index are automatically created and maintained by the RDBMS.

Indexes within MUMPS are maintained explicitly, for example in the table's update function. Because of the hierarchical nature of the MUMPS data repository, the primary record doubles as an index based on the primary key.

For example, consider the MUMPS function for adding a new record to the ORDER table:

```
setOrder(orderNo,data) ;
 new rec,itemNo,ok
 if orderNo="" Quit 0
 ; Calculate the value of the order
 set data("totalValue")=0
 set itemNo=""
 for set itemNo=$Order(^ITEM(orderNo,itemNo)) Quit:itemNo=""  do
 . set ok=$$getItem(orderNo,itemNo,.itemData)
 . Set data("totalValue")=data("totalValue")+itemData("price")
 set rec=data("custNo")_"~"_data("orderDate")_"~"_data("invoiceDate")
 set rec=rec_"~"_data("totalValue")
 set ^ORDER(orderNo)=rec
 Quit 1
```

Notice the code for calculating the value of the order. This parses all the ITEM records for the specified Order Number, and uses the $$getItem() function to reteieve the value for each item. The $$getItem() function would contain the following:

```
getItem(orderNo,itemNo,itemData)
 kill itemData
 s itemData(“price”)=0
 if orderNo=”” Quit 0
 if itemNo=”” Quit 0
 if ‘$data(^ITEM(orderNo,itemNo)) Quit 0
 set itemData(“price”)=^ITEM(orderNo,itemNo)
 Quit 1
```

Note the line in this function that checks for the existence of a global node for the specified Order Number and Item Number. This uses the MUMPS function $data, and also the *not* operator (“’”).

Now, suppose we wish to add the index to facilitate easy access to orders on a per-customer basis. We’re representing this by another global named ^ORDERX1. To do this we create an index on Customer Number (custNo) and Order Number (orderNo). This would be implemented by extending the setOrder function as follows:

```
setOrder(orderNo,data) ;
 new rec,itemNo,ok
 if orderNo="" Quit 0
 ; Calculate the value of the order
 set data(“totalValue”)=0
 set itemNo=””
 for set itemNo=$Order(^ITEM(orderNo,itemNo)) Quit:itemNo=””  do
 . set ok=$$getItem(orderNo,itemNo,.itemData)
 . Set data(“totalValue”)=data(“totalValue”)+itemData(“price”)
 set rec=data("custNo")_"~"_data("orderDate")_"~"_data("invoiceDate")
 set rec=rec_”~”_data(“totalValue”)
 set ^ORDER(orderNo)=rec
 if data(“custNo”)’=”” set ^ORDERX1(data(“custNo”),orderNo)=””
 Quit 1
```

We’d use this function to create an order as follows:

```
 set orderNo=21
 kill data
 set data(“custNo”)=101
 set data(“orderDate”)=”4/5/2003”
 set data(“invoiceDate”)=”4/7/2003”
 set ok=$$setOrder(orderNo,.data)
```

Referential Integrity

Referential Integrity is, perhaps, the best known of a general class of ‘referential actions’. Referential actions include all operations that involve changes to tables that must be made as a direct consequence of modifications to other tables.

The process of maintaining Referential Integrity is responsible for maintaining semantic integrity between related tables. Specifically, it is concerned with maintaining the natural joins or primary/foreign key correspondences between tables.

For example, when a customer number (CUSTOMER.custNo) is changed from one value to another (or removed), in order to maintain semantic integrity between the customer table and the order table, a corresponding change is indicated in the order table (ORDER.custNo). Similarly, when an order number (ORDER.orderNo) is changed from one value to another (or removed), in order to maintain semantic integrity between the order table and the item table, a corresponding change is indicated in the item table (ITEM.orderNo).

In an RDBMS, these integrity rules are implied by the information contained within the data definition layer (primary and foreign keys). In MUMPS, these rules can be implemented directly within our functions for maintaining the data store.

Taking the relationship between the customer and the order table, an update operation for CUSTOMER would be expressed as:

SQL

```
UPDATE CUSTOMER A
SET custNo = :newCustNo
WHERE A.custNo = :oldCustNo
```

This query will result in the corresponding records being updated in the ORDER table (according to the join CUSTOMER.custNo = ORDER.custNo)

MUMPS

```
updateCustomer(oldCustNo,newCustNo,newData) ;
 new result,orderData,orderNo
 if (oldCustNo="")!(newCustNo="") Quit 0
 set orderNo=""
 for  set orderNo=$order(^ORDERX1(oldCustNo,orderNo)) Quit:orderNo=""  do
  . set result=$$getOrder(orderNo,.orderData)
  . set orderData("custNo")=newCustNo
  . set ok=$$setOrder(orderNo,.orderData)
 set ok=$$setCustomer(newCustNo,.newData)
 if newCustNo'=oldCustNo set ok=$$deleteCustomer(oldCustNo)
 Quit 1
```

Notice that this function is largely built using functions that we have already created for the purpose of maintaining the customer and order table. It is easy to reuse code in MUMPS.

