A short tutorial on your first Macromedia Director Xtra programming

By Michał Ścioch and Zbigniew "Ziggi" Szczęsny

Introduction

The process of creation of xtras is fascinating though not easy art. By many programmers it is perceived a very specific activity, far beyond the scope of inexperienced developers.
Several reasons stand behind this opinion, the most important ones are: a bit messy XDK documentation, lack of available samples and publications and – for the time being - limited support from Macromedia available for independent developers.
The purpose of this article is to improve the situation and familiarize the readers with programming techniques used while coding xtras. Step-by-step we will provide you a complete description how to build your very first xtra.

Fundamentals

The sample xtra we will teach you how to code will be a very simple one. This will be a scripting xtra – that means the xtra the functionality of which is only available through Lingo. Our xtra will add a special Lingo command to Director: "ssOpenFolder()". Execution of this command will force the system "select folder" dialog to open and the selected directory string will be returned to Lingo.

Because this new Lingo command is going to be a global function we have to take care about its name what should be unique to avoid possible conflicts with existing Lingo commands and commands added to Lingo by other xtras installed by user. "ssOpenFolder" seems to be a good name. Prefix "ss" stands for "StarSoft" – the name of company owned by Michal.

Please notice we limit our discussion to Windows-based operating systems. Because of limited popularity of Macintosh systems in Poland we have never gone into Mac specific aspects of xtras programming.

Our sample xtra will be developed under Borland C++ Builder 6.0 programmatic environment with XDK v. 8.5 available from Macromedia: http://www.macromedia.com/support/xtras/xdks/xdk.html.
Director for Windows will be necessary to test our product. We tested it with Director v. 8.5 but we believe it should work properly with previous versions of Director as well.

So, let's begin !

We should organize our work on disk drive, so let's create a new folder dedicated for xtra development and let's call it "C:\Xtras" for instance. Inside this folder let's create a subfolder: "C\Xtras\OpenFolder". Before we go into scripting we should ensure we have an access to necessary XDK files, so unpack the "xdk85_win.zip" file provided by Macromedia and copy its complete "Include" directory into C:\Xtras, so C:\Xtras\Include subfolder will appear on your hard disk.

We can notice this directory is populated with many files (most of them has ".h" extension) but at the moment we will only need one of them called "moaxtra.h". Unfortunately, Macromedia has prepared XDK for Microsoft Visual Studio, so it is necessary to modify "moaxtra.h" file in a purpose to avoid compilation problems with Borland Builder. Necessary corrections are as follows (line numbering refers to the original file):

Line 280:
Before "#define EXTERN_GUID(name) \" it's a need to insert "#undef EXTERN_GUID" because otherwise macro "EXTERN_GUID" will be re-declared and Builder will return a warning.

Line 613:
We need to delete completely the entry "EXTRA_MOA_DELETE(macro_CLASS) \" because otherwise Builder will return a warning due to multiple definition of "DELETE" operator.

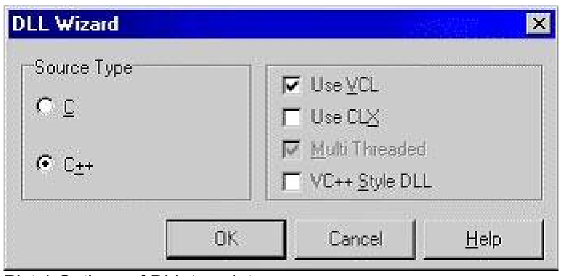
Lines 860-861:
We need to delete completely these two lines because Builder does not allow to insert comment lines in the body of macros.

Lines 836, 845, 870, 895, 927, 950, 976, 1063, 1093, 1121 and 1171:
We need to add a line containing text "return 0;\" just after "X_EXIT \". Without these entries Builder will return "function returned no value" compilation warnings.

For your convenience we provide a modified "moaxtra.h" file here. Please notice this file contains the interface core for all xtras, so you will use it for any further xtra development.

Project

Open Borland Builder, take "File > New > Other" menu command. The list of templates will appear. Because xtras are DLLs (with an extension changed into ".x32"), select "DLL Wizard". The following dialog should appear:



Source Type: C [✓ C++]

[✓ Use VCL]
Use CLX
[✓ Multi Threaded]
VC++ Style DLL

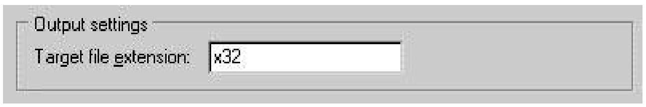
Pict.1 Options of DLL template

Please, leave these settings unchanged as projects compiled with “Use VCL” unmarked used to produce hard to debug runtime errors.

When Builder create a new, empty DLL project we shall save it in the “C:\Xtras\OpenFolder” directory. Please, change the name of the module file “Unit1.cpp” to “OpenFolder.ccp”, and save the project file as “OpenFolder.bpr”.

The file “OpenFolder.ccp” is a base DLL file. It contains one function called “DllEntryPoint”, what is called by the system each time the DLL is loaded or unloaded. We will not modify this file in our simple example. In case of more complicated xtra it is possible to modify this function, so appropriate data structures will be initiated while DLL is loaded or cleared while unloaded.

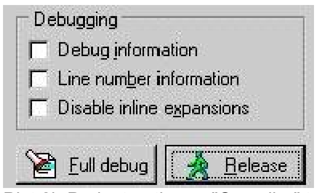
Now we will change default compilation options for our project. Select “Project > Options” menu item (or Shift+Ctrl+F11) and choose “Application” tag. Change “Target file extension” to “x32”:



Output settings
Target file extension: x32

Pict.2a Project options - "Application" tag

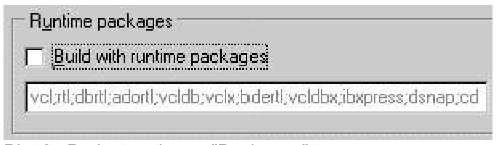
Select the “Compiler” tag and press “Release” button – this will ensure the project will be compiled without unnecessary debugging code:



Debugging
Debug information
Line number information
Disable inline expansions

Pict.2b Project options - "Compiler" tag

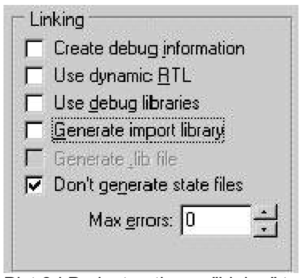
Select the “Packages” tag and obligatory uncheck “Build with runtime packages” – this is the must as otherwise the code compiled will expect Borland Builder runtime components to be installed on the end-user machine what is normally untrue:



Runtime packages
Build with runtime packages
vcl;rtl;dbrtl;adortl;vcldb;vclx;bdertl;vcldbx;ibxpress;dsnap;cd

Pict.2c Project options - "Packages" tag

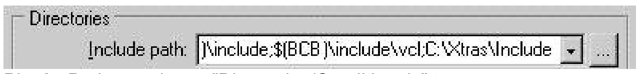
Select the “Linker” tag and uncheck unnecessary RTL and import libraries:



Linking
Create debug information
Use dynamic RTL
Use debug libraries
Generate import library
Generate .lib file
[✓ Don't generate state files]
Max errors: 0

Pict.2d Project options - "Linker" tag

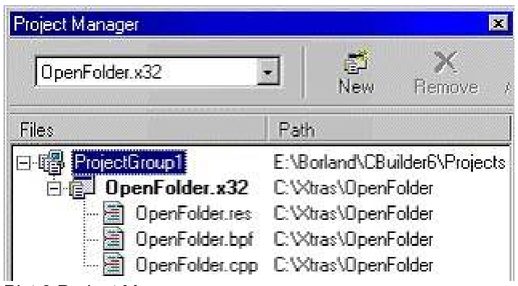
And finally select the tag “Directories/Conditionals” and instruct Builder where to look for necessary XDK files by appending the “Include path” field with the following information: “;C:\Xtras\Include”:



Directories
Include path:)\include;$(BCB)\include\vcl;C:\Xtras\Include

Pict.2e Project options - "Directories/Conditionals" tag

At that moment we have completed options setup and we should save our project (File > Save).
We can now observe the “Project Manager” window (View > Project Manager or Ctrl+Alt+F11) what should look like this:



```
ProjectGroup1        E:\Borland\CBuilder6\Projects
  OpenFolder.x32     C:\Xtras\OpenFolder
    OpenFolder.res   C:\Xtras\OpenFolder
    OpenFolder.bpf   C:\Xtras\OpenFolder
    OpenFolder.cpp   C:\Xtras\OpenFolder
```

Pict.3 Project Manager

You can see that the resource file “OpenFolder.res” and DLL project file “OpenFolder.bpf” were automatically generated by Builder and we don’t need to worry about it. These files are just necessary for Builder itself and are not the point of our interest.

“CScript” class

All steps we have already completed were just an introduction to the actual programming. We have constructed the necessary frame for functional modules constituting our xtra.
The process of coding a scripting xtra consists of building a class definition implementing the MOA interface called “IMoaMmXScript”. Here we will name such a class “CScript” where letter “C” is just a handy abbreviation for “class”. Its useful to stick to a convention where each class is defined within a separate .cpp file, so now we will create a file called “script.cpp”. Select “File > New > Unit” and save a new file in “C:\Xtras\OpenFolder” directory as “script.cpp”. Please, notice Builder will automatically create the relating header file: “script.h”.

And here we begin: open “script.h” file and add necessary entries referring to XDK header files:

```
#include "moaxtra.h"
#include "mmixscrp.h"
#include "mmiutil.h"
```

Remember, we have modified the file “moaxtra.h” appropriately for usage with Borland Builder!

Because MOA was developed according to Microsoft’s *Component Object Model* (COM) technology, we are obligated to follow its specification what tells to identify every class or interface to be used within COM with a unique 16-bytes long number called GUID (Global Unique Identifier). Such a number must be really unique – multiple problems can occur in case two identical GUIDs are found on the same system. Microsoft provides a special tool for GUID generation called “guidgen.exe”. You can find this software within Visual Studio C++ or you may get it here :-)

Our class “CScript” is undoubthly COM class, so we have to provide a unique GUID number using guidgen.exe tool’s “define guid” algorithm. The result comes as follows:

```
// {8349B742-A563-11d6-8677-0010A709D781}
DEFINE_GUID(<<name>>, 0x8349b742, 0xa563, 0x11d6, 0x86, 0x77,
                0x0, 0x10, 0xa7, 0x9, 0xd7, 0x81);
```

so we must give it a proper name:

```
// {8349B742-A563-11d6-8677-0010A709D781}
DEFINE_GUID(CLSID_CScript, 0x8349b742, 0xa563, 0x11d6, 0x86, 0x77,
                0x0, 0x10, 0xa7, 0x9, 0xd7, 0x81);
```

where prefix “CLSID” indicates we are defining a class rather than interface (then we would use a “IID” prefix).

Now we are in charge to declare the body of “CScript” class, what means providing the list of fields (variables) and methods (functions) being used. But we have to remember we are in scope of COM technology, where it’s not that simple any more. In case we were obligated to do it by our own we would have some hard task to do but thanks Macromedia we can use dedicated macros provided with XDK, that makes our life easier. Using macros is typical for xtra development and we will meet them all day long.

Let’s declare “CScript” class now – our xtra is going to be a very simple one, we don’t need any variables etc. Just one plain function makes an actual content of the xtra. Let’s call it “OpenFolder” – technically this name has nothing to do neither with the name of the xtra itself nor with functions added to Lingo by the xtra:

```
EXTERN_BEGIN_DEFINE_CLASS_INSTANCE_VARS(CScript)
  void OpenFolder(PMoaMmCallInfo callPtr);
EXTERN_END_DEFINE_CLASS_INSTANCE_VARS
```

Macros:

"EXTERN_BEGIN_DEFINE_CLASS_INSTANCE_VARS" and "EXTERN_END_DEFINE_CLASS_INSTANCE_VARS"

are here to declare a COM class as supported with XDK. The name of a new class is visible within parenthesis. In between two macros there is an actual class body – as already said one function “OpenFolder” is enough for our purpose. But we are on duty to explain its parameter (callPtr). This is a pointer to “MoaMmCallInfo” structure where, in between others, calling parameters (as scripted in Lingo) are stacked and providing the return of function result back to Lingo. Thanks to this pointer our function is capable of accepting arguments from Lingo and returning result values back to Lingo. There is not too many useful functions where we could skip “callPtr” parameter.

MOA was developed as a set of programmatic interfaces integrating more or less independent elements. Consequently, a programmer developing an xtra in charge of “binding” it’s code with interfaces existing within host application. As for now we have “CScript” class properly declared but how Director engine can “guess” which one of it’s many interfaces our class is going to use? Also, we ought to provide a proper interface binding at the xtra side – for the time being this functionality is missing in the “CScript” class code. The first can be improved with the usage of another macro delivered with XDK:

```
EXTERN_BEGIN_DEFINE_CLASS_INTERFACE(CScript, IMoaMmXScript)
  EXTERN_DEFINE_METHOD(MoaError, Call, (PMoaMmCallInfo))
EXTERN_END_DEFINE_CLASS_INTERFACE
```

In the first line we provide the information “CScript” class is going to use “IMoaMmXScript” interface of the host application (Director). In the next line the method “Call” of this interface is specified. BTW – “Call” is in fact the one and only method of “IMoaMmXScript” interface, but do not treat it as a rule for other MOA interfaces. “Call” method of “IMoaMmXScript” interface is called each time the Director’s engine detects Lingo is using function provided by the xtra. All parameters sent from Lingo at this moment are then stacked in the “MoaMmCallInfo” structure and the pointer to this structure (PMoaMmCallInfo) is sent to “Call” method. Since now Director waits till all the code in the scope of “Call” method is executed and afterwards takes values from the “MoaMmCallInfo” structure and sends them back to Lingo. The last is triggered by returning value of the “Call” method itself – the “MoaError”. “MoaError” is a numeric container for the error code. Most of MOA interfaces returns this type of value.
This is a general template for all scripting xtras and it is worth to remember.

