--

Mark Wieder
mwieder@ahsoftware.net
27 August 2004

Writing Windows External DLLs for Runtime Revolution:

For some time now, my working definition of programming has been the act of getting a programming language to do something it wasn't designed to do. If there's something already built into the language, you just use it. If not, you have do some programming. If you want to print the string "hello, world" you will probably have some print function already built into the language: print(), printf(), system.println(), or whatever. If you want to display a traffic signal control on the screen and have it change when some event happens, chances are you're going to have to write some code in whatever language you're using. I think part of the process of maturing as a programmer is being able to pick the right programming tool for any given task.

Runtime Revolution <http://www.runrev.com> is a cross-platform development environment with a rich set of scripting keywords. Standalone executables can be generated from a single source that will run on Windows, Mac Classic, OSX, and linux platforms. The Transcript scripting language contains some 1700 keywords, but there are situations where you need something that you can't get to using the built-in operations. Fortunately, there's now an SDK at <http://www.runrev.com/revolution/downloads/distributions/sdk> for building external functions and commands (XFCNs and XCMDs in xTalk / Transcript language). These external routines are written in (insert your favorite language here, but we'll talk about C in this article) and communicate with the Revolution engine using a well-defined API (Application Programming Interface) that details how arguments are passed back and forth.

Note that extending the language in native code obviously breaks the cross-platform utility of a Revolution stack, but if you can compile your externals on the different platforms if the functions are supported and then you can regain your cross-platform capability by bundling the externals. This article will focus on creating DLLs for the Windows environment, but everything that is not specific to the Windows compiler or to specific operating system calls is also applicable in the Macintosh and linux environments as well. Note that in a Macintosh Classic environment the Runtime Revolution XCMD/XFCN interface is compatible with HyperCard's, so that existing resources can be used without changes, although this again limits the stack to running on only that platform.

The Revolution engine is built on the MetaCard <http://www.metacard.com> engine (MetaCard was acquired in 2003 by Runtime Revolution) and some of the nomenclature here reflects that fact. For example, the scripting engine internal binary element representation is MCstring, declared as

```
typedef MCstring
{
        char    *sptr;          // pointer to data
        int     length;         // length of data
}
```

The MCstring structure will be necessary for the GetVariableEx(), SetVariableEx(), GetArray() and SetArray() methods below.

Transcript XCMDs (external commands) and XFCNs (external functions) differ in that XFCNs return values, while XCMDs do not. From Transcript they are treated somewhat differently:

myXCMD Arg1, Arg2
put myXFCN (Arg1, Arg2) into field fldAnswer

Note that from Transcript you can check the error result of an XCMD or XFCN by examining "the result", as opposed to examining "it" which contains the result of the function call. Naturally, since externals become extensions to the Transcript language, this is true of all Transcript calls.

myXCMD field fldArg1, field fldArg2
put the result into fldError

put myXFCN (field fldArg1, field fldArg2) into fldAnswer
put the result into fldError

The functions you can use in your external library to communicate with the Revolution engine are:

> // sending messages to the engine
> **SendMCMessage** (char *, int *);
> **SendCardMessage** (char *, int *);
>
> // dealing with variables
> **char *GetGlobal** (char *, int *)
> **SetGlobal** (char *, char *, int *)
> **char *GetVariable** (char *, int *)
> **SetVariable** (char *, char *, int *)
>
> // dealing with fields
> **char *GetFieldByName** (char *, char *, int *)
> **char *GetFieldByNum** (char *, char *, int *)
> **char *GetFieldByID** (char *, char *, int *)
> **SetFieldByName** (char *, char *, char *, int *)
> **SetFieldByNum** (char *, char *, char *, int *)

SetFieldByID (char *, char *, char *, int *)

// stack objects
GetVariableEx (char *, &strKey, &MCstrValue, int *)
SetVariableEx (char *, &strKey, &MCstrValue, int *)
GetArray (char *, *int, MCstring *, char **, int *)
SetArray (char *, *int, MCstring *, char **, int *)

// images
GetImageByNum()
GetImageByID()

// status
GetMCStatus()

Number one:

Use the proper compiler. You can't use Borland C++ Builder for this. I lost a couple of days deep in the debugger trying to figure out why I couldn't get this to work, and the answer turned out to be "it just can't be done". The Metacard / Revolution engine uses the Microsoft OBJ convention for Windows DLLs. Here's the convention you need to support:

1. arguments passed on the stack, not in registers
2. no prepended underscores in function names
3. function names are mixed case
4. calling, not called, function cleans up the stack

Borland's compiler switches let you select combinations of these factors, but there is no combination of switches that will let you select *all* the necessary ones at the same time. In particular, you can't turn off prepended underscores unless either a) you use the __pascal option, which makes all function names uppercase; or b) you explicitly specify "turn off underscores" in the project options, which means you have to change all your standard library calls to include underscores.

Number two:

Make sure the getXtable function is exported properly:

```
extern "C"
{
        void getXtable()
}
```

There are two header files you'll need to #include, **XCmdGlue.h** and **external.h**. XCmdGlue.h contains prototype and other useful things to glue your code to the Revolution engine. It probably shouldn't need to be modified. External.h contains the stuff you need to handle your own XCMDs and XFCNs.

--

The Metacard/Revolution engine needs a single exported function "getXtable" (note the mixed case here - that's important). The getXtable function returns a pointer to an array of Xternal structures which specify the XCMDs and XFCNs that the DLL supports. The Xternal structure is described in the sample XcmdGlue.h, but just to reiterate:

```
/*
 * This global table has one entry per XCMD/XFCN.
 * The first entry is the name of the handler
 * The second entry is the type (XCOMMAND or XFUNCTION)
 * The third entry is a space for the atom (used by MetaCard)
 * The fourth entry is the name of the 'C' function to call
 * The fifth entry is a callback called if the user aborts
 * Note that the last entry in the table is a NULL entry
 * which is used to measure the size of the table.
 */
```

Here's a simple Xtable with one XCMD and one XFCN:

```
Xternal Xtable[] = {
        {"revXCMD", XCOMMAND, 0, _MyXCMD, XCabort},
        {"revXFCN", XFUNCTION, 0, _MyXFCN, XCabort},
        {"", XNONE, 0, NULL, NULL}
};
```

In this example revXCMD and revXFCN are the commands you would use from Transcript code:

revXCMD "http://runrev.com"
put revXFCN (3.1416) into card field result

and _MyXCMD and _MyXFCN are the internal names of routines in the external library.

Each XFCN or XCMD receives the same set of arguments:

a pointer to an array of null-terminated strings,
the number of arguments passed,
a pointer to a null-terminated return string for Transcript,
a pointer to a boolean indicating whether to pass the handler up the chain,
a pointer to a boolean error flag.

```
void _MyXCMD (char *args[], int nargs, char **retstring, Bool *pass, Bool *error);
void _MyXFCN (char *args[], int nargs, char **retstring, Bool *pass, Bool *error);
```

Note that all arguments passed to and from the Transcript engine are C-style null-terminated character strings. They are internally converted by the engine when necessary, but you'll have to do your own conversion if you need to, either using the **EvalExpr()** function or some other conversion on your own.

Each XFCN is expected to return a string. Use the istrdup() function for this:

```
        *retstring = istrdup ("here is your result");
```

XCMDs should return a retstring indicating no error:

```
        *retstring = (char *) calloc(1,1);     // indicate no errors
```

Notes:

The pass flag should always return a false. Returning a True for the pass flag is a signal to the engine that the external did not handle the command and it should continue the search up the command hierarchy for an appropriate handler. If no other handler by that name is found the script will stop executing with a runtime error.