We'd need to create the $$getOrder function :

```
getOrder(orderNo,orderData) ;
 new record
 if (orderNo="") Quit 0
 set record=$g(^ORDER(orderNo))
 set orderData("custNo")=$piece(record,"~",1)
 set orderData("orderDate")=$piece(record,"~",2)
 set orderData("invoiceDate")=$piece(record,"~",3)
 set orderData("totalValue")=$piece(record,"~",4)
 Quit 1
```

We'll also need to extend our original, simple $$deleteCustomer() function. Similar considerations apply to operations that remove customer records from the database. The SQL query together with its equivalent M function is shown below.

SQL

```
DELETE FROM CUSTOMER A
WHERE A.custNo = :custNo
```

This query will result in the corresponding records being deleted from the ORDER table (according to the join CUSTOMER.custNo = ORDER.custNo and the integrity rules specified within the data-definition layer).

MUMPS

```
deleteCustomer(custNo) ;
 new orderNo
 if custNo="" Quit 0
 set orderNo=""
 for  set orderNo=$order(^ORDERX1(custNo,orderNo)) Quit:orderNo=""  do
 . set result=$$deleteOrder(custNo,orderNo)
 kill ^CUSTOMER(custNo)
 Quit 1
```

This requires the $$deleteOrder() function:

```
deleteOrder(custNo,orderNo) ;
 kill ^ITEM(orderNo) ; remove all order items
 kill ^ORDER(orderNo) ; remove the order
 kill ^ORDERX1(custNo,orderNo) ; remove the customer-order relationship
 Quit 1
```

Note that all item records in the ITEM table can be removed by applying the kill command at the orderNo hierarchy level.

This assumes that a 'cascading' relationship is required between the customer and order table. A 'breaking link' relationship would be implemented as follows:

```
deleteCustomer(custNo) ;
 new orederNo,result,orderData
 if custNo="" Quit 0
 set orderNo=""
 for  set orderNo=$order(^ORDERX1(custNo,orderNo)) Quit:orderNo=""  do
 . set result=$$getOrder(orderNo,.orderData)
 . set orderData("custNo")=””
 . set result=$$setOrder(orderNo,.orderData)
 kill ^CUSTOMER(custNo)
 Quit 1
```

Similar logic will apply with respect to data held in the ITEM table when an Order Number (ORDER.orderNo) is modified in, or deleted from the ORDER table.

Triggers

Triggers are simply a way of automatically invoking pre-defined code when certain conditions are met within the database. A trigger is usually fired as a result of records being updated in a certain way within the database.

In an RDBMS, triggers are defined within the data definition layer. In MUMPS we can add trigger code directly to the functions that we write for the purpose of maintaining the data store.

For example, consider the number of orders for a customer (CUSTOMER.totalOrders). We could define a trigger to automatically update this field as records are added (or removed) from the order table.

SQL

The following SQL would be included in the trigger code within ORDER for the purpose of creating CUSTOMER.totalOrders:

```
SELECT COUNT(A.orderNo)
FROM A.ORDER
WHERE A.custNo = :custNo
```

This query will be triggered for all constructive and destructive operations that are applied to the ORDER table (according to the join CUSTOMER.custNo = ORDER.custNo)

MUMPS

We simply add the following (trigger) code to the function for adding records to the order table:

```
setOrder(orderNo,data) ;
 new rec,itemNo,ok
 if orderNo="" Quit 0
 ; Calculate the value of the order
 set data(“totalValue”)=0
 set itemNo=””
 for set itemNo=$Order(^ITEM(orderNo,itemNo)) Quit:itemNo=””  do
 . set ok=$$getItem(orderNo,itemNo,.itemData)
 . Set data(“totalValue”)=data(“totalValue”)+itemData(“price”)
 set rec=data("custNo")_"~"_data("orderDate")_"~"_data("invoiceDate")
 set rec=rec_”~”_data(“totalValue”)
 set ^ORDER(orderNo)=rec
 if data(“custNo”)’=”” set ^ORDERX1(data(“custNo”),orderNo)=””
 ;
 ; Trigger the update of the CUSTOMER.totalOrders field
 new custData
 Set ok=$$getCustomer(data(“custNo”),.custData)
 Set ok=$$setCustomer(data(“CustNo”),.custData)
 ;
 Quit 1
```

The same considerations apply to operations that remove data from the ORDER table.