Please, notice the last parameter of EXTERN_DEFINE_METHOD macro call is a list. This list contains only one element of “PMoaMmCallInfo” type an that is a general rule.

There is one little problem remaining – how Director “knows” a particular function is belonging to a particular xtra? This is controlled by the xtra registration procedure but we will take care about this later.

A little comment: we had a full freedom while declaring “CScript” class. There were no limits on it’s content. But at the moment of implementation of “IMoaMmXScript” interface we lost the most of this freedom. Simply speaking we had to fit to what was already declared in XDK header files by Macromedia engineers. Director engine expects that the “Call” method of “IMoaMmXScript” interface uses only one parameter pointing to the “MoaMmCallInfo” variable (structure). In case we try to code EXTERN_DEFINE_METHOD macro call differently, so the parameter list would be shorter or longer – we should be prepared to get familiarized with “Director.exe: Abnormal program termination” message window.

So at the moment our “script.h” header file should look like that:

```
#ifndef scriptH
#define scriptH

#include "moaxtra.h"
#include "mmixscrp.h"
#include "mmiutil.h"

// {8349B742-A563-11d6-8677-0010A709D781}
DEFINE_GUID(CLSID_CScript, 0x8349b742, 0xa563, 0x11d6, 0x86, 0x77,
                        0x0, 0x10, 0xa7, 0x9, 0xd7, 0x81);

EXTERN_BEGIN_DEFINE_CLASS_INSTANCE_VARS(CScript)
  void OpenFolder(PMoaMmCallInfo callPtr);
EXTERN_END_DEFINE_CLASS_INSTANCE_VARS

EXTERN_BEGIN_DEFINE_CLASS_INTERFACE(CScript, IMoaMmXScript)
  EXTERN_DEFINE_METHOD(MoaError, Call, (PMoaMmCallInfo))
EXTERN_END_DEFINE_CLASS_INTERFACE

#endif
```

“CScript” class is already properly declared, and now we will try to implement its functionality as we promised in the introduction. Do not forget our target is to deliver “ssOpenFolder()” function to Lingo and our xtra has to provide methods for browsing and selecting disk directories. This functionality will be contained within the code of the “script.cpp” file, so now we should save any changes made to “script.h” and switch to “script.cpp”.

Its also important to mention that until now we were focused on instructing the Director engine which of its multiple MOA interfaces our xtra will use. Now we will focus on implementing the proper interface on the xtra side, so it matches appropriate interface of the host application.

First of all we have to define a few symbols used within XDK header files and important for the compiler. Missing them will cause multiple compilation problems. These symbols have to be defined in a “.cpp” file in case its relating “.h” file refers directly to XDK:

```
#define CPLUS
#define WINDOWS
#define _WINDOWS
#define WIN32
```

Implementation of the “CScript” class will start with the following macro:

```
BEGIN_DEFINE_CLASS_INTERFACE(CScript, IMoaMmXScript)
END_DEFINE_CLASS_INTERFACE
```

Necessary explanation at this moment is that according to MOA documentation in C++, a MOA interface is declared as an abstract class. The methods in an interface are declared as pure virtual functions. The MOA macros for declaring interfaces provide the implementation automatically when you specify C++ as your coding model. Pure virtual functions are member functions with no implementation in the class being declared. A class that declares only pure virtual functions is an abstract class. Thus the interface is declared as an abstract C++ class.

So in our example, a class actually implementing “IMoaMmXScript” interface is a not directly our “CScript” class but a class inherited from “CScript” class. In this model “CScript” is an abstract class and the inheritance is provided by the “BEGIN_DEFINE_CLASS_INTERFACE” macro. Name of the new class inherited from “CScript” will reflect both it’s parent abstract class and the interface implemented: "CScript_IMoaMmXScript". This is the actual class the “Call” method is belonging to.

We have to start with building the constructor and destructor for CScript class so we have to notice that at the moment when we declared our “CScript” class with XDK macros five minutes ago – we caused two new methods have been auto magically declared: “MoaCreate_CScript” and “MoaDestroy_CScript”. We were so lucky – Macromedia declared necessary constructor and destructor methods for “CScript” class for us! So, we have to follow these names and define both functions within “script.cpp” file, but immediately we can observe some differences to “normal” programming. Usually we would write a method called “CScript::CScript()”, because this is the way to build a class constructor in C++. But here we are in the scope of COM technology, so we have to employ some XDK macros again:

```
STDMETHODIMP MoaCreate_CScript(CScript FAR * This)
{
  X_ENTER
    MoaError err = kMoaErr_NoErr;
    X_STD_RETURN(err);
  X_EXIT
  return 0;
}

STDMETHODIMP_(void) MoaDestroy_CScript(CScript FAR * This)
{
  X_ENTER
  X_EXIT
}
```

As you see, not really much coding at the moment. In case of more advanced xtras, this is the place where variables are initialized and necessary functions are called, But in our simple example constructor and destructor practically do nothing.

A few words of comment, macros: STDMETHODIMP, X_ENTER, X_EXIT, X_STD_RETURN are all required and there is no need to explain their functionality. Just presume this structure is a consequence of MOA and is necessary for our xtra to work properly. More important is that both methods are parameterized with “This”, what means with a pointer to “CScript” class. Thanks that, if needed, we may have an access to fields and properties of the main class. Also it is important to notice “kMoaErr_NoErr” constant. This is a pre-defined error code. This error code means “no error”. We can easily take this approach as according to a general rule: “who does nothing, makes no mistakes”, we are on the safe side. Our constructor does nothing, so we expect no error.

Happily we managed to complete constructor and destructor for “CScript” class but we remember “CScript” is an abstract class, while the critical “Call” method of “IMoaMmXScript” interface will belong to inherited "CScript_IMoaMmXScript" class. So we are on duty to provide constructor and destructor for this class as well – somehow that was not provided by any macro, so we do it in a “classic” way:

```
CScript_IMoaMmXScript::CScript_IMoaMmXScript(MoaError FAR * pErr)
{
  *pErr = kMoaErr_NoErr;
}

CScript_IMoaMmXScript::~CScript_IMoaMmXScript()
{}
```

Again, both methods are as simple as possible. The only comment is that the pointer to “MoaError” is present in the constructor definition and that its return value is “no error” error code.
Destructor method has no parameters at all.

