Row & List library

ActiveX classes for managing in memory data sets

Developer's Guide

©2002, 2003, DIS - Développement Informatique Services
Francesco Foti
36, avenue Cardinal-Mermillod
1227 Carouge (GE)
Suisse (CH)

CMapStringToLong Class

Introduction

The CMapStringToLong class is responsible for associating a `Long` integer value (32 bits) to one (or many) string(s).
A `Long` value that was associated with a string can then be found using the string itself *or part of it*.

VB offers the powerful and built in collection class for grouping related items and accessing them by using a string (the key). Although the VB collection class is a good choice for solving this sort of problem, it lacks some useful functionalities and is a bit too restrictive by design (see Collections vs CMapStringToLong paragraph later).

vbAccelerator's article: "A fast index based collection"
(http://vbaccelerator.com/home/VB/Code/Techniques/A_Fast_Index-Based_Object_Collection/article.asp)

Definitions

Item	A signed long integer (32 bits) value, ie a value which data type is `Long`. Although a collection can store values of any data type, we restrict the items data type to `Long` , as this is the only data type supported by the CMapStringToLong class.
Key	A non empty alphanumeric string associated with an item. For a collection object, a key is unique. For a CMapStringToLong object, multiple items can have the same key.

Collections vs CMapStringToLong

The collection object misses some important facilities:

- Collections have a one way only, (hashed) element access; from a key or by element position, you can retrieve a value. You cannot retrieve the key associated with a collection item.
- The item keys in a collection are case insensitive. The key "kEY1" is equivalent to the key "Key1". If you try to add 2 items with those keys, VB generates a runtime error.
- You cannot have duplicate keys in a collection.
- You cannot search with a partial key to find the first item which key begins with the given partial key.
- You cannot remove all the items having the same value, without iterating the whole collection.

Faster than collections

Collections use an hashing algorithm for retrieving an item by key.
CMapStringToLong uses a dichotomic search in a sorted string array for retrieving an item by its key.

Good performance was not a requirement when I wrote this class. The purpose was not to beat the collection class, but rather to implement the features the collection is missing. However, I was amazed to see the simple dichotomic retrieval algorithm used by the class perform faster than the hashing algorithm used by the VB collection class.

How it works

String allocator

CMapStringToLong contains an internal string allocator. The allocator manages an array of strings, where the item keys are stored.

When a string is freed by the allocator, its array element index is placed in a garbage queue. The internal implementation of this garbage queue is a circular array.

String allocator member functions

Private Function AllocString(ByRef sNewString As String) As Long

Parameters
`sNewString` is the string that must be stored.

Description
AllocString inserts `sNewString` in the allocator's internal string array and returns a string handle. The string handle is a long integer that is the string array element index that can be used to later access the string.

The function calls `GarbQPop` to see if a previously removed array element can be reused. If the garbage queue is empty, the AllocString function handles the dynamic resizing of the internal string array if it needs to allocate a new array element for the sNewString.

Return value
The function returns a `Long` which is the string handle.

Note: The string handle corresponds to the internal string array element index used to store the string.

Private Sub FreeString(ByVal lHandle As Long)

Parameters
`lSlot` is a string handle previously returned by the `AllocString` function. Once freed, the string handle becomes invalid.

Note: String handles are reused. The same string handle value may be returned by a later call to the `AllocString` function, although it doesn't represent the same string.

Mapping strings to longs

Once passed to the string allocator, we've got a string handle for accessing the item key. The string handle is a `Long` value and the value that is associated to the key is also a `Long` value.
The `TMapItem` data type associates the string handle to the long value:

```
Private Type TMapItem
  lIndex      As Long
  lLongValue  As Long
End Type
```

The CMapStringToLong class maintains an array of `TMapItem` elements: `matMap()`.
The benefits of having string handles instead of normal strings in the `TMapItem` data structure now becomes obvious.
When a `TMapItem` element must be moved, the fast `CopyMemory()` API can be used.
The performance of internal class algorithms like `Quicksort`, which is used to sort the `matMap()` array, is greatly enhanced by using this string storage technique.

Note: The `matMap()` array grows by blocks of `GrowSize` elements, as the allocator's string array does.

Adding and finding items

To be able to find a key in the allocator's internal string array using a dichotomic search algorithm, the string array must be sorted. The only case where this internal array may become unsorted is when you add items using the `Add` method.

There are two ways to use the class for adding items:

1. The fastest way

This is the method I've used in the TestDriver application (that we'll soon discuss):

a.Set the Sorted property to False
b.Add the items
c.Set the Sorted property to True

2. The other way

d.Set the Sorted property to True
e.Add the items

With method 1, the internal array becomes unsorted when you add an item. You have to set the `Sorted` property back to `True`, which will trigger a sort procedure that uses a quicksort algorithm.
The problem with method 1 is that you cannot search for an item while you are adding items. This means that if you want to check for an item uniqueness while adding items, you'll have to use method 2.

With method2, the internal array remains sorted. The items you add are inserted into the internal array using an insertion sort algorithm. This is a far less efficient method, but it is the only one that allows you to call the `Find` methods of the class while adding items.

To be fair with the VB collection class, I have to admit that we have to use method 1 to beat the collection when inserting items. With method two, the CMapStringToLong class is very far from the performance of the collection class. The insertion sort algorithm simply cannot compete against the powerful hashing algorithm used by the VB collection class.

