Dealing with Dlls in AutoIt

Table of Contents

Introduction

Welcome to this tutorial about using the various Dll* functions in AutoIt. These functions are very powerful and are one of AutoIt's greatest features since it allows you to do almost anything in AutoIt. Some of the things that can be done are direct access to the Windows API, 3D programming, direct memory access and much more.

This tutorial is meant to give you the basic knowledge and understanding to write function that uses Dll* function and will most likely give you a much better understanding of programming in general.

Each chapter is divided in 3 parts, theory, example code & exercises.

All of the coding in this tutorial is using standard windows dll's and information from Microsoft's documentation site msdn.

Good Luck!

Data Types

First of all, if you have any experience of any language that uses specific types for data you can skip this chapter completely.

So what is a data type? A data type is a specific format that data is stored in. In AutoIt there is only one data type, the variant. The variant can store all kind of stuff, numbers, strings, handles etc. This is not the case however in most languages, and you have to specify what you're going to store in the variable.

For example in C/C++ you use the 'int' keyword to create a variable that can store an integer and not only that, it is limited to hold only four bytes as well. 'float' is also a widely used type as it is the standard way to create floating point numbers.

Strings however is not that easy in type specific languages. First of all there is a type for **one** character, it is simply named char (or wchar_t when dealing with Unicode). But since it is only made for one char it's not possible to store entire strings in them. So what's the solution? To create an array of chars and put a NULL char (ASCII 0) in the last element to specify that the string ends there. As you can imagine it's not very pleasant to work with this kind of stuff, and that's one of the best reasons to use a type-less language (like AutoIt) to work with strings. However when dealing with dlls, this is something we must have in mind.

For more information on what types AutoIt supports and what C/C++ supports check the AutoIt help file under DllCall() and this link:

http://www.cplusplus.com/doc/tutorial/variables.html

That's it for now. Don't worry about it if you don't get it. It won't be terrible important to understand this in details until you begin working with advanced structs.

Time for the first DllCall()!

My First DllCall()

The first function we’re going to look at is the Sleep() function, which resides in Kernel32.dll. This is the exact same function that is being called when you write Sleep(100) in AutoIt, it’s just that AutoIt acts a layer between you and the function. Here’s the documentation of the function: http://msdn.microsoft.com/en-us/library/ms686298(VS.85).aspx

As you could see when reading the documentation Sleep has got 1 parameter and does not return any value. Now, let’s check the syntax of the AutoIt function DllCall():

DllCall ("dll", "return type", "function" [, "type1", param1 [, "type n", param n]])

Let’s step through all the parameters, one at a time. First we have “dll”, this is the dll file that the function is in. As I said before and as the documentation specified the dll is named “Kernel32.dll”, just put that in there.

Second is the return type. MSDN said that there was no return so put in “none” there (AutoIt version of void). Next shouldn’t be too hard, we know the name of the function, that’s right, Sleep. Just put it in there as well.

Moving on to the parameter. As you can see the first thing you need to specify is the type of the parameter. This is called “DWORD” on the msdn page and is in AutoIt called simply “dword”. So put in “dword” in the “type 1” parameter. The parameter’s type is specified so in “param1” you can specify what value you want it to sleep here.

There should be no more parameters in the DllCall() now, if the Sleep function had more parameters you should repeat the last time x amount of times. DllCall is so cool that it has optional amount of parameters.

If there would have been any returned data it would be in an array, so the data in the first element ([0]) would have been the data returned from the dll.

Hopefully you got some working code by now, but don’t despair if you don’t here’s the working version!

```
; Sleeps for 1 second
DllCall("Kernel32.dll","none","Sleep","dword",1000)
```

Hopefully you got it now!

Before finish this lesson you should make sure you can implement these functions by yourself!

- LockWorkStation – Tip! BOOL is an int and when there are no parameters you completely skip the parameter section of DllCall.

- GetCurrentProcessId – Tip! The returned data is returned as an array where array[0] is the returned data

Strings in DllCall()

As you should know strings in dlls are in fact arrays of chars, fortunately AutoIt helps us with passing this so all you have to do is to choose “str” or “wstr” and then just put in your string in the paramx param in DllCall(). A very important thing to remember here is that since windows was not fully Unicode until win2000 almost all function that deals with strings is implemented in two versions. They usually have an ‘A’ or a ‘W’ in the end of its name, like this “MyStringFunctionA”, where A stands for ANSI and W stands for wide char, aka Unicode.