We remember that when Director engine notices execution of Lingo command “ssOpenFolder()”, it triggers its “IMoaMmXScript” interface and finally “Call” method of "CScript_IMoaMmXScript" is called. It is interesting to know the same “Call” method is called regardless the actual number of Lingo accessible functions implemented in an xtra. Information what function should be used when we script in Lingo “ssOpenFolder()” and what function should be called when we script something else is provided through an index field called “methodSelector” present within “MoaMmCallInfo” structure. Each function is identified through its index, where index equal 0 (zero) is reserved to a “new()” method, what is the Lingo creating new instance of an xtra. Taking this into account, a typical implementation of the “Call” method looks like this:

```
STDMETHODIMP CScript_IMoaMmXScript::Call(PMoaMmCallInfo callPtr)
{
  X_ENTER
```

```
    MoaError err = kMoaErr_NoErr;

    enum {
      m_new = 0,
      m_ssOpenFolder
    };

    switch ( callPtr->methodSelector )
    {
      case m_new:
        break;
      case m_ssOpenFolder:
        pObj->OpenFolder(callPtr);
        break;
    }

    X_STD_RETURN(err);
  X_EXIT
  return 0;
}
```

As you can see, at the beginning (after defining error code) we declare enumerating type facilitating access to "callPtr->methodSelector". That is useful, as if our xtra would contain many functions we could get lost with indexes while now we can use custom function names. Then we provide links to actual functions. In our example we don't implement explicit "new()" method, because – that will be explained later – our "ssOpenFolder()" function is going to be global, so the xtra instance will be automatically created by Director while "ssOpenFolder()" is executed first time. In this article we don't go too far, so we will not explain here how to manage in case explicit "new()" command is necessary.

In case the parameter of the "Call" function was "ssOpenFolder" the index search leads to "OpenFolder" function call with "callPtr" parameter. We have to remember we are "inside" an instance of "CScript_IMoaMmXScript" class and that class was created by a XDK macro from "CScript" class. This macro provided an access from the child class instance to its parent class instance – this is possible because the variable "pObj" is in fact a pointer to an instance of the parent class. Thus "pObj->OpenFolder(callPtr) is a call to "OpenFolder" function of "CScript" class. So what is "callPtr" parameter of this call? This is a pointer to "MoaMmCallInfo" structure that is necessary because this structure is used as an obligatory container for eventual parameters "OpenFolder" function should use (these ought to be sent to "MoaMmCallInfo" in advance) and for eventual return values (in our example this is going to be path to a selected directory).

OK, but what about "OpenFolder" method of "CScript" class itself? – Yes, we have to implement it right away. It's here where we finally can express our creativity.

According our plan we would like to implement a function opening "select folder" dialog and returning a path to selected directory. We will do this soon, be patient, but because there is still a long way ahead, now we will do something more elementary. We will display a simple message window. This is good enough to test is our xtra working or not.

Let's write something like this:

```
void CScript::OpenFolder(PMoaMmCallInfo callPtr)
{

  MessageBox(0, "OpenFolder", "My xtra", 0);

}
```

There is no need to explain WinApi "MessageBox" function but it's necessary to say that according to XDK Macromedia provides a specific MOA interfaces to display modal dialogs, so we should not be surprised to observe some unexpected behaviours (flickering, etc.) while a modal dialog was displayed with WinApi interface instead of MOA. Again, we remind you the purpose of this is only to let us check the xtra functionality with something as simple as possible.

So, "script.cpp" file is ready now and it should looks like this at the moment (we will modify it further on to implement "a real" OpenFolder method):

```
#define CPLUS
#define WINDOWS
#define _WINDOWS
#define WIN32

#pragma hdrstop
#include "script.h"
#pragma package(smart_init)

BEGIN_DEFINE_CLASS_INTERFACE(CScript, IMoaMmXScript)
END_DEFINE_CLASS_INTERFACE

STDMETHODIMP MoaCreate_CScript(CScript FAR * This)
{
  X_ENTER
    MoaError err = kMoaErr_NoErr;
    X_STD_RETURN(err);
  X_EXIT
  return 0;
}

STDMETHODIMP_(void) MoaDestroy_CScript(CScript FAR * This)
{
  X_ENTER
  X_EXIT
}

CScript_IMoaMmXScript::CScript_IMoaMmXScript(MoaError FAR * pErr)
{
  *pErr = kMoaErr_NoErr;
}

CScript_IMoaMmXScript::~CScript_IMoaMmXScript()
{}

STDMETHODIMP CScript_IMoaMmXScript::Call(PMoaMmCallInfo callPtr)
{
  X_ENTER
    MoaError err = kMoaErr_NoErr;

    enum {
      m_new = 0,
      m_ssOpenFolder
    };

    switch ( callPtr->methodSelector )
    {
      case m_new:
        break;
      case m_ssOpenFolder:
        pObj->OpenFolder(callPtr);
        break;
    }

    X_STD_RETURN(err);
  X_EXIT
  return 0;
}

void CScript::OpenFolder(PMoaMmCallInfo callPtr)
{
  MessageBox(0, "OpenFolder", "My xtra", 0);
}
```

As you see leave default Borland Builder compiler directives unchanged. These are “#pragma hdrstop” and “#pragma package(smart_init). The first one is about to inform the compiler where to stop precompilation of header files and is important when VCL components are used. Because XDK header files should not be precompiled at all we stand a rule: first “#pragma hdrstop” directive and then inclusion of XDK headers (direct or indirect).
The second directive is only used when VCL components are used. Otherwise it does nothing, so we leave it.

If we try to compile (“Project > Make Project” or Ctrl+F9)our project we should observe the desired “Done: Make” message window and we should observe the “OpenFolde.x32” file appeared in our project folder but... in case we try our xtra in Director we notice it will not work. Why? Because it simply was not registered by the Director engine! The xtra registration, understood as a proper notification of the Director engine about the new xtra is a necessary and important part of xtra development. In the next chapter we take care about this step.

“CRegister” class

At least one class of each xtra has to implement “IMoaRegister” interface to let the Director register an xtra. We remind you that here the word “registration” does not mean provision of some “serial number” but the process when the Director is informed about the xtra and makes a proper environment for this xtra to work (for instance, in case of a ‘tool’ or ‘asset’ xtra some entries have to be done within Director’s menu system, tool icons etc.).

This is not the Director what is in charge of this process. When requested the Director only provides a pointer to a special ‘Application Cache’ object what is a kind of database where information about all active xtras is stored. During Director startup it’s “Xtras” directory is browsed for ‘.x32’ files. Director presumes such files somehow implement “IMoaRegister” interface and tries to send a pointer to the “Application Cache” object through this interface to an xtra. Then it is the xtra itself in charge to make necessary entries to this database, so it can be used by the Director engine during further initialization steps.

Because all interfaces are implemented by classes inherited from an abstract class we could use our “CScript” abstract class to implement many interfaces (finally each interface will be managed by a separated inherited class) but we take more universal and scalable approach and we create a next abstract class only in charge of xtra registration.

We call this new class “CRegister”. We create a new module and save it in the “register.cpp” file. A related “register.h” header file will be created automatically by Builder. We open this header file and at first include necessary XDK header files:

```
#include "moaxtra.h"
#include "moastdif.h"
#include "script.h"
```

We need to include “script.h” file as this is the header file of “CScript” class what is going to be registered.

