Appendix A

How to duplicate an MS Access table using ADOX

This tutorial explains a method for creating a function that will connect with an existing Microsoft Access database, retrieve one of its tables and make a duplicate of that table's structure, using ADOX. To do this, I created the CopyTable function. This function takes two parameters. One is the name of the existing table you wish to duplicate and the other is the path to the MS Access database that will act as the data source. (This tutorial should be read in conjunction with my next tutorial on how to duplicate the data in an existing MS Access table using ADOX).

Make sure you have the following include and import statements in your header file.

```
#include "stdafx.h"

// NOTE:  Must perform these "imports" before including other headers,
// except for stdafx.h which must be included before these imports.
#import "c:\Program Files\Common Files\system\ado\msadox.dll" \
 no_namespace[3]

#import "c:\Program Files\Common Files\system\ado\msado15.dll"[4]

#include <conio.h>
```

---

[3] ADOX Fundamentals

Microsoft® ActiveX® Data Objects Extensions for Data Definition Language and Security (ADOX) is an extension to the ADO objects and programming model. ADOX includes objects for schema creation and modification, as well as security. Because it is an object-based approach to schema manipulation, you can write code that will work against various data sources regardless of differences in their native syntaxes.

ADOX is a companion library to the core ADO objects. It exposes additional objects for creating, modifying, and deleting schema objects, such as tables and procedures. It also includes security objects to maintain users and groups and to grant and revoke permissions on objects.

To use ADOX with your development tool, you should establish a reference to the ADOX type library. The description of the ADOX library is "Microsoft ADO Ext. for DDL and Security." The ADOX library file name is Msadox.dll, and the program ID (ProgID) is "ADOX". For more information about establishing references to libraries, see the documentation of your development tool.

The Microsoft OLE DB Provider for the Microsoft Jet Database Engine fully supports ADOX. Certain features of ADOX may not be supported, depending on your data provider. For more information about supported features with the Microsoft OLE DB Provider for ODBC, the Microsoft OLE DB Provider for Oracle, or the Microsoft SQL Server OLE DB Provider, see the MDAC readme file.

MSDN Online Reference

[4] See "The ADO Rosetta Stone" by Don Willitis for a complete explanation of this import statment and other ADO topics.

```
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include <ole2.h>
#include <time.h>
```

Using the C++ Coding Standard[5], choose a name for your function that is descriptive of what it does. I this example I chose CopyTable. Your return type may be anything you wish. In this example I return an integer that represents how many tables have been copied.

```
int CopyTable(const char* tableName,const char* dataSource)
{
```

An "if" statement provides error checking for calls to this function with no data source or table name by forcing the function to return negative one.

```
        if (!tableName || !dataSource)
                return -1;
```

You will need a number of different variables types for implementing this function. The HRESULT data type is a 32-bit value that is used to describe an error or warning. Here it is initialized to OK, i.e. no error. Create an underscore variant underscore t object to hold the index of a catalog. You will also need an underscore bstr underscore t to hold the path of the database you wish to connect to.

```
        // Define ADOX variable types.
        HRESULT[6] hr = S_OK;
```

---

[5] C++ Coding Standard by Todd Hoff

[6] HRESULT

The HRESULT data type is a 32-bit value that is used to describe an error or warning.

```
typedef LONG            HRESULT;
```

REMARKS

On 32-bit platforms, the HRESULT data type is the same as the SCODE data type. On 16-bit platforms, an SCODE value is used to generate an HRESULT value.

An HRESULT value is made up of the following fields:

- A 1-bit code indicating severity, where zero represents success and 1 represents failure.
- A 4-bit reserved value.
- An 11-bit code indicating responsibility for the error or warning, also known as a facility code.
- A 16-bit code describing the error or warning.

Most MAPI interface methods and functions return HRESULT values to provide detailed result information. HRESULT values are also used widely in OLE interface methods. OLE provides several macros for converting between HRESULT values and SCODE values, another common data type for error handling. For information about the OLE use of HRESULT values, see the *OLE Programmer's Reference*. For more information about the use of these values in MAPI, see Error Handling and any of the following interface methods:

MSDN Online Reference

_variant_t[7] vIndex;
_bstr_t[8] strcnn;

---

[7] variant_t Class

Microsoft Specific

A **_variant_t** object encapsulates the VARIANT data type. The class manages resource allocation and deallocation and makes function calls to **VariantInit** and **VariantClear** as appropriate.