The TestDriver application

The TestDriver application is a VB project that has a reference set to the " DIS - CRow and CList smart objects library" (DISRowList.dll) ActiveX library.
It is important to use the compiled version of the ActiveX library (in which we find the CMapStringToLong class), for our tests to be significant.

Preparing the test bench and running the test

Be sure that the DISRowList.dll is registered on your system; if not, register it with the regsvr32.exe utility.
Then, simply open the TestDriver.vbp VB project. Go to the immediate pane (Alt+G) and type "Test1"+[Enter]. This will execute the test bench.

What's tested

Summary:

1. Loading a collection and a CMapStringToLong object with 100'000 elements, sequentially
2. Randomly find 1'000 items using a numeric index
3. Randomly find 1'000 items using an alphanumeric key
4. Randomly removing 1'000 items
5. Destroying the objects

TestDriver bench marks

Here are the results of three consecutive runs, obtained on Windows 2000, Pentium III 660 Mhz processor with 384Mb RAM.

First run

```
Collection: Adding 100000... 4.4744 seconds.
CMapStringToLong: Adding 100000... 2.8453 seconds.
```

```
Collection: Retrieving (by numerical index) 1000 elements... 8.7769 seconds.
CMapStringToLong: Retrieving (by numerical index) 1000 elements... 0.0006 seconds.
Collection: Retrieving (by key) 1000 elements... 0.0182 seconds.
CMapStringToLong: Retrieving (by key) 1000 elements... 0.0157 seconds.
Collection: Removing 1000 elements... 0.0211 seconds.
CMapStringToLong: Removing 1000 elements... 4.7084 seconds.
Collection: destroy... 0.2301 seconds.
CMapStringToLong: destroy... 0.1094 seconds.
```

Second run

```
Collection: Adding 100000... 4.4832 seconds.
CMapStringToLong: Adding 100000... 2.4849 seconds.
Collection: Retrieving (by numerical index) 1000 elements... 8.9604 seconds.
CMapStringToLong: Retrieving (by numerical index) 1000 elements... 0.0007 seconds.
Collection: Retrieving (by key) 1000 elements... 0.0184 seconds.
CMapStringToLong: Retrieving (by key) 1000 elements... 0.0161 seconds.
Collection: Removing 1000 elements... 0.0215 seconds.
CMapStringToLong: Removing 1000 elements... 4.7118 seconds.
Collection: destroy... 0.1938 seconds.
CMapStringToLong: destroy... 0.1024 seconds.
```

Third run

```
Collection: Adding 100000... 4.5272 seconds.
CMapStringToLong: Adding 100000... 2.4465 seconds.
Collection: Retrieving (by numerical index) 1000 elements... 8.9137 seconds.
CMapStringToLong: Retrieving (by numerical index) 1000 elements... 0.0006 seconds.
Collection: Retrieving (by key) 1000 elements... 0.0182 seconds.
CMapStringToLong: Retrieving (by key) 1000 elements... 0.0161 seconds.
Collection: Removing 1000 elements... 0.0372 seconds.
CMapStringToLong: Removing 1000 elements... 4.7146 seconds.
Collection: destroy... 0.2017 seconds.
CMapStringToLong: destroy... 0.1133 seconds.
```

Score table

The score array summarizes the averages of the benchmarks obtained by the three consecutive runs of our bench. You see in bold the best performance numbers.

	Add	Retrieve by index	Retrieve by key	Remove	Destroy
Collection	4.4949	8.8836	0.0182	**0.0266**	0.2085
CMapStringToLong	**2.5922**	**0.0006**	**0.0159**	4.7116	**0.1083**

Interpreting the results

Where the collection class wins

The collection class is faster than the CMapStringToLong class when it has to remove an element.
The only explanation I can imagine is that the collection class stores its items in a sort of linked list. Linked lists are very fast when removing elements. Also, when removing an element, CMapStringToLong has to move all the `TMapItem` elements of its internal array that are below the removed element
This is where I've had a problem with my implementation of the `Remove` method. I could not figure out a way to use the `CopyMemory()` API without crashing VB, so I've used a `For` loop to move the elements. This slows down the performance of the removal process and it remains quite far from the collection's performance.

Where the collection class poorly performs

As you can see, the performance of the collection class dramatically decreases when you access its items using a numeric index. This seems to confirm the idea that the collection stores its items in a sort of linked list. It's like when the collection

class accesses an item by a numerical index, it does something like an iteration from the first element to the one that is accessed.
CMapStringToLong instead, does an almost direct access to an array element, to get the corresponding item.

Hashing vs dichotomic search

The third column of the score table indicates that the dichotomic algorithm used by the CMapStringToLong class beats the hashing algorithm used by the collection class, although the difference is quite small.

Destruction of class instances

I just can't imagine why, but a CMapStringToLong object instance destruction is faster than a collection object instance destruction. What I can comment is that when a CMapStringToLong object instance has to be destroyed, it is just a matter of destroying arrays. The collection class may have to release internal linked list pointers instead, thus being a little bit slower.

CMapStringToLong Class Reference

Public Sub Clear()

Remove all the items in the set.