Then we continue like we did in case of “script.h” file. We have to generate a new GUID and assign it to “CRegister” class:

```
// {A98C3041-A572-11d6-8677-0010A709D781}
DEFINE_GUID(CLSID_CRegister, 0xa98c3041, 0xa572, 0x11d6, 0x86, 0x77,
                             0x0, 0x10, 0xa7, 0x9, 0xd7, 0x81);
```

Then we declare “CRegister” class itself (we do not need and internal functions or variables) and declare implementation of “IMoaRegister” interface together with its “Register” method:

```
EXTERN_BEGIN_DEFINE_CLASS_INSTANCE_VARS(CRegister)
EXTERN_END_DEFINE_CLASS_INSTANCE_VARS

EXTERN_BEGIN_DEFINE_CLASS_INTERFACE(CRegister, IMoaRegister)
  EXTERN_DEFINE_METHOD( MoaError, Register, (THIS_ PIMoaCache pCache,
                                             PIMoaDict pDict))
EXTERN_END_DEFINE_CLASS_INTERFACE
```

As you see, the “Register” method of “IMoaRegister” interface returns “MoaError” type of result and it is fed with two arguments of “PIMoaCache” and “PIMoasDict” type. These are pointers to already mentioned ‘Application Cache’ object. Macro “THIS_” is only important in C programming language and it its visible as empty string for C++ compiler.

Our “register.h” file should look like this then:

```
#ifndef registerH
#define registerH

#include "moaxtra.h"
#include "moastdif.h"
#include "script.h"

// {A98C3041-A572-11d6-8677-0010A709D781}
DEFINE_GUID(CLSID_CRegister, 0xa98c3041, 0xa572, 0x11d6, 0x86, 0x77,
                             0x0, 0x10, 0xa7, 0x9, 0xd7, 0x81);

EXTERN_BEGIN_DEFINE_CLASS_INSTANCE_VARS(CRegister)
EXTERN_END_DEFINE_CLASS_INSTANCE_VARS

EXTERN_BEGIN_DEFINE_CLASS_INTERFACE(CRegister, IMoaRegister)
  EXTERN_DEFINE_METHOD( MoaError, Register, (THIS_ PIMoaCache pCache,
                                             PIMoaDict pDict))
EXTERN_END_DEFINE_CLASS_INTERFACE

#endif
```

Now, let’s take care about “register.cpp”. There is many analogies with “script.cpp” as well. We declare the following symbols at the beginning:

```
#define CPLUS
#define WINDOWS
#define _WINDOWS
#define WIN32
```

Then class definition must include necessary XDK macros:

```
BEGIN_DEFINE_CLASS_INTERFACE( CRegister, IMoaRegister)
END_DEFINE_CLASS_INTERFACE
```

Then we must provide constructors and destructors for “CRegister” and “CRegister_IMoaRegister” classes:

```
STDMETHODIMP MoaCreate_CRegister(CRegister FAR * This)
{
  X_ENTER
    MoaError err = kMoaErr_NoErr;
    X_STD_RETURN(err);
  X_EXIT
  return 0;
}

STDMETHODIMP_(void) MoaDestroy_CRegister(CRegister FAR * This)
```

```
{
  X_ENTER
  X_EXIT
}

CRegister_IMoaRegister::CRegister_IMoaRegister( MoaError FAR * pErr)
{
  *pErr = kMoaErr_NoErr;
}

CRegister_IMoaRegister::~CRegister_IMoaRegister()
{}
```

As the code above is nearly identical to the code of “script.cpp” file we do not discuss it again.

Now it is a time to the key part of “Cregister” calss – the implementation of it’s “Register” method. This method will be implemented by the “CRegister_IMoaRegister” class inherited from “CRegister” abstract class. The purpose of this implementation is to provide a proper entry to “Application Cache” database object. The procedure for this is fairly schematic and repetitive for every new xtra.

First we have to prepare some static character table and we initialize it with a text required for xtra registration. This seems strange but this text will be scanned by Director and will it will be a source of methods supported by the xtra. This character table will be called “msgTable”:

```
static char msgTable[] = {
  "xtra OpenFolder \n"
  "new object me \n"
  "-- Functions: \n"
  "* ssOpenFolder -- This function will open the open_folder dialog \n"
};
```

As you see the first line contains a keyword “xtra” and the name of our xtra. Please notice this name has nothing to do with a name of its ‘.x32’ file. Expression “/n” means “end of line” and this suggest the complete information may be somehow displayed in Director. This is true – we will come back to this point soon.

Then the list of all functions (as visible for Lingo) are provided. First there is a “new” function with parameters required for new xtra instance creation. This will not be used by our simple xtra but we may leave it unchanged.

The third line as a comment line – it starts with “--“ and will be ignored.

The last line contains the name “ssOpenFolder” and this is the name of the only one function provided by our xtra as visible for Lingo with a comment about its functionality. Please, notice a star at the beginning of this line. This is very important – a star symbol means that the following function is global. That means it is always available for Lingo, so there is no need to create an explicit instance of the xtra with a “new” command to be allowed to call this function.

All the comments in the text of the “msgTable” are of that importance that names of functions supported by the xtra and appropriate comments are accessible from Director with “put Xtra(“xtra_name”).interface()” Lingo command. This is of course a very nice feature in case we lost documentation of an xtra.

Another point is that the order the functions are listed within the “msgTable” is the order the functions are indexed by the “methodSelector” indexer of the “MoaMmCallInfo” structure (look the implementation of the “Call” method of the “CScript_IMoaMmXScript” class). So the “new” function will be indexed as “0” and “ssOpenFolder” as “1”.

Now we can code the actual implementation of the “Register” method. All we need is to use interface methods passed through parameters:

```
STDMETHODIMP_(MoaError) CRegister_IMoaRegister::Register (
                                        THIS_ PIMoaCache pCache,
                                        PIMoaDict pDict)
{
  MoaError err;
  X_ENTER
    PIMoaDict pRegDict;

    err = pCache->AddRegistryEntry(
            pDict,
            &CLSID_CScript,
            &IID_IMoaMmXScript,
            &pRegDict);
    if (err == kMoaErr_NoErr)
    {
      pRegDict->Put(kMoaMmDictType_MessageTable, msgTable, 0,
                  kMoaMmDictKey_MessageTable);
    }

    X_STD_RETURN(err);
  X_EXIT
  return err;
}
```

Let’s look closer on these parameters:
Parameter “pCache” of “PIMoaCache” type is a pointer to “IMoaCache” interface what is responsible for communication with the “Application Cache” object. This interface has many methods but we will use only one: “AddRegistryEntry”, what will add a record about our xtra.
Parameter “pDict” is a pointer to “IMoaDict” interface managing internal Lingo dictionary of particular Director’s instance.

The registration procedure is as follows:
First we call “AddRegistryEntry” method of “IMoaCache” interface with necessary parameters:

- “pDict”, the pointer to the GUID of the “CScript” method
- the pointer to the GUID of the interface implemented by the “CScript” class (this is of course the “IMoaMmScript” interface and its GUID is provided by the “IID_IMoaMmXScript” constant defined within XDK)
- a variable which will contain a pointer to the “IMoaDict” interface (this variable and its type has been just declared one line above)

In case addition of the new record to “Application Cache” database object was successful (“err==kMoaErr_NoErr”) we call the “Put” method of the “IMoaDict” interface (the pointer to this interface has been just returned by the “AddRegistryEntry” method of the “IMoaCache” interface). We provide our “msgTable” as an argument for this method and we inform that “msgTable” is of “kMoaMmDictType_MessageTable” type. And this ends the xtra registration. We advice you to browse the XDK documentation for more detailed information on this procedure.