Construction

_variant_t	Constructs a **_variant_t** object.

Operations

Attach	Attaches a **VARIANT** object into the **_variant_t** object.
Clear	Clears the encapsulated **VARIANT** object.
ChangeType	Changes the type of the **_variant_t** object to the indicated **VARTYPE**.
Detach	Detaches the encapsulated **VARIANT** object from this **_variant_t** object.
SetString	Assigns a string to this **_variant_t** object.

Operators

operator =	Assigns a new value to an existing **_variant_t** object.
operator ==, !=	Compare two **_variant_t** objects for equality or inequality.
Extractors	Extract data from the encapsulated **VARIANT** object.

MSDN Online Reference

[8] bstr_t Class

Microsoft Specific

A **_bstr_t** object encapsulates the **BSTR** data type. The class manages resource allocation and deallocation through function calls to **SysAllocString** and **SysFreeString** and other **BSTR** APIs when appropriate. The **_bstr_t** class uses reference counting to avoid excessive overhead.

Construction

_bstr_t	Constructs a **_bstr_t** object.

Create three character arrays to hold the connection string, the newly duplicated table and an optional format string if you want to customize the new table's name.

```
// Declare and initialize local variables.
char connectionString[1024];
char newTableName[128]="";
char formatString[128]="";

int i = 0, iCopied=0;
```

Initialize your connection string with the path to the database that you wish to connect to.

```
strcnn("Provider='Microsoft.JET.OLEDB.4.0';Data Source='d:\\Data\\VSS\\Sacman\\Database\\RunTimeData.mdb';");
```

For this example, one of the attributes contained in the database table I copied was a DATE object. In order to properly copy this attribute I needed to declare an ADO Enumerated Type DataTypeEnum.

```
enum DataTypeEnum columnType = adDate[9];
```

Operations

Assign	Copies a **BSTR** into the **BSTR** wrapped by a **_bstr_t**.
Attach	Links a **_bstr_t** wrapper to a **BSTR**.
copy	Constructs a copy of the encapsulated **BSTR**.
Detach	Returns the **BSTR** wrapped by a **_bstr_t** and detaches the **BSTR** from the **_bstr_t**.
GetAddress	Points to the **BSTR** wrapped by a **_bstr_t**.
GetBSTR	Points to the beginning of the **BSTR** wrapped by the **_bstr_t**.
length	Returns the length of the encapsulated **BSTR**.

Operators

operator =	Assigns a new value to an existing **_bstr_t** object.
operator +=	Appends characters to the end of the **_bstr_t** object.
operator +	Concatenates two strings.
operator !	Checks if the encapsulated **BSTR** is a **NULL** string.
operator ==, !=, <, >, <=, >=	Compares two **_bstr_t** objects.
operator wchar_t* | char*	Extract the pointers to the encapsulated Unicode or multibyte **BSTR** object.

MSDN Online Reference

[9] DataTypeEnum

Specifies the data type of a Field, Parameter, or Property. The corresponding OLE DB type indicator is shown in parentheses in the description column of the following table. For more information about OLE DB data types, see Chapter 13 and Appendix A of the *OLE DB Programmer's Reference*.

adDate	7	Indicates a date value (DBTYPE_DATE). A date is stored as a double, the whole part of which is the number of days since December 30, 1899, and the fractional part of which is the fraction of a day.

```
long columnCount = 0, tableCount = 0, columnSize =0;
```

In this example, I created time structures to show the date and time the tables were duplicated. This is optional for this tutorial.

```
// Create time structures.
time_t ltime;
struct tm *today = { 0 };
```

Next you need underscore table, column and catalog pointers as well as a properties pointer. The table[10] pointers are necessary to work with any access tables whether existing or newly created. The catalog[11] pointers are used to access the tables contained in your database.

MSDN Online Reference

10 Table Object

Represents a database table including columns, indexes, and keys.