Public Property Get GrowSize() As Long

Public Property Let GrowSize(ByVal lGrowSize As Long)

Controls the number of elements added to the internal arrays when they need to be dynamically expanded.
Default value: `DEFAULT_GROWSIZE` which is 20.

Public Property Get Count() As Long

Returns the number of items in the set.

Public Property Get Key(ByVal lIndex As Long) As String

Public Property Let Key(ByVal lIndex As Long, ByRef sNewValue As String)

The `Key` property returns or changes the key associated to an item of the set. The item's position in the set `[1..Count]` must be given.
The `Key` property for an item can only be changed when the `Sorted` property is `False`.

Public Property Get Item(ByVal lIndex As Long) As Long

Public Property Let Item (ByVal lIndex As Long, ByVal lNewLong As Long)

The `Item` property returns or changes the key associated to an item of the set. The item's position in the set `[1..Count]` must be given.

Public Property Get Sorted() As Boolean

Public Property Let Sorted(ByVal fSorted As Boolean)

The ability to control the internal sorted state of the key array is exposed to allow maximum performance when adding or updating many items or item keys at once. When the `Sorted` property is `False`, the `Find`, `FindFirst` and `RemoveDuplicates` methods cannot be used. On the contrary, when the `Sorted` property is `True`, the `CaseSensitive` property cannot be changed.
Setting the property to `True` sorts the internal array of keys using a quicksort algorithm.

Public Property Get CaseSensitive() As Boolean

Public Property Let CaseSensitive(ByVal fCaseSensitive As Boolean)

The `CaseSensitive` property controls the behavior of the class regarding letter case for item keys.
The property cannot be changed when the `Sorted` property is `True`.

Public Sub Add(ByRef sKey As String, ByVal lItem As Long)

Parameters
`sKey` is the item's key and it can be an empty string. As duplicate keys are allowed, there may be multiple items associated with the same key value or an empty key.
`lItem` is the `Long` value to store in the set.

See the `Test1` or `Test3` procedure in the TestDriver application for an example.

Public Function Find(ByRef sSearch As String) As Long

Parameters
`sSearch` is the key to search. The search is case sensitive according to the `CaseSensitive` property.

Return value
The index of the first item found which key is `sKey` is returned. If the key is not found, the function returns 0 (zero).

Note
When a set contains items with duplicate keys, the item found by the `Find` function is one of the items that share the same key. However, it is hardly predictable which one of these items will be found; you shall instead use the `FindFirst` function to find the first one. Then you can loop on the following items in sequential order, until the key of the addressed item changes.

See the `Test3` procedure in the TestDriver application for an example.

Public Function FindFirst(ByRef sSearch As String, Optional ByVal fRootSearch As Boolean = False) As Long

Parameters
`sSearch` is the key to search. The search is case sensitive according to the `CaseSensitive` property.
`fRootSearch` is an optional flag (which defaults to `False`). If `True`, then `sSearch` is considered to be a key root for which to search for.

Return value
The index of the first item which key is `sKey` is returned. If the key is not found, the function returns 0 (zero).

See the `Test3` procedure in the TestDriver application for an example.

Public Sub Remove(ByVal lIndex As Long)

Remove the item in the set at position `lIndex`.

Public Sub RemoveMappingsFor(ByVal lLongValue As Long)

Removes all the items in the set that have the specified `lLongValue` value, regardless of their key value.

Public Sub RemoveDuplicates()

For all groups of items that share the same key, only one among them is kept in the set; all the others are removed from the set.

Note

2. The Sorted property must be True before calling this method, otherwise the class raises an error.
3. For each group of items sharing the same key, only the item found by the FindFirst function is kept. Thus, it is not always obvious which one of the items of a group is kept.

See the `Test3` procedure in the TestDriver application for an example.

CRow class

A CRow object is an ordered set of columns. Columns may be named, but multiple unnamed columns can also exist in a row.
We'll call a CRow object instance a row.

A column has the following properties:

- name
- data type*
- data size*
- flags*
- Value

The properties marked with an asterisk (*) are under the class user control. The CRow class doesn't make any use of these properties, except storing them and offering accessors member functions (ColType(), ColSize() and ColFlags()).

There are many ways to define the set of columns managed by the CRow class. These different ways of defining the columns have been implemented for rapid application development and provide a great level of flexibility.

The CRow class is built to be used in tight integration with the CList class. Defining the columns of a CRow object is very similar to defining the columns of a CList object. Both classes expose methods to define the column set for an object of the other class, thus reducing the need to repeat code and still adding flexibility.

Defining the column set

Previewing the final result

Let's choose an example of what we want to achieve. We'll then see the different ways to get the job done.
Here is an illustration of the columns (and their properties) that we want to define:

	ClientID	Name	Address	Zip	City	State
Value	1468	John Doe	47 Main Street	12345	Geneva	Switzerland
Data type	vbLong	vbString	vbString	vbString	vbString	vbString
Data size	8	0	0	0	0	0
Flags	1	0	0	0	0	0

This is an example, and just to use some flags, I assume that as a convention, we consider that a flag value of 1 indicates a database key field and that a data size of 0 for the `vbString` data type indicates an unlimited size string.

All the examples we'll see are taken from the TestDriver application.