If you return a true in the error flag script execution will stop. If you do this it's good form to return a return string specifying the error.

```
        *retstring = istrdup ("error: wrong number of arguments");
```

```
void _MyXCMD(char *args[], int nargs, char **retstring, Bool *pass, Bool *error)
{
        *pass = False;          // we handled the command
        *error = False;         // indicate no errors
        *retstring = (char *) calloc(1,1);     // indicate no errors
}

void _MyXFCN(char *args[], int nargs, char **retstring, Bool *pass, Bool *error)
{
        *pass = False;          // we handled the command
        if (1 != nargs)
        {
                *error = True;          // stop script processing here
                *retstring = istrdup ("wrong number of arguments");
        }
        else
```

```
	{
		*error = False;		// indicate no errors
		*retstring = istrdup ("here is your result");
	}
}
```

If you need to send a message to the current card you can use the **SendCardMessage**() function:

Example:

```
void reverse(char *args[], int nargs, char **retstring, Bool *pass, Bool *error)
{
	int	retValue;
	int	x, y, length;
	char buffer[256];

	length = strlen(args[0]);
	for (x=length, y=0; x>0; x--, y++)
		buffer[y] = args[0][x];
	buffer[length] = 0;
	SendCardMessage (buffer, &retValue);
	*pass = False;		// we handled the command
	*error = False;		// indicate no errors
	*retstring = istrdup ("here is your result");
}
```

from Transcript:

reverse ("Magrat")

will invoke the **on myHandler** routine in the stack, using "Magrat" as an argument, internally generating the buffer string "myHandler Magrat", so that if you have a **myHandler** function in your stack

```
on myHandler theString
	put theString into field fldTest
end myHandler
```

you would end up with "Magrat" in field fldTest.

You can also tell the Metacard/Revolution engine to execute a Transcript command line using the **SendMCMessage**() function:

Example:

```
Void XMessageTime(char *args[], int nargs, char **retstring, Bool *pass, Bool *error)
{
	int	retValue;
	char buffer[256];

	sprintf (buffer, "put  \" Message time is %d\" into field %s, (int)time(NULL),
args[0]);
	SendMCMessage (buffer, &retValue);
}
```

from Transcript:

XMessageTime (fldResult)

will put the current time into card field fldResult

--

Getting and setting variables:

The following routines are available. The GetGlobal and SetGlobal functions operate on variables that have been declared using the "global" Transcript keyword. The GetVariable and SetVariable functions operate on local variables:

char *GetGlobal (strFieldName, &intRetValue)
SetGlobal (strFieldName, strValue, &intRetValue)

char *GetVariable (strFieldName, &intRetValue)
SetVariable (strFieldName, strValue, &intRetValue)

Note that the GetGlobal and GetVariable routines are responsible for freeing up the memory allocated automatically by the routines.

--

Getting and setting fields:

The first argument to the field routines (strWhere) should be a null-terminated string specifying where to look for the field:

"true" to look for the field only on the card
"false" to look only on the background

"" to look on both the card and the background
or a group name to look for the field in that group.

The following routines are available:

char *GetFieldByName (strWhere, strFieldName, &intRetValue)
char *GetFieldByNum (strWhere, strFieldNum, &intRetValue)
char *GetFieldByID (strWhere, strFieldID, &intRetValue)

SetFieldByName (strWhere, strFieldName, strValue, &intRetValue)
SetFieldByNum (strWhere, strFieldNum, strValue, &intRetValue)
SetFieldByID (strWhere, strFieldID, strValue, &intRetValue)

Example:

The following routine copies a field contents from a field on the background (fldBackground) into a field on the current card (fldOnCard):

from Transcript:

XCopyFromBackground (fldSource, fldDestination)

```
Void XCopyFromBackground(char *args[], int nargs, char **retstring, Bool *pass, Bool
*error)
{
	int	intRetValue;
	char	*strData;

	// get information from source field in background
	strData = GetFieldByName ("false", args[0] & intRetValue);
	// put the retrieved information into destination field on card
	SetFieldByName ("true", args[1], strData, &intRetValue;
	// GetFieldByName allocated memory, so we need to free it.
	free (strData);
}
```

Note that the GetFieldByXXX routines are responsible for freeing up the memory allocated automatically by the routines in order to avoid memory leaks.