REMARKS

The following code creates a new **Table**:

```
Dim obj As New Table
```

With the properties and collections of a **Table** object, you can:

- Identify the table with the Name property.
- Determine the type of table with the Type property.
- Access the database columns of the table with the Columns collection.
- Access the indexes of the table with the Indexes collection.
- Access the keys of the table with the Keys collection.
- Specify the Catalog that owns the table with the ParentCatalog property.
- Return date information with the DateCreated and DateModified properties.
- Access provider-specific table properties with the Properties collection.

  Note Your data provider may not support all properties of **Table** objects. An error will occur if you have set a value for a property that the provider does not support. For new **Table** objects, the error will occur when the object is appended to the collection. For existing objects, the error will occur when setting the property.

```
// Define ADOX object pointers.
// Initialize pointers on define.
// These are in the ADOX:: namespace.
_TablePtr pTable = NULL;
_ColumnPtr pColumns = NULL;
_CatalogPtr pCatalog = NULL;
_TablePtr pCopiedTable = NULL;
_ColumnPtr pColumnItem = NULL;
PropertiesPtr pProperties = NULL;
```

Further you will need an underscore connection pointer and underscore recordset pointers. The connection pointer will be used to establish a connection to your database and the recordset pointers will be used to work with the recordset objects. The ADODB:: prefix is required to create objects defined in the "msado15.dll you included in your header file.

```
// Define ADODB object pointers
ADODB::_ConnectionPtr pConnection = NULL;
ADODB::_RecordsetPtr pRsCopiedTable = NULL;
ADODB::_RecordsetPtr pRsNewTable = NULL;

// Initialize time objects to the current time.
_tzset();
```

When creating **Table** objects, the existence of an appropriate default value for an optional property does not guarantee that your provider supports the property. For more information about which properties your provider supports, see your provider documentation.

MSDN Online Reference

11 Catalog Object

Contains collections (Tables, Views, Users, Groups, and Procedures) that describe the schema catalog of a data source.

REMARKS

You can modify the **Catalog** object by adding or removing objects or by modifying existing objects. Some providers may not support all of the **Catalog** objects or may support only viewing schema information.

With the properties and methods of a **Catalog** object, you can:

- Open the catalog by setting the ActiveConnection property to an ADO Connection object or a valid connection string.
- Create a new catalog with the Create method.
- Determine the owners of the objects in a **Catalog** with the GetObjectOwner and SetObjectOwner methods.

MSDN Online Reference

```
time( &ltime );
today = localtime( &ltime );
```

The first thing you need to do is create an instance of a catalog. Do this by making a call to the CreateInstance[12] function, passing it the "__uuidof"[13] operator and the expression (Catalog).

---

[12] Defining and Instantiating ADO Objects with #import

Manipulating an Automation object takes two steps:

1. Define and instantiate a variable to be used to manipulate a COM object.

2. Instantiate an actual instance of a COM object and assign it to the variable.

With #import you can accomplish both steps in a single line of code, using the smart pointer's (_com_ptr_t) constructor to pass in a valid CLSID, or PROGID. You could also declare an instance of the object, and then use the **_com_ptr_t::CreateInstance()** method to instantiate the object. Both techniques are demonstrated in Listing 4.

Listing 4: Instantiating an ADO Connection object with #import

```
#import <msado15.dll> rename( "EOF", "adoEOF" )

...

struct InitOle {

InitOle()  { ::CoInitialize(NULL); }

~InitOle() { ::CoUninitialize();   }

} _init_InitOle_;

...

// Method #1: Declaring and instantiating a Connection object

_ConnectionPtr  Conn1( __uuidof( Connection ) );



// Method #2: Declaring and instantiating a Connection object

_ConnectionPtr  Conn1 = NULL;
```

```
HRESULT         hr    = S_OK;




hr = Conn1.CreateInstance( __uuidof( Connection ) );
```

The recommended technique is the second one because the constructor of _com_ptr_t does not return a failed HRESULT if something goes wrong. The first method is flawed because it cannot test if the creation of the ADO Connection object succeeded or failed.