Defining and populating the row

Method 1: Populate the column set of a CRow object using the AddCol method

The `AddCol` method of the CRow class provides the most flexible way for populating the column set. This is the only method that allows to specify a reference column when adding a new column. A new column can then be added before or after the reference column.
To get the ordinal position of a column in the column set by its name, we use the `ColPos` member function.

<u>Example</u>:

```
Sub Row1()
  Dim oRow          As New CRow
  Dim iColPos       As Long
```

```
  'Define using the AddCol method
  With oRow
    .AddCol "ClientID", 1468&, 8&, 1&
    .AddCol "Name", "John Doe", 0&, 0&
    .AddCol "Address", "47 Main Street", 0&, 0&
    .AddCol "City", "Geneva", 0&, 0&
    .AddCol "State", "Switzerland", 0&, 0&
    'insert the Zip column after the Address column
    iColPos = .ColPos("Address")
    .AddCol "Zip", "12345", 0&, 0&, plInsertAfter:=iColPos
  End With
  RowDump oRow, "AddCol() method"
End Sub
```

Method 2: Using "on the fly" arrays with the VB Array() method

This method is less flexible than method 1 because columns cannot be positioned relatively.
Also, a second class method has to be used to assign the column values.

Example:

```
'Defining column set Using "on the fly" arrays with the VB Array() method
Sub Row2()
  Dim oRow          As New CRow
  oRow.ArrayDefine Array("ClientID", "Name", "Address", "Zip", "City", "State"), _
                   Array(vbLong, vbString, vbString, vbString,vbString,vbString), _
                   Array(8&, 0&, 0&, 0&, 0&, 0&), _
                   Array(1&, 0&, 0&, 0&, 0&, 0&)
  oRow.Assign 1468&, "John Doe", "47 Main Street", "12345", "Geneva", "Switzerland"
  RowDump oRow, "ArrayDefine() method"
End Sub
```

Of course, the result of `Row2` is similar to the result of `Row1`:

Immediate

```
Row1
---------------+
AddCol() method|
--------+----+-------+---+----+-----+
ClientID|Name|Address|Zip|City|State|
--------+----+-------+---+----+-----+
1468    |John|47 M...|123|Gene|Switz|

Row2
--------------------+
ArrayDefine() method|
--------+----+-------+---+----+-----+
ClientID|Name|Address|Zip|City|State|
--------+----+-------+---+----+-----+
1468    |John|47 M...|123|Gene|Switz|
```

Method 3: The "one line definition" method

This method is the fastest and most compact way of defining the column set, but you have to take extreme care not to forget a parameter. We use the `Define` member method of the CRow class, where you can queue the column name, type, size and flags, for each column. We still have to assign the column values with a second method call.

Example:

```
'Defining column set Using the Define method
Sub Row3()
  Dim oRow          As New CRow
```

```
  'name, type, size, flags
  oRow.Define "ClientID", vbLong, 8&, 1&, _
              "Name", vbString, 0&, 0&, _
              "Address", vbString, 0&, 0&, _
              "Zip", vbString, 0&, 0&, _
              "City", vbString, 0&, 0&, _
              "State", vbString, 0&, 0&
  oRow.Assign 1468&, "John Doe", "47 Main Street", "12345", "Geneva", "Switzerland"
  RowDump oRow, "Define() method"
End Sub
```

Populating row data using the ArrayAssign method

In our last two examples, the row data has been inserted using the `Assign` method. Using the `ArrayAssign` method is just a slight variation where we use again the VB `Array`() method. We could have used an array created by any other means, somewhere else in our code.

Example:

```
'Populate column set Using the ArrayAssign method
Sub Row4()
  Dim oRow As New CRow
  'name, type, size, flags
  oRow.Define "ClientID", vbLong, 8&, 1&, _
              "Name", vbString, 0&, 0&, _
              "Address", vbString, 0&, 0&, _
              "Zip", vbString, 0&, 0&, _
              "City", vbString, 0&, 0&, _
              "State", vbString, 0&, 0&
  oRow.ArrayAssign Array(1468&, "John Doe", "47 Main Street", _
                         "12345", "Geneva", "Switzerland")
  RowDump oRow, "Define() method"
End Sub
```

Merging rows

When you have two row objects with different or overlapping columns, you may want to merge them in a new row containing the merged column set. The Merge method merges the columns of the row parameter with the columns of the row on which the method is invoked (destination row). If both rows have the same column, the properties of the source row are ignored; the properties of the column already defined in the destination row have precedence over the properties of the equivalent column in the destination row. Note however, that *only named columns are merged*.

Example:

```
'Merging rows
Sub Row6()
  Dim oRow1     As New CRow
  Dim oRow2     As New CRow

  oRow1.Define "ClientID", vbLong, 8&, 1&, _
               "Name", vbString, 0&, 0&, _
               "Address", vbString, 0&, 0&
  oRow1.Assign 1468&, "John Doe", "47 Main Street"

  oRow2.Define "Zip", vbString, 0&, 0&, _
               "Name", vbString, 0&, 0&, _
               "City", vbString, 0&, 0&, _
               "State", vbString, 0&, 0&
  oRow2.Assign "12345", "Patrick Doe", "Geneva", "Switzerland"

  'Merge Row2 into Row1
  'The name column of row1 will be kept.
  oRow1.Merge oRow2
  RowDump oRow1, "oRow1 merged with oRow2"
End Sub
```

Accessing column properties

Column properties each have corresponding property procedure for accessing their values (`Col`*property*()).
For all column properties methods, you can specify either a numeric index or the column name for the column parameter (`pvIndex`); except for the `ColName` property.