A similar scheme can be employed for the purpose of automatically updating the value of an order (ORDER.totalValue) as items are added to the order.

SQL

The following SQL would be included in the trigger code for the ITEM table for the purpose of generating ORDER.Value:

```
SELECT SUM(A.price)
FROM A.ITEM
WHERE A.orderNo = :orderNo
```

This query will be triggered for all constructive and destructive operations that are applied to the ITEM table (according to the join ORDER.orderNo = ITEM.orderNo)

MUMPS

We simply add the following (trigger) code to the function for adding records to the ITEM table:

```
setItem(orderNo,itemNo,data) ;
      new ok
      if (orderNo="")!(itemNo=“”) Quit 0
      set ^ITEM(orderNo,itemNo)=data("price")
      set^ORDERX1(custNo,orderNo)=“”
      ; Trigger the update of the ORDER.totalValue field
      set ok=$$getOrder(orderNo,.orderData)
      set ok=$$setOrder(orderNo,.orderData)
      Quit 1
```

The same considerations apply to operations that remove data from the ITEM table.

A further example of the use of triggers in our customer and orders database would be to automatically generate and raise an invoice for the customer as soon as an invoice date is added to the order table (ORDER.invoiceDate). We can very simply add this functionality to our procedure for updating the ORDER table:

```
setOrder(orderNo,data) ;
 new rec,itemNo,ok
 if orderNo="" Quit 0
 ; Calculate the value of the order
 set data(“totalValue”)=0
 set itemNo=“”
 for set itemNo=$Order(^ITEM(orderNo,itemNo)) Quit:itemNo=“”  do
 . set ok=$$getItem(orderNo,itemNo,.itemData)
 . Set data(“totalValue”)=data(“totalValue”)+itemData(“price”)
 set rec=data("custNo")_"~"_data("orderDate")_"~"_data("invoiceDate")
 set rec=rec_”~”_data(“totalValue”)
 set ^ORDER(orderNo)=rec
 if data(“custNo”)’=“” set ^ORDERX1(data(“custNo”),orderNo)=“”
 ;
 ; Trigger the update of the CUSTOMER.totalOrders field
 new custData
 Set ok=$$getCustomer(data(“custNo”),.custData)
 Set ok=$$setCustomer(data(“CustNo”),.custData)
 ;
 ; Raise an invoice if an invoice date is entered
 if Data(“invoiceDate”)’=“” Set Result=$$invoiceOrder(orderNo)
 ;
 Quit 1
```

Of course, the extrinsic function $$invoiceOrder() would have to be written to carry out the appropriate processing logic to raise the invoice.

Transaction Processing

A complete update to a database often consists of many individual updates to a number of tables. All contributing updates must be guaranteed to complete before the whole update (or *transaction*) can be said to be complete.

In an RDBMS, transaction control is usually active as a matter of default behaviour. In other words, changes to the database are not committed until the modifying process issues a ‘commit’ command, at which point the updates are committed to the database and a new transaction begins.

MUMPS is very similar in this respect with the exception that transaction control has to be explicitly turned-on. A program must issue a ‘transaction start’ command in addition to the ‘transaction’ commit command.

SQL

Commit a transaction and (implicitly) start a new transaction:

```
COMMIT
```

MUMPS

Start a new transaction:

```
TSTART
```

Commit a transaction:

```
TCOMMIT
```

If transaction control is *not* explicitly switched on in a MUMPS system, then all updates for that process will be immediately made to the live data store. In other words, each and every SET or KILL of a global node will be regarded as a complete transaction in its own right.

Conclusions

There are many key advantages to using MUMPS over a traditional RDBMS.

- Maintainability and core reuse. With discipline, an extremely high level of code reuse can be achieved. The examples we’ve described in this paper demonstrate how you can encapsulate all your database actions in terms of *get*, *set* and *delete* functions for each table – write these once and re-use them.

- Transportability. The definition of the data is contained within the functions that operate on it. There is no division between definition and implementation.

- Flexibility. The update code can be modified to trigger any action or event within the system.

- Performance and fine-tuning. Experienced MUMPS analysts will recognise opportunities for tuning the functions shown here for maintaining the data store. This is possible because the definition of the data and the implementation of the operations that are applied to it are held together within discrete functions. The analyst using SQL has no fine control over the way triggers, referential actions or transactions are managed.

- The advantages of using SQL to maintain or retrieve the data (use of third party reporting tools, for example) are not lost because a relational data definition can be transparently layered over the top of an existing set of MUMPS tables (globals). This can even be done retrospectively. For example, both Caché SQL and the third-party KB_SQL product (http://www.kbsystems.com/), implement a full SQL environment, layered on MUMPS globals.