In both cases, use the Visual C++ extension __uuidof(Connection), which in this case retrieves a GUID defined by #import in the .tlh file corresponding to the ADO Connection object. By passing it to **CreateInstance**, you create a valid ADO Connection object for the Connection smart pointer. There are other forms you could pass into either the constructor or **CreateInstance** to reference the ADO Connection object and accomplish the same result. For more information, see the Visual C++ documentation for topics about #import.

The only flaw with the code above is that it assumes you have imported only ADO. If you import multiple libraries and one or more of those libraries have an object with the same name, you must provide some differentiation between the two. For example, both ADO and DAO contain an object named Recordset.

Another attribute of #import, no_namespace, prevents the compiler from qualifying the classes in a namespace, which is to say the name of the library the type library defines. In the case of ADO, this is ADODB. Using no_namespace means you don't have to reference the namespace when initializing or defining variables whose types are defined by what #import generates. However, if you have many type libraries imported into your application, it is safer to omit the no_namespace attribute.

Listings 5 and 6 show the difference in your code with and without the no_namespace clause. While it may be more work to omit no_namespace, it does ensure your code will be more robust in case it uses other Automation servers whose objects might share the same name as an object found in ADO.

MSDN Online Reference

13 __uuidof Operator

Microsoft Specific

The **__uuidof** keyword retrieves the GUID attached to the expression.

```
__uuidof ( expression )
```

The *expression* can be a type name, pointer, reference, or array of that type, a template specialized on these types, or a variable of these types. The argument is valid as long as the compiler can use it to find the attached GUID.

A special case of this intrinsic is when either **0** or **NULL** is supplied as the argument. In this case, **__uuidof** will return a GUID made up of zeros.

Use this keyword to extract the GUID attached to:

- An object by the uuid extended attribute.

- A library block created with the module attribute.

The following code (compiled with ole32.lib) will display the uuid of a library block created with the module attribute:

```
// expre_uuidof.cpp

// compile with: ole32.lib

#include "stdio.h"

#include "windows.h"



[emitidl];

[module(name="joe")];

[export]

struct stuff {

   int i;

};



void main() {

   LPOLESTR lpolestr;

   StringFromCLSID(__uuidof(joe), &lpolestr);

   wprintf(L"%s", lpolestr);

   CoTaskMemFree(lpolestr);
```

```
// Get "access" to the database that contains the table you want to copy.
// Get a pointer to the table and create a duplicate of it passing
// the append function the parameters of the table to be copied.
try
{
        // Create an instance of the catalog.
        TESTHR(hr = pCatalog.CreateInstance(__uuidof(Catalog)));
```

Next create the connection string that will tell the catalog which database to connect to. Pass this string to the PutActiveConnection[14] function to open the catalog.

```
        // Create the connection string to tell the catalog which database
        // to associate with.
        sprintf(connectionString, "Provider='Microsoft.JET.OLEDB.4.0';"
                "data source=%s;", dataSource);
        strcnn = connectionString;

        //Open the catalog
        pCatalog->PutActiveConnection(connectionString);
```

To start working with existing tables in the open catalog, use the pCatalog pointer to access the table collection[15] and call the GetItem function. Pass this function the name of the table you wish to access. If this table exists, the GetItem function will return a pointer to the selected table.

```
}
```

In cases where the library name is no longer in scope, you can use __LIBID_ instead of **__uuidof**. For example:

```
StringFromCLSID(__LIBID_, &lpolestr);
```

MSDN Online Reference.

[14] ActiveConnection Property

Indicates the ADO Connection object to which the Catalog belongs.

SETTINGS AND RETURN VALUES

Sets a **Connection** object or a **String** containing the definition for a connection. Returns the active **Connection** object.

REMARKS

The default value is a null object reference.