Accessing unnamed columns

Unnamed columns can be accessed by their positional index (which is a number), but also by using a special column name that is the the "#" character followed by the column's index (let's call it the "# notation"). For methods that only accept column names, like `Sort` or `FindFirst`, you can use the # notation to specify unnamed columns. The drawback of the # notation, is that you can't assign a name beginning with "#" to a named column.

Note: For every method or property that accepts a column name, you should care about the letter case of the column name if the `ColCaseSensitive` property is `True`.

Removing columns

You can remove a column specifying either its position `[1..ColCount]` or its name, with the `RemoveCol` method. The `Clear` method removes all columns at once.

Copying and cloning rows

To *copy* a row, you *create* a CRow object and use the CopyFrom method.
To *clone* a row, you *declare* a CRow object variable and call the Clone method of another valid instance of a CRow object variable; the CRow class handles the creation of the new instance and returns a reference on it.

Example:
```
'Copying and cloning rows
Sub Row5()
  Dim oRow1      As New CRow
  Dim oRow2      As New CRow
  Dim oRowClone As CRow

  oRow1.Define "ClientID", vbLong, 8&, 1&, _
               "Name", vbString, 0&, 0&, _
               "Address", vbString, 0&, 0&, _
               "Zip", vbString, 0&, 0&, _
               "City", vbString, 0&, 0&, _
               "State", vbString, 0&, 0&
  oRow1.Assign 1468&, "John Doe", "47 Main Street", "12345", _
               "Geneva", "Switzerland"

  'Copy row
  oRow2.CopyFrom oRow1
  RowDump oRow2, "oRow2 copied from oRow1"

  'Clone row
  Set oRowClone = oRow1.Clone()
  RowDump oRowClone, "oRowClone created from oRow1"
End Sub
```

The `DefineRow` method is *almost* the equivalent of the `CopyFrom` method, but works as its inverse. `CopyFrom` copies a source row (the method parameter) into the row on which the method is called, and it also copies the column values. However (and this is important), `DefineRow` doesn't copy the values, but just the column set definition. With both methods, previously existing columns are destroyed, before redefining the destination column set.

CRow and CList class coupling

The `DefineList` method is the equivalent of the `DefineRow` method, but it defines the columns for a CList target object, applying the column set definition of the row to the list.
The CList class also has a `DefineRow` method, which defines the column set of a row by replacing it by the column set of a list.

CRow Class Reference

Most of the methods of the class have been discussed earlier, so I just present here the method signatures.

```
Public Sub Clear()
```

Defining the column set

```
Public Property Get ColCount() As Long
Public Property Get ColCaseSensitive() As Boolean
Public Property Let ColCaseSensitive(ByVal fColCaseSensitive As Boolean)
Public Sub Define(ParamArray pavDefs() As Variant)
Public Sub DefineRow(ByRef poDestRow as CRow)
Public Sub DefineList(ByRef poDestList as CList)
Public Sub ArrayDefine(pavColName As Variant, _
                       Optional pavDataType As Variant, _
                       Optional pavDataSize As Variant, _
                       Optional pavDataFlags As Variant)
Public Sub AddCol(ByRef psColName As String, _
                  ByVal pvColValue As Variant, _
                  ByVal plDataSize As Long, _
                  ByVal plFlags As Long, _
                  Optional ByVal plInsertAfter As Long = 0&, _
                  Optional ByVal plInsertBefore As Long = 0&)
Public Sub RemoveCol(ByVal pvColIndex As Variant)
```

Accessing column properties

```
Public Function ColPos(ByVal psColName As String) As Long
Public Function ColExists(ByVal psColName As String) As Boolean
Public Property Get ColValue(ByVal pvIndex As Variant) As Variant
Public Property Let ColValue(ByVal pvIndex As Variant, ByVal pvNewValue As Variant)
Public Property Get ColName(ByVal plColIndex As Long) As String
Public Property Let ColName(ByVal plColIndex As Long, ByVal psNewName As String)
Public Property Get ColType(ByVal pvIndex As Variant) As Integer
Public Property Let ColType(ByVal pvIndex As Variant, ByVal piNewType As Integer)
Public Property Get ColSize(ByVal pvIndex As Variant) As Long
Public Property Let ColSize(ByVal pvIndex As Variant, ByVal plNewSize As Long)
Public Property Get ColFlags(ByVal pvIndex As Variant) As Long
Public Property Let ColFlags(ByVal pvIndex As Variant, ByVal plNewFlags As Long)
```

Assigning column values from arrays

```
Public Sub Assign(ParamArray pavValues() As Variant)
Public Sub ArrayAssign(ByRef pavValues As Variant)
```

Copying and cloning a Crow object

```
Public Function Clone() As CRow
Public Sub CopyFrom(ByRef poRowSource As CRow)
```

CList Class

A list is a two dimensional array. The columns of the two dimensional array are defined by a set of column definitions added to the list. We'll call a cell the memory location defined by a row and column intersection, in other words, an array element of the two dimensional array is a cell. We'll call a CList object instance a list.