--

Getting and setting stack objects:

The following routines are available. The GetVariableEx and SetVariableEx functions work with binary stack objects of variable sizes. GetArray and SetArray work with arrays of stack objects. The MCstring object is the internal scripting engine representation for binary objects. It is declared as:

```
typedef MCstring
{
	char	*sptr;		// pointer to data
	int	length;		// length of data
}
```

GetVariableEx (strObjectName, &strKey, &MCstrValue, &intRetValue)
SetVariableEx (strObjectName, &strKey, &MCstrValue, &intRetValue)
GetArray (strObjectName, intNumElements, MCstrValues, MCstrKeys, &intRetValue)
SetArray (strObjectName, intNumElements, MCstrValues, MCstrKeys, &intRetValue)

Use the GetVariableEx function to retrieve a Revolution stack binary object, for example a.jpg image. A pointer to the object is returned in an MCstring structure. The MCstring.sptr member points to the object itself. Use the MCstring.length member to determine the size of the data.

```
	MCstring	mcData;
	int		intRetValue;
	char		*image;

	GetVariableEx ("myPicture", "", &mcData, &intRetValue);
	if (NULL != mcData.sptr)
	{
		// now you can manipulate the data...
	}
	SetVariableEx ("myPicture", "", &mcData, &intRetValue);
```

Note that GetVariableEx() retrieves raw data. If you need to retrieve strings from the stack, they will not be null-terminated. You will need to add the null termination yourself if necessary.

Arrays of stack objects:

Arrays in Transcript are associative and identified by an ArrayName and a Key, as in

```
put "A" into myList["firstLetter"]
get myList["secondLetter"]
put "Ringo" into band["drummer"]
put "Monday" into days[2]
put "Tuesday" into days[3]
put "Wednesday" into days[4]
```

Pass a NULL as the MCstrKeys argument to GetArray() to find the size of the data to be retrieved. Then you can allocate enough space for the data.

```
    int         intNumElements;
    MCstring    *MCstrValues = NULL;
    char        **strKeys = NULL;
    int         intRetValue;

    GetArray(args[0], &intNumElements, MCstrValues, strKeys, &intRetValue);
    // since strKeys is NULL, this will not retrieve data,
    // but will instead fill intNumElements and MCstrValues for you.

    // now that you know the size of the data you can allocate space for it.
    if (0 != intNumElements)
    {
        MCstrValues = malloc (sizeof (MCstring) * intNumElements);
        strKeys = malloc (sizeof (char *) * intNumElements);

        if ((NULL != MCstrValues) & (NULL != strKeys))
            {
            // and get the data
            GetArray(args[0], &intNumElements, MCstrValues, strKeys,
&intRetValue);

            // now we can work with our array

            // and save it back to the stack
            SetArray(args[0], &intNumElements, MCstrValues, strKeys,
&intRetValue);

            // free up allocated memory when we're done
            free (strKeys);
            free (MCstrValues);
        }
        else
        {
            fprintf(stderr, "oops - array doesn't exist");
        }
    }
```

Evaluate the given expression and return the result as a string:

char ***EvalExpr** (char *strExpression, int &intRetValue)

Example:

from Transcript (to update the global gameCounter by 1):

```
global gameCounter
UpdateHitCount "gameCounter"
```

In the external library:

```
void UpdateHitCount (char *args[], int nargs, char **retstring, Bool *pass, Bool *error)
{
	int	intRetValue;
	char	*strData, *buffer;

	// read the global variable from the stack
	strData = GetGlobal (args[0], &intRetValue);

	// prepare to add one to the variable
	sprintf (buffer, "%s + 1", strData);
	free (strData);		// free the memory allocated by GetGlobal

	// evaluate the expression in the buffer
	//	(which now contains "variable + 1")
	strData = EvalExpr (buffer, &intRetValue);

	// write the global variable back to the stack
	SetGlobal (args[0], strData, &intRetValue);
	free (strData);		// free the memory allocated by EvalExpr
}
```

Note that the EvalExpr routine is responsible for freeing up the memory allocated automatically by the routine in order to avoid memory leaks.

An Example

OK. Enough with the theoretical stuff. Let's put this into action. Let's say we want to retrieve the computer name for robust error reporting. This functionality isn't built into the engine, so we'll need to extend the language to make a call out to the Win32 API. We'll do this by creating a new DLL project, including the XCMDGlue.h header file, and creating a source and header file of our own. We'll need to call the GetComputerName() function declared in winbase.h.