MSDN Online Reference

[15] Tables Collection

Contains all Table objects of a catalog.

```
// Inform the catalog, which tables you, would like to return a recordset
// pointer to.
pCopiedTable = pCatalog->Tables->GetItem(tableName);
```

Now to work with a newly created table using ADOX, use the pTable pointer you created and call the CreateInstance function again passing it the "__uuidof" operator and the expression "(Table)."

```
// Create new instance of a table.
TESTHR(hr = pTable.CreateInstance(__uuidof(Table)));
```

To give your table a name, use the pTable pointer to pass a string to the PutName function.

```
// Establish a new name for the table based on the table it was
// copied from and the time it was copied.
sprintf(formatString,"%s %s",tableName,"%c");
strftime(newTableName,128,formatString,today);
pTable->PutName(newTableName);
```

To make the duplicate table, first iterate through each of the columns in the "copied table" and retrieve each one of their indexes. Store these values in the variant_t object vIndex that you created.

```
for(columnCount;columnCount<pCopiedTable->
Columns->GetCount();columnCount++)
{
vIndex = columnCount;
```

Use the pCopiedTable's Columns attribute pointer and pass the index of each column in the copied table to its GetItem function. This returns a pointer to each individual column in the table you want to copy.

REMARKS

The Append method for a **Tables** collection is unique for ADOX. You can:

- Add a new table to the collection with the **Append** method.

The remaining properties and methods are standard to ADO collections. You can:

- Access a table in the collection with the Item property.
- Return the number of tables contained in the collection with the Count property.
- Remove a table from the collection with the Delete method.
- Update the objects in the collection to reflect the current database's schema with the Refresh method.

Some providers may return other schema objects, such as a View, in the Tables collection. Therefore, some ADOX collections may contain references to the same object. Should you delete the object from one collection, the change will not be visible in another collection that references the deleted object until the Refresh method is called on the collection. For example, with the OLE DB Provider for Microsoft Jet, Views are returned with the Tables collection. If you drop a View, you must Refresh the Tables collection before the collection will reflect the change.

MSDN Online Reference

```
pColumns = pCopiedTable->Columns->GetItem(vIndex);
```

Using the pTable pointer, the pointer to the soon be to created duplicate of the "copied table," access its Columns[16] collection and call the Append[17] method. Pass to this function all of the *Name*, *Type* and *Size* of each column in the "copied table." This "appends" the new columns to the new table's Columns collection and creates a new table that is a duplicate in structure to the "copied table."

---

16 Columns Collection

Contains all Column objects of a table, index, or key.

REMARKS

The Append method for a **Columns** collection is unique for ADOX. You can:

- Add a new column to the collection with the **Append** method.

The remaining properties and methods are standard to ADO collections. You can:

- Access a column in the collection with the Item property.
- Return the number of columns contained in the collection with the Count property.
- Remove a column from the collection with the Delete method.
- Update the objects in the collection to reflect the current database's schema with the Refresh method.

Note An error will occur when appending a **Column** to the **Columns** collection of an Index if the **Column** does not exist in a Table that is already appended to the Tables collection.

MSDN Online Reference

17 Append Method (Columns)

Adds a new Column object to the Columns collection.

SYNTAX

```
Columns.Append Column [, Type] [, DefinedSize]
```

PARAMETERS

Column
The **Column** object to append or the name of the column to create and append.

Type
Optional. A **Long** value that specifies the data type of the column. The *Type* parameter corresponds to the Type property of a **Column** object.

```
pTable->Columns->Append(
                                        pColumns->Name,
                                        (DataTypeEnum)pColumns->GetType(),
                                        pColumns->GetDefinedSize());}
```

Finally, using the catalog pointer, access the Tables collection and call the Append[18] method. Pass it the expression _variant_t((IDispatch *)pTable). This will add the new table to the catalog's tables collection.

```
pCatalog->Tables->Append(_variant_t((IDispatch *)pTable));
```

Here is the complete code.

```
// Get a pointer to the "to be copied" table's columns and pass
// the attributes of each column to the append method called by
// the new table. This creates a duplicate table.
// Open the Employees table for updating as a Recordset.
for(columnCount=0;columnCount<pCopiedTable->
Columns->GetCount();columnCount++)
{
        vIndex = columnCount;
        pColumns = pCopiedTable->Columns->GetItem(vIndex);
        pTable->Columns->Append(
                pColumns->Name,
                (DataTypeEnum)pColumns->GetType(),
                pColumns->GetDefinedSize());
```

DefinedSize

Optional. A **Long** value that specifies the size of the column. The *DefinedSize* parameter corresponds to the DefinedSize property of a **Column** object.