Defining list columns

You define the columns of a CList object as you do for a CRow object; CList has the same member functions for defining its column set, but there are some subtle differences. First of all, the columns of a CList object have the same properties as the columns of a CRow object, except for the value property. This should be obvious, as a list is a set of rows and not a single row.

The `AddCol` method of the CList class closely resembles the `AddCol` method of the CRow class, but the second parameter of the method is called a "template value". A list is a collection of rows, so there is no sense to provide a column value when defining a new list column. Instead, the `AddCol` method of CList, uses the "template value" to automatically determine a data type for the column (internally using the VB `VarType` function). Note however that the method leaves it to you to provide the data size parameter.

The `ColCount` method returns the number of columns in the column set defining the list. This is the same property name as the CRow class. Looking at the CList class reference, you'll find a `Count` method. The `Count` method returns the number of *rows* in the list. Now it should be clear that the column count property was called `ColCount` in the CRow class (if you ever happened to ask yourself the question), in order to avoid confusion.

Copying a list or a row definition

You can copy the definition of an existing list with the `DefineList` method. You can also copy a row definition with the `DefineList` member method of the CRow class, which was discussed earlier.

Additional notes

There is no `RemoveCol` method for the CList class. This is a class design limitation, because the CList class stores list data in an internal, bi dimensional array: the `DataArray`.

Although the columns have a defined data type, when adding or assign row column values, there is no data type checking done by the class. This means that the class methods will not complain if you assign a string to a list cell which column has been defined as `Long` data type.

Adding, updating and accessing list data

Accessing cell data

The Item property of the CList class is the brother of the ColValue property of the CRow class. Both are used to access the values of the data stored inside object instances. The difference between the two method lies in the method parameters. The Item property of CList has a second parameter, used to specify the row number of the cell, which must be in the range `[1..Count].`

Removing data

Removing data is easy. The Clear method removes all the list rows and the column definitions. To remove the data rows and keep the column definitions, use the Reset method. To remove a single row of data, use the Remove method.

When a row index has to be specified for a method, it is always in the `[1..Count]` range (ie, its one based).

Adding data

There are three methods to add a row of data to a list.
See the `List1` method in the TestDriver application for examples.

Method 1: using a CRow object

You add a row of data to a CList object using the `AddRow` method.
The `AddRow` method is the most flexible of the add methods, because it copies only the row column values which are also present in the list. The matching is done on the column names. Take extreme care with the `CaseSensitive` property of the row, which should match the `ColCaseSensitive` property of the list.
If a list column is not found in the row object that is added, the corresponding list cell will be assigned a `Null` value.

Method 2: passing an array of data, using the AddValuesArray method
AND
Method 3: appending a row of data with the AddValues method

These two methods are almost the same. In one case (`AddValuesArray`), you pass an array of values (for example, you can use the VB `Array`() function), in the other case (`AddValues`), you list the values you want to add, as the method parameters.
For both methods you have to specify the cell values, in the order of the column set.

Copying lists

You can copy entire lists with the CopyFrom method. However you should care that copying entire lists is a slow process.

Updating data

As for adding data, there are three methods for updating list data, by row: `AssignRow`, `AssignValues` and `AssignValuesArray`. Their behavior is similar to the `Add` methods, except that they take an extra `RowIndex` parameter.

Sorting list data

Amazingly, the `Sort` method is used to sort the list. The current implementation of the sorting algorithm only supports sorting on a single column. Anyway, the format of the sSortColumns parameter is designed to support the specification of multiple columns. Future implementations may support the use of multiple sort columns without changing the method signature.

Sort columns string format: `[!]Column name[+|-][, [!]ColumnName2[+|-] [, ...] ]`

The optional “!” character (*bang*), preceding the column name, forces a case sensitive string comparison when sorting the column data, if the column data type is `vbString`. If the bang is omitted then the string comparisons in the sort algorithm will be case insensitive (ie `vbTextCompare`).

Once a list is sorted, its `Sorted` property will return `True`, until data belonging to one of the sort columns is modified.

Partial sort

The sort method provides two optional parameters, `lStartRow` and `lEndRow`, that allow to specify a contiguous range of rows to be sorted. To sort a range of rows, `lStartRow` must be specified. If `lEndRow` is omitted, the range is supposed to be `[lStartRow..`*`list`*`.Count]`.
If the range of rows being sorted doesn't include all the list rows, then the `Sorted` property will be set to `False`. This a rather important behavior, as we'll see that when the `Sorted` property is `False`, using the `Find` methods results in sequential searches, instead of more performant dichotomic searches.

Special column flags

Column flags are Long integer values, but the CList class reserves the high 16 bits for its own uses. When you read a column's flags, you'll get the full 32 bits value, but you can't change the highest 16 bits.
There are two bit values for each column's flags maintained by the class:

```
Const klColFlagSortedCaseSensitive     As Long = &H10000
Const klColFlagSortedCaseInsensitive   As Long = &H20000
```

Finding list data

You can always use the `Find` or `FindFirst` methods to search for any data in a list, whether the list is sorted or not.
In the current implementation of the Clist class, you can sort the list on one column. Similarly, you can search for data specifying a value to search for one column at a time. When you search for data in the column on which the list is sorted, the class uses a fast dichotomic search algorithm, while the `Sorted` property is `True`. If you search for data that is not in the sorted column or when the `Sorted` property if `False`, the class uses a dumb sequential search algorithm, far less performant than the dichotomic algorithm.