Here you have a complete “register.cpp” file listing:

```
#define CPLUS
#define WINDOWS
#define _WINDOWS
#define WIN32

#pragma hdrstop
#include "register.h"
#pragma package(smart_init)

BEGIN_DEFINE_CLASS_INTERFACE( CRegister, IMoaRegister)
END_DEFINE_CLASS_INTERFACE

STDMETHODIMP MoaCreate_CRegister(CRegister FAR * This)
{
            X_ENTER
            MoaError err = kMoaErr_NoErr;
              X_STD_RETURN(err);
            X_EXIT
  return 0;
}

STDMETHODIMP_(void) MoaDestroy_CRegister(CRegister FAR * This)
{
            X_ENTER
            X_EXIT
}

CRegister_IMoaRegister::CRegister_IMoaRegister( MoaError FAR * pErr)
{
  *pErr = kMoaErr_NoErr;
}

CRegister_IMoaRegister::~CRegister_IMoaRegister()
{}

static char msgTable[] = {
  "xtra OpenFolder \n"
  "new object me \n"
  "-- Functions: \n"
  "* ssOpenFolder -- This function will open an open_folder dialog \n"
};

STDMETHODIMP_(MoaError) CRegister_IMoaRegister::Register (
                                             THIS_ PIMoaCache pCache,
                                             PIMoaDict pDict){
  MoaError err;
  X_ENTER
    PIMoaDict pRegDict;

    err = pCache->AddRegistryEntry(
            pDict,
            &CLSID_CScript,
            &IID_IMoaMmXScript,
            &pRegDict);
    if (err == kMoaErr_NoErr)
    {
      pRegDict->Put(kMoaMmDictType_MessageTable, msgTable, 0,
                    kMoaMmDictKey_MessageTable);
    }

    X_STD_RETURN(err);
  X_EXIT
  return err;
 }
```

Is this enough - we have declared two classes and we have determined what interfaces are implemented by these classes? No! – Director still needs some more. We have to use a few XDK macros what will finally “bind” all classes with their interfaces. Fortunately, there is not so much coding in this point.

Let’s create a new module and let’s call its file “xtra.cpp”. Similarly to “registry.cpp”, this module will be used each time we develop a new xtra (with some minor adjustments, of course):

In the “xtra.h” file we add only one line:

```
#ifndef xtraH
#define xtraH

#include "register.h"

#endif
```

In the “xtra.cpp” file we must add necessary macros. Here you are the final file:

```
#define INITGUID
#define CPLUS
#define WINDOWS
#define _WINDOWS
#define WIN32

#pragma hdrstop
#include "xtra.h"
#pragma package(smart_init)

BEGIN_XTRA

  BEGIN_XTRA_DEFINES_CLASS(CRegister, 1)
    CLASS_DEFINES_INTERFACE(CRegister, IMoaRegister, 1)
  END_XTRA_DEFINES_CLASS

  BEGIN_XTRA_DEFINES_CLASS(CScript, 1)
    CLASS_DEFINES_INTERFACE(CScript, IMoaMmXScript, 1)
  END_XTRA_DEFINES_CLASS

END_XTRA
```

Here you are some explanation:

the symbol “define INITGUID” declaration may only happen one time during compilation and this must happen in the first linked file of the whole project. This is because that some public XDK identifiers like IID_IMoaMmXScript for instance may only be declared once or the complier will return error. The “xtra.h” file will be the file linked first simply because it is not included elsewhere.

The “BEGIN_XTRA_DEFINES_CLASS” macro determines what class is defined and what is its “version number”. According to XDK, the xtra developer is in charge to provide such an information and this “version number” should be incremented with every new release of the xtra.
This information is useful in this sense, that in case Director finds multiple copies of an xtra within its “Xtras” directory it will bind it’s interfaces with classes of the higher version number.

The “CLASS_DEFINES_INTERFACE” macro provides an information what interface is implemented by a particular class (there may be more than one). The version number is required as well.

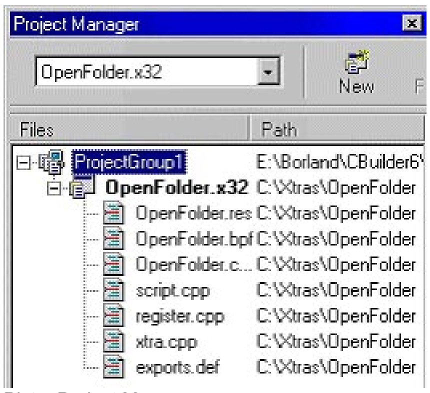
This is nearly all but we have to “export” (make global) functions providing the Director engine the access to methods of our xtra (so it can create instances of its classes):

Let’s create a simple text file (“File > New > Other > Text) and let’s save it as “exports.def”. This file should have the following content:

```
EXPORTS
  DllGetClassObject PRIVATE
  DllGetInterface
  DllGetClassForm
  DllCanUnloadNow PRIVATE
  DllGetClassInfo
```

Such section is obligatory for every xtra. Director uses these functions to get access to classes of the xtra. We have manually add "exports.def" file to our project. In Borland Builder we have to use "Project > Add to project" menu item for that purpose.

At the moment the "Project Manager" window should look like that:



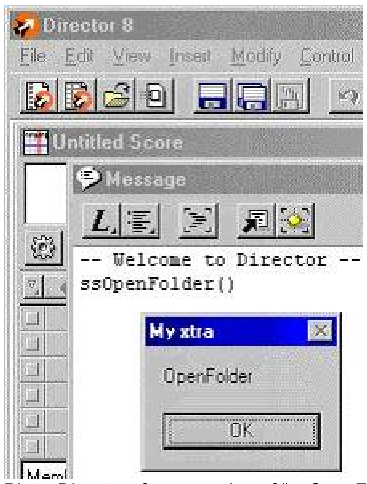
```
Files               Path
ProjectGroup1       E:\Borland\CBuilder6\
OpenFolder.x32      C:\Xtras\OpenFolder
OpenFolder.res      C:\Xtras\OpenFolder
OpenFolder.bpf      C:\Xtras\OpenFolder
OpenFolder.c...     C:\Xtras\OpenFolder
script.cpp          C:\Xtras\OpenFolder
register.cpp        C:\Xtras\OpenFolder
xtra.cpp            C:\Xtras\OpenFolder
exports.def         C:\Xtras\OpenFolder
```

Pict.4 Project Manager

Let's compile the project (Ctrl+F9). Unless we made some errors we should observe "Done: Make" message and "OpenFolder.x32" file was created in "C:\Xtras" directory.

Les's copy this file to Director's "Xtras" folder, then start Director, open its "Message" window and type there: "ssOpenFolder()".