Note An error will occur when appending a **Column** to the **Columns** collection of an Index if the **Column** does not exist in a Table that is already appended to the Tables collection.

MSDN Online Reference

[18] Append Method (Tables)

Adds a new Table object to the Tables collection.

SYNTAX

```
Tables.Append Table
```

PARAMETERS

Table

A **Variant** value that contains a reference to the **Table** to append or the name of the table to create and append.

REMARKS

An error will occur if the provider does not support creating tables.

MSDN Online Reference

```
                }

                pCatalog->Tables->Append(_variant_t((IDispatch *)pTable));
                iCopied++;
        }
        catch(_com_error &e)
        {
                // Notify the user of errors if any.
                char message[300];
                sprintf (message, "Error:\t%s\nSource:\t%s\n\n%s",
                        (LPCSTR)e.ErrorMessage(), (LPCSTR)e.Source(),           (LPCSTR)e.Description());

                MessageBox(NULL,message,"ArchiveLibrarian Error!",MB_OK);
                iArchived=-1;
        }

        catch(...)
        {
                cout << "Error occured in include files...."<< endl;
                iArchived=-1;
        }
```

Close any open connections or objects before ending the program. For the recordset pointers, check to see if their current state is open. If so, close the recordset.

```
        // Clean up objects before exit.
        if(pConnection)
                pConnection->Close();

        if (pRsCopiedTable)
                if (pRsCopiedTable->State[19] == ADODB::adStateOpen)
                        pRsCopiedTable->Close()[20];
```

---

[19] State Property

Indicates for all applicable objects whether the state of the object is open or closed.

Indicates for all applicable objects executing an asynchronous method, whether the current state of the object is connecting, executing, or retrieving.

RETURN VALUE

Returns a **Long** value that can be an ObjectStateEnum value. The default value is **adStateClosed**.

REMARKS

You can use the **State** property to determine the current state of a given object at any time.

The object's **State** property can have a combination of values. For example, if a statement is executing, this property will have a combined value of **adStateOpen** and **adStateExecuting**.

The **State** property is read-only.

MSDN Online Reference

[20] Close Method

Closes an open object and any dependent objects.

```
        if (pRsNewTable)
                if (pRsNewTable->State == ADODB::adStateOpen)
                        pRsNewTable->Close();

        return iCopied;
}
```

SYNTAX

```
object.Close
```

REMARKS

Use the **Close** method to close a Connection, a Record, a Recordset, or a Stream object to free any associated system resources. Closing an object does not remove it from memory; you can change its property settings and open it again later. To completely eliminate an object from memory, set the object variable to *Nothing* (in Visual Basic) after closing the object.

Connection

Using the **Close** method to close a **Connection** object also closes any active **Recordset** objects associated with the connection. A Command object associated with the **Connection** object you are closing will persist, but it will no longer be associated with a **Connection** object; that is, its ActiveConnection property will be set to **Nothing**. Also, the **Command** object's Parameters collection will be cleared of any provider-defined parameters.

You can later call the Open method to re-establish the connection to the same, or another, data source. While the **Connection** object is closed, calling any methods that require an open connection to the data source generates an error.

Closing a **Connection** object while there are open **Recordset** objects on the connection rolls back any pending changes in all of the **Recordset** objects. Explicitly closing a **Connection** object (calling the **Close** method) while a transaction is in progress generates an error. If a **Connection** object falls out of scope while a transaction is in progress, ADO automatically rolls back the transaction.

Recordset, Record, Stream

Using the **Close** method to close a **Recordset**, **Record**, or **Stream** object releases the associated data and any exclusive access you may have had to the data through this particular object. You can later call the Open method to reopen the object with the same, or modified, attributes.

While a **Recordset** object is closed, calling any methods that require a live cursor generates an error.

If an edit is in progress while in immediate update mode, calling the **Close** method generates an error; instead, call the Update or CancelUpdate method first. If you close the **Recordset** object while in batch update mode, all changes since the last UpdateBatch call are lost.

If you use the Clone method to create copies of an open **Recordset** object, closing the original or a clone does not affect any of the other copies.

MSDN Online Reference