Search criteria

When you search for data and when you want to sort the list, the data type you specified when defining the list plays an important role.
If you search for data on a column which data type is not `vbString`, the only option you have is to search for an exact match; you specify the value to search as the criteria and the CList class will look for an exact match.

When the column on which you search has a `vbString` data type (*`list`*`.ColType(`*`column index`*`) = vbString`), then you have two additional possibilities to search for data:

1. Specifying the root of the string to search, appending the root expansion operator, the “*” character (*wildcard*) at the end of the search criteria.

   Example: `oList.Find(“ProductName”, “gnocchi*”)`

2. Search for a pattern match. A search pattern can include a leading or trailing wilcard, providing root or suffix string search and it can include one or more jokers (the “?” character), which will match any other character.

   Example: `oList.Find(“ProductName”, “*gno??hi*”)`

If you search on the column on which the list is sorted, then the case sensitivity used by the search algorithm will be the same that was specified when sorting the column, regardless of an eventual presence of a bang operator in the search criteria.
If the list isn't sorted or you search on another column than the one on which the list is sorted, then you'll have to bang the column name passed to the `Find` functions, if you want to specify a case sensitive search.

Notes:

- You can restrict a search on a partial, contiguous range of rows, by using the optional parameters of the `Find` functions.

- Each of the `Find` functions return 0 if the search was unsuccessful or the row index for the first found row.

Removing duplicates

When a list is sorted you can use the `RemoveDuplicates` method to automatically remove rows having the same column value (on which the list is sorted).
Only the first row of a group of rows having the same key value is retained.

The DataArray

The `DataArray` is the list memory. It is a member variable of the CList class, which has a module level scope. The data type of the `DataArray` variable is a `Variant`, which subtype is `vbArray`. In other words, its a two dimensional array of `Variant`s, contained in a `Variant` variable.
The `DataArray` instance variable of a list object is automatically exposed to the outside of the class, because it is declared as a `Public` variable. It is not encapsulated inside `Property Get/Let` methods.

Now you should be screaming, very loud, or at least be deeply shocked by what you just read.
The few preceding lines sound like an OOP heresy, doesn't they ?

Why ?

Exposing the DataArray this way, completely breaks the class encapsulation and would certainly be the last and baddest thing to do when designing a class.
This exposure puts every CList instance variable in great danger, when left in irresponsible coding hands.
But there is a pretty good reason I had to do that, and the danger is the price to pay.

The reason is the `GetRows` method of the ADO recordset class.

Because `GetRows` is a powerful and fast method to load a recordset (or part of it) in memory, I've decided to keep that power and let the CList class benefit from it. The result of the `GetRows` method has to be assigned to a `Variant` variable.
Think about it and you'll certainly conclude, like me, that this is the only performant and reasonable method to quickly morph a recordset into a CList object; other techniques would have to somehow copy the `Variant` variable returned by GetRows into a list, and that would be far less efficient.

Take a look at this sample code:

```
' Quickly get the snapshot of an SQL query in a CList object.
Public Function ADOGetSnapshotList(oConn As ADODB.Connection, ByRef SQL As String, ByRef
oList As CList) As Boolean
  Dim rsSnap          As New ADODB.Recordset
  Dim lErr            As Long
  Dim sErr            As String
  Dim vErr            As Variant

  On Error GoTo ADOGetSnapshotList_Err
  ClearErr

  rsSnap.Open SQL, oConn, adOpenStatic, adLockReadOnly, adCmdText
  ADODefineListFromSet oList, rsSnap
  If Not rsSnap.EOF Then
    oList.DataArray = rsSnap.GetRows()
    oList.SyncWithDataArray
  End If
  rsSnap.Close

  ADOGetSnapshotList = True
  Exit Function

ADOGetSnapshotList_Err:
  If Not oConn Is Nothing Then
```

```
    If oConn.Errors.Count Then
      sErr = ""
      lErr = Err.Number Xor vbObjectError
      For Each vErr In oConn.Errors
        If Len(sErr) Then sErr = sErr & vbCrLf
        sErr = sErr & (vErr.Number Xor vbObjectError) & ": " & vErr.Description
      Next
      SetErr lErr, sErr
    Else
      SetErr Err.Number, Err.Description
    End If
  Else
    SetErr Err.Number, Err.Description
  End If
End Function
```

I let you figure out what the `ADODefineListFromSet` sub does, but you can help yourself by looking at the source code of the MADOScript.bas code module, which encapsulates an easy to use and flexible API for using ADO with the classes of this library. You can find this module in the DataTestDriver project joined to the library source code.

The call to the `SyncWithDataArray` is needed for the CList class to organize its internal data structures around the newly assigned `DataArray` value. This is the only valid use of the `SyncWithDataArray` method, ie after having assigned the `DataArray` value.