After pressing the [Enter] key we should see the result of our hard work:



```
-- Welcome to Director --
ssOpenFolder()
```

Pict.5 Director after execution of "ssOpenFolder" command

So, happily we see IT'S WORKING!!!

But we have to say – the functionality of our xtra is very limited at the moment. This is the good time to fulfill our promise – we will make our "ssOpenFolder()" function functional. It will open the "open folder" dialog with a text parameter passed (an informative message) and it will return a path to selected directory back to Lingo.

"OpenFolder" function

We have a situation where Director, when a Lingo command "ssOpenFolder" is executed is creating an instance of our xtra and its method "OpenFolder" of "CScript" class is called. Function "OpenFolder" is practically doing nothing now. It only display a simple dialog window and – to be honest – this is done in an improper way (because a dialog is a modal window and there are specific ways to open a modal window – separate ones for Windows and Macintosh platform – look in XDK documentation for more details).

Within a minute we will enhance the functionality of the "OpenFolder" function but now let's think for a while what we are going to do:

First we have to take arguments sent when a function is called from Lingo. Let's presume our "ssOpenFolder()" function will require only one argument – a string containing text of a message displayed on "open folder" window. This text can be something like "Please, select a directory" for instance. This is a pure informative message for the end-user.

So, the OpenFolder" method of "CScript" class has to accept this argument and store it in some internal variable. Then, a standard WinApi "open folder" function has to called with the message text passed as an argument. Finally, when the "open folder" dialog is closed, the returning value containing a path to selected directory has to be passed back to Lingo.

At the beginning we have to include necessary header files to "script.cpp". These header files are required in case WinApi functions are going to be called by an xtra. We should put the following lines right after the line containing "#include script.h":

```
#define NO_WIN32_LEAN_AND_MEAN
#include <shellapi.hpp>
#include <shlobj.hpp>
```

Symbol “NO_WIN32_LEAN_AND_MEAN” is necessary for a proper compilation of “shellapi.hpp” file code.

Now we will provide you a complete code replacing:

```
void CScript::OpenFolder(PMoaMmCallInfo callPtr)
{
  MessageBox(0, "OpenFolder", "My xtra", 0);
}
```

and then we will comment it:

```
void CScript::OpenFolder(PMoaMmCallInfo callPtr)
{
  char strResult[255];
  MoaMmValue valArgument;
  PIMoaMmUtils pMmUtils;

  // here we get pointer to IMoaMmUtils interface
  pCallback->QueryInterface(&IID_IMoaMmUtils, (PPMoaVoid) &pMmUtils);

  // here we get an argument indexed as “1” (passed from Lingo)
  // and we convert it to a table of characters
  GetArgByIndex( 1, &valArgument);
  pMmUtils->ValueToString( &valArgument, strResult, 255);

  // here is a preparation necessary to API call
  MoaMmDialogCookie cookie;
  pMmUtils->WinPrepareDialogBox(&cookie);

  // here we get a handler of the “Stage” window
  MoaMmHInst hInstance;
  MoaMmHWnd hWinParent;
  pMmUtils->WinGetParent( &hInstance, &hWinParent);

  // here we prepare the “BROWSEINFO” data structure
  char displayName[MAX_PATH];
  LPITEMIDLIST retItem;
  BROWSEINFO bi;
  ZeroMemory(&bi, sizeof(bi));
  bi.hwndOwner = hWinParent;
  bi.pidlRoot = NULL;
  bi.pszDisplayName = displayName;
  bi.lpszTitle = strResult;
  bi.ulFlags = 0;
  bi.lpfn = NULL;

  // here we call the API’s “open folder” function
  retItem = SHBrowseForFolder(&bi);

  // here we initialize the returning value
  strcpy(displayName, "");

  if (retItem)
  {
    // here we get data containing a path to selected directory
    SHGetPathFromIDList(retItem, displayName);

    // here the “ITEMIDLIST” structure is released
    IMalloc *imalloc = 0;
    if ( SHGetMalloc( &imalloc ))
    {
      imalloc->Free( retItem);
      imalloc->Release();
    }
  }

  // this is necessary “cleaning” after API call
  pMmUtils->WinUnprepareDialogBox(cookie);

  // here the returning value (a path to selected directory) is passed back to Lingo
  pMmUtils->StringToValue( displayName, &callPtr->resultValue);
  pMmUtils->Release();
}
```

OK, at the beginning we declare some variables what are going to be soon in use. The character table “strResult” will be the container for argument sent from Lingo, so the text message displayed. We presume a fixed size of such a message: 255 bytes. We do not anticipate larger text to be needed.

A very important data type is “MoaMmValue”. This is a structure defined within XDK and it is used as a container for all data types (strings, integers and floating point numbers, Lingo symbols and even such data types like “point” and “rectangle”) exchanged between the Director and the xtra. One of the reasons such a structure is defined in MOA is its versatility and scalability. MOA was developed as a multiplatform technology and because of that it can not be restricted to data types only meaningful on certain platform but non existent on the other.

We will use a variable of “MoaMmValue” type called “valArgument” to store an argument passed from Lingo before converting it into a character table.

Then we have a declaration of a pointer to the “IMoaMmUtils” interface. This is a very useful interface providing multiple methods necessary during xtra development. For instance it contains methods of conversion between C++ and “MoaMmValue” data types. We have to initialize this pointer at first. We know that in case of interfaces we do not create their instances manually but we use methods of other interfaces for that purpose. The main and fundamental interface of every xtra is “IMoaCallback” interface. The pointer to this interface is created automatically by MOA an is directly accessible for the xtra at any moment. This pointer is called “pCallback” and we use it to get a pointer to the “IMoaMmUtils” interface called “pMmUtils”. We can get the result using a standard method of the “IMoaCallback” interface called “QueryInterface”.

Now it is a time to get the argument passed from Lingo. This is achieved by the “GetArgByIndex” macro (defined in MOA). It’s first argument is an index of argument passed by Lingo and the second argument is an address of a variable of “MoaMmValue” type where the argument will be stored. At this moment the variable “valArgument” will contained what was passed from Lingo.

We anticipate “valArgument” will contain a string data, so now we will use a method of the “IMoaMmUtils” interface called “ValueToString” to convert “MoaMmValue” data type into “char[]” type accepted by C++.

Then we have the following code:

```
MoaMmDialogCookie cookie;
  pMmUtils->WinPrepareDialogBox(&cookie);
```

This is necessary step before calling a WinApi function resulting with a dialog display. Simply speaking, Director needs to perform some actions before moving focus to another window as well as some actions have to be undertaken when the focus is going to be moved back to Director when the WinApi dialog is closed.

These is managed by the “IMoaMmUtils” interface and its methods: “WinPrepareDialogBox” and “WinUnprepareDialogBox”.

Method “WinPrepareDialogBox” requires two arguments. The first is the address of an object of “WinPrepareDialogCookie” type where some data required by “WinUnprepareDialogBox” are stored. Of course, before providing the address of a “cookie” object we have to declare it and its type.