We'll use this function from a stack as in

```
put xgetcomputername() into card field fldComputerName
```

Looking at the API, we see that this function takes two parameters: one a long pointer to a buffer for the name itself, and the other a long pointer to a DWORD that contains the length of the buffer and contains the length of the name after the function returns. The function itself returns a BOOL informing the user as the the success or failure of the call.

```
#include "XCMDGlue.h"
#include "winbase.h"         // GetComputerName(), SetComputerName()

void XGetName(char *args[], int nargs, char **retstring, Bool *pass, Bool *error)
{
        char            buffer[MAX_COMPUTERNAME_LENGTH+1];
        DWORD       dwLength;
        BOOL          blnSuccess;

        dwLength = MAX_COMPUTERNAME_LENGTH+1;
        blnSuccess = GetComputerName(buffer, &dwLength);
        *pass = False;          // don't pass the command up the chain
        if (blnSuccess)
        {
                *error = False;         // call was successful
                *retstring = istrdup (buffer);
        }
        else
        {
                *error = True;           // call failed
                *retstring = istrdup ("error: GetComputerName failed");
        }
}
```

Just for fun while we're here let's make a function to set the computer name as well. We'll need one argument this time, the new name for the computer. We'll pass this from Transcript as

xsetcomputername "weatherwax"

and the next time we reboot the computer it will have a new name on the network. To do this we'll need to get the first passed argument out of the argument array. There's only one argument we need for this call. We could just ignore any extra arguments, but we want to make sure that at least one argument was passed, so let's make sure nargs is exactly 1.

```
void XSetName(char *args[], int nargs, char **retstring, Bool *pass, Bool *error)
{
        char            buffer[MAX_COMPUTERNAME_LENGTH+1];
        BOOL          blnSuccess = False;
```

```
	// get the new name from the array of arguments
	if (1 == nargs)
	{
		blnSuccess = SetComputerName(args[0]);
	}

	*pass = False;		// don't pass the command up the chain
	if (blnSuccess)
	{
		*error = False;		// call was successful
		*retstring = istrdup (buffer);
	}
	else
	{
		*error = True;		// call failed
		*retstring = istrdup ("error: SetComputerName failed");
	}
}
```

Now we're ready to set up our Xtable structure:

```
Xternal Xtable[] = {
	{"xsetcomputername", XCOMMAND, 0, XSetName, XCabort},
	{"xgetcomputername", XFUNCTION, 0, XGetName, XCabort},
	{"", XNONE, 0, NULL, NULL}
};
```

Here **xsetcomputername** and **xgetcomputername** are the Transcript keywords we'll use in our stacks and **XSetName** and **XGetName** are the names of the functions we just created. Now compile your DLL project and you've got two new functions available.

Including and Testing

Testing external libraries is always tricky. If your library has compiled properly you can test it from Runtime Revolution in the following way:

From Revolution the Transcript command:

set the externals of this stack to "C:\MyDirectory\MyDll.dll"

will tell your stack to look to your library for extensions to the Transcript language. However, this won't actually take effect until you either

save your stack, close it, and open it again, or quit Revolution and launch it again.

Once you've done that, the Transcript code:

```
put the externalFunctions
        or
put the externalCommands
```

will show you what's currently available. If you see the list of commands you've compiled then your linkage with the getXtable function is working properly.

To save typing, I use a button handler script to set the externals:

```
on mouseUp
    ask file "Where is the external library?"
    if it is not empty then
        set the externals of this stack to it
    end if
end mouseUp
```

and another to view them:

```
on mouseUp
    put "Commands" & cr into field fldList
    put the externalCommands of this stack after field fldList
    put cr & cr & "Functions" & cr after field fldList
    put the externalFunctions of this stack after field fldList
end mouseUp
```

Once your stack knows about the external routines compiled into a library you can specify bundling the library with the stack in the Revolution Distribution Builder so that you can create a single standalone application without needing additional libraries (and without needing an installer to ensure that the libraries get put into the proper place).

Runtime Revolution offers a powerful and flexible development environment with the Transcript scripting language, and with the ability to extend the language by writing external commands and functions there are virtually no limits to what you can do in a stack.