CList Class Reference

Most of the methods of the class have been discussed earlier, so I just present here the method signatures.

```
Public Sub Clear()
Public Sub Reset()
```

Defining and accessing columns informations

```
Public Property Get ColCount() As Long
Public Property Get ColCaseSensitive() As Boolean
Public Property Let ColCaseSensitive(ByVal fColCaseSensitive As Boolean)
Public Sub AddCol(ByRef sColName As String, _
                  ByVal vTemplateValue As Variant, _
                  ByVal lDataSize As Long, _
                  ByVal lFlags As Long, _
                  Optional ByVal lInsertAfter As Long = 0&, _
                  Optional ByVal lInsertBefore As Long = 0&)
Public Function ColPos(ByVal sColName As String) As Long
Public Function ColExists(ByVal psColName As String) As Boolean
Public Property Get ColName(ByVal lColIndex As Long) As String
Public Property Let ColName(ByVal lColIndex As Long, ByVal sNewName As String)
Public Property Get ColType(ByVal vIndex As Variant) As Integer
Public Property Let ColType(ByVal vIndex As Variant, ByVal iNewType As Integer)
Public Property Get ColSize(ByVal vIndex As Variant) As Long
Public Property Let ColSize(ByVal vIndex As Variant, ByVal lNewSize As Long)
Public Property Get ColFlags(ByVal vIndex As Variant) As Long
Public Property Let ColFlags(ByVal vIndex As Variant, ByVal lNewFlags As Long)
```

Defining the column set

```
Public Sub Define(ParamArray pavDefs() As Variant)
Public Sub ArrayDefine(avColName As Variant, _
                       Optional avDataType As Variant, _
                       Optional avDataSize As Variant, _
                       Optional avDataFlags As Variant)
Public Sub DefineList(poRetList As CList)
```

Adding Data

```
Public Property Get Count() As Long
Public Function AddRow(oRow As CRow, _
                       Optional ByVal lInsertAfter As Long = 0&, _
```

```
                       Optional ByVal lInsertBefore As Long = 0&) As Long
Public Function AddValues(ParamArray avValues() As Variant) As Long
Public Function AddValuesArray(avValues As Variant, _
                               Optional ByVal lInsertAfter As Long = 0&, _
                               Optional ByVal lInsertBefore As Long = 0&) As Long
Public Sub CopyFrom(ByRef poListSource As CList)
```

Updating list data

```
Public Sub AssignRow(ByVal lRowIndex As Long, ByRef oSourceRow As Crow)
Public Sub AssignValues(ByVal lRowIndex As Long, ParamArray avValues() As Variant)
Public Sub AssignValuesArray(ByVal lRowIndex As Long, ByRef avValues() As Variant)
```

Accessing list data thru CRow objects

```
Public Property Get Row(ByVal lRowIndex As Long) As CRow
Public Sub GetRow(oFillRow As CRow, ByVal lRowIndex As Long)
Public Sub DefineRow(poRetRow As CRow)
Public Property Let Row(ByVal lRowIndex As Long, ByRef oRow As CRow)
Public Sub Remove(ByVal lIndex As Long)
```

Controlling the DataArray

```
Public Property Get GrowSize() As Long
Public Property Let GrowSize(ByVal lGrowSize As Long)
```

Accessing the DataArray outside the class

```
Public Sub SyncWithDataArray()
```

Sorting the list

```
Public Property Get Sorted() As Boolean
Public Sub Sort(ByVal sSortColumns As String, Optional ByVal lStartRow As Long = 1&,
Optional ByVal lEndRow As Long = 0&)
Public Sub SetSorted(ByVal sSortColumns As String)
Public Property Get SortColumns()
```

Accessing list cell values

```
Public Property Get Item(ByVal vColIndex As Variant, ByVal lRowIndex As Long) As Variant
Public Property Let Item(ByVal vColIndex As Variant, ByVal lRowIndex As Long, ByRef
vCellValue As Variant)
```

Finding values

```
Public Function Find(ByVal vSearchColumns As Variant, ByVal vSearchCriteria As Variant,
                     Optional ByVal lStartFrom As Long = 1&,
                     Optional ByVal lEndTo As Long = 0&) As Long
Public Function FindFirst(ByVal vSearchColumns As Variant,
                          ByVal vSearchCriteria As Variant,
                          Optional ByVal lStartFrom As Long = 1&,
                          Optional ByVal lEndTo As Long = 0&) As Long
Public Sub RemoveDuplicates()
```

Encapsulating data access technologies

Most database objects (tables, views or queries, records, etc..) and metadata (objects definition) can be represented as tabular data, that can be easily managed with list and row variables. All existing data access technologies already have sophisticated objects that encapsulate data sets coming from their underlying data source; for example, ADO and DAO have powerful recordset objects, while the ODBC and MySQL portable APIs provide functions for creating, accessing and manipulating data sets, as many other technologies do as well.

The problem is that if we write our code using one of these data access technologies, it will be tightly coupled with that specific technology. It would then be hard to switch to another technology, because it would require us to rewrite all the data access code.

To avoid binding our data access code to a specific technology, we can encapsulate it inside a data access abstraction layer. Using list and row objects, we can hide the details of the data access technique into a wrapper class and let our application work exclusively with list and row objects. When it will be time to switch to another data access technology, we'll just have to rewrite the internals of the wrapper class.

This discussion will continue on another manual.

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty

Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this

License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new