If we are on the position to create a new window (a WinApi dialog), we have to provide an information about its “parent” window (according to WinApi documentation, every window has to have it’s “parent” or “owner”). We use for that the “WinGetParent” method of the “IMoaMmUtils” interface. This method returns the handler of the “Stage” window (what is a main window of the Director application) and the pointer to the main application process instance (we do not use this in our example).

Now we are ready to display the dialog window itself. We will use for that a function of WinApi called “SHBrowseForFolder” what will do the job. This function opens the “open folder” dialog and feeds it with information passed through an argument of the “BROWSEINFO” type. Please, look into WinApi documentation for more details on the subject – now we only like to say that in our example we support necessary data to the “BROWSEINFO” structure with:

- bi.hwndOwner = hWinParent; (this is a handler to the parent window)
- bi.lpszTitle = strResult; (this is a message to be displayed)

The dialog is actually opened when

- retItem = SHBrowseForFolder(&bi);

is executed.

Function “SHBrowseForFolder” returnes a full path to a selected directory, but this is “encoded” within the “ITEMIDLIST” data structure (stored within a variable called “retItem”), so we have to “extract” the information from this structure.

First we have to verify is the pointer to this structure equal “NULL” what would mean that the end-user pressed the “Cancel” button. In another case we use WinApi macro called “SHGetPathFromIDList” to convert a value stored in the “ITEMIDLIST” structure into a string.

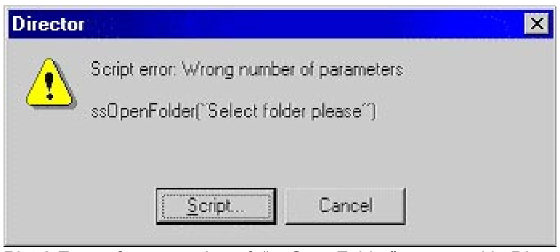
Then, according to WinApi documentation, we are on duty to release the memory used by the “ITEMIDLIST” structure. We use for that another WinApi function (“Free”), provided by a class called “SHGetMalloc”, so first we have to create a working instance of this class, and finally we have to release the memory used by this instance (“Release”).

Now we call the “WinUnprepareDialogBox” method of the “IMoaMmUtils” interface with an argument got form the “WinPrepareDialogBox” method of the same interface and we are ready to conclude with passing returning value (a path to selected directory) back to Lingo.

As we remember, we have to use an intermediate “MoaMmValue” structure when exchanging data between Director and xtra.

Fortunately, “IMoaMmCallInfor” interface support a field called “resultValue” and this field is (of course) of the “MoaMmValue” type. As we have a pointer to this interface in hand (“callPtr” parameter), all we need is to convert the value stored within “displayName” variable into “MoaMmValue” data type. For that purpose we use the “StringToValue” method of the “IMoaMmUtils” interface and finally, as the instance of this interface is not needed any more we have (obligatory!) to remove it from the memory (“pMmUtils->Release();”).

Wow! Our “OpenFolder” function is finally ready. We can compile the project (Ctrl+F9) and install the xtra in Director. But if we execute “ssOpenFolder(“Select folder please”), we will see the following error message:



Script error: Wrong number of parameters

ssOpenFolder("Select folder please")

Pict.6 Error after execution of “ssOpenFolder” command in Director

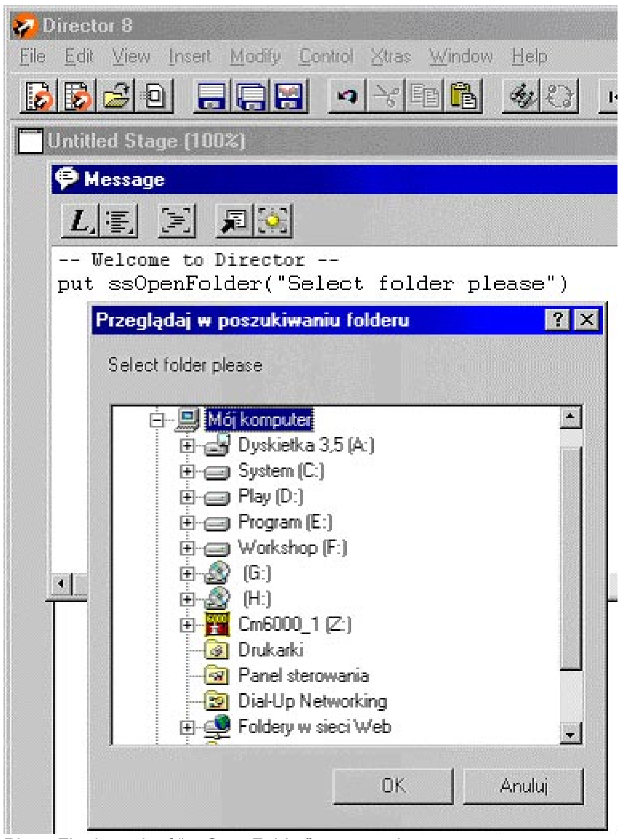
Oupps! What’s going on? It seems Director is not expecting this number of arguments being passed to xtra. That’s true – at the moment Director is not expecting any arguments at all, because our “ssOpenFolder” folder function was registered as non-argument one!

We have to go back to “register.cpp” file and look for the “msgTable” declaration. It is here where we have to provide an information about any arguments passed from Lingo to the xtra. Let’s add an entry informing Director that “ssOpenFolder” is accepting one argument of the “string” type. We can call this argument “Title” for instance:

```
static char msgTable[] = {
  "xtra OpenFolder \n"
  "new object me \n"
  "-- Lista funkcji: \n"
  "* ssOpenFolder string Title -- This function will open the open_folder dialog \n"
};
```

Let’s compile the project again and install the xtra in Director. Because new we are expecting success, let’s type in the message window: “put ssOpenFolder(“Select folder please”)”.

And here we are! We have got the expected result:



```
-- Welcome to Director --
put ssOpenFolder("Select folder please")
```

Select folder please

Pict.7 Final result of “ssOpenFolder” command

And you can notice the proper path to selected directory is returned to Lingo!

Hard to say... this is the end of our simple tutorial on xtra programming. We hope we managed to provide you a base to further development. The xtra we together managed to create is open for modifications. You can add additional arguments to “ssOpenFolder” function (like a path to starting directory), you can modify the “open folder” window itself (with WinApi functions of the “callback” type – look in WinApi documentation for details), you can add more functions to the xtra itself... this is all in your hands!

We wish you good luck with xtra programming!
/ the final “OpenFolder” xtra for PC is available here :-) /

About the authors:

Michał Ścioch is a graduate of the Technical University of Warsaw,
Department of Computer Engineering.
He is the owner of “StarSoft Multimedia” company based in Warsaw, Poland.
He is an experienced multimedia and database developer.

Zbigniew “Ziggi” Szczęsny is a graduate of the Technical University of Wrocław,
Department of Chemistry.
For many years he has been an independent multimedia developer and a specialist on Internet technologies.
He manages http://www.pm-studio.pl/xtraforum, a web-based discussion list focused on xtra development.
He lives in Warsaw, Poland.