Now let’s look at this lesson’s Dll function: MessageBox, yes its good ol’ MessageBox that is also included in autoit.

As you can see there are different flags that you can specify here and unfortunate msdn only specifies the symbol name of them, which means that the different names you see there actually is actually a number. To find this value try Google the name or look it up in some version of the winapi language extensions (like windows.h for C/C++).

For this time being we use 0 for “uType”.

Here’s the source code for this lesson. Study it and change stuff in it. For example try changing uType to 16.

```
; We call MessageBoxW because we want unicode, which means we have to use wstr
; hwnd is 0 because we don't have a window to be the owner of the messagebox
DllCall("user32.dll","int","MessageBoxW","hwnd",0,"wstr","Hello from Dll
tutorial!","wstr","Info","uint",0)
```

That’s it! Study it and don’t move on until you finish these exercises:

- Change the code to use the ASCII version instead
- Implement FindWindow

Passing a parameter as a reference

If have ever seen the AutoIt keyword “ByRef” in a function declaration you will know how this works. Basically it means that you send a value to the dll and the dll modifies it for you. This is useful if the dll wants to signalize its success through the return value but still give back values to the caller.

A sample function from windows that uses it is GetDiskFreeSpace.

To pass a parameter as reference, just add a ‘*’ to the parameters type.

Here’s how the function is implemented:

```
; Variables to pass as reference
Local $SectorsPerCluster
Local $BytesPerSector
Local $NumberOfFreeClusters
Local $TotalNumberOfClusters

$calldata=DllCall("Kernel32.dll","int","GetDiskFreeSpaceW","wstr","C:\","dword*",$SectorsPerCluster,"dword*",$BytesPerSector,"dword*",$NumberOfFreeClusters,"dword*",$TotalNumberOfClusters)

; The data is returned as an array, even the changed values of the variables
$SectorsPerCluster=$calldata[2]
$BytesPerSector=$calldata[3]
$NumberOfFreeClusters=$calldata[4]
$TotalNumberOfClusters=$calldata[5]


MsgBox(0,"","Total number of clusters: "&$TotalNumberOfClusters)
```

As you can see all data is in the array, and the variables are not actually modified. You will have to overwrite their values to the ones in the array if you want the actual values changed.

That’s all for this chapter, I couldn’t find any appropriate exercises so make sure the above example is clear as water.

A word about calling conventions

A final word when it comes to DllCall is calling conventions. A calling convention is a set of rules about how the calling of a function should be done. All windows dlls uses the stdcall calling convention and it is also the standard mode in autoit. However a calling convention called “cdecl” is pretty poular as well and switching to it in autoit is simple. Just put :cdecl after the return type. So:

```
; With stdcall
DllCall(”SomeDll.dll”,”int”,”Func”)
```

Becomes:

```
; With cdecl
DllCall(”SomeDll.dll”,”int:cdecl”,”Func”)
```

DllStructs – Hell yeah!

It's now time to start with the really important and useful stuff, DllStructs, so what is a DllStruct? Try to think of them as packets of data where all the variables lies next to eachother in memory.

So why do we need them? Couldn't we just pass all the data as parameters to the function? In some situations, yes, we could. But in most situations this is just stupid and sometimes the size of the data could vary and then it would create unnecessary long and complicated code. It's also necessary when the function want to return many values, in this case it could return a pointer (memory address) to a struct that contains lots of variables (or members as I will refer them as from now on). Another way would be to pass a pointer to a struct as a parameter to the function and let it modify the struct that it points to.

Another good thing about is that it allows direct memory access in AutoIt. This is not terrible useful, but when you work with advanced low-level (relative lov-level) code you'll see that AutoIt would be crippled without it.

As a first example we're going to take a look at the GetSystemTime function. As you can see the function is very straightforward, just one parameter. However that parameter is a pointer to a SYSTEMTIME struct. To use this struct you will have to translate it to an AutoIt DllStruct, as you can see if you follow the link to the documentation for SYSTEMTIME, the struct look like this:

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

The first thing you'll probably notice is the type WORD, this type is not specified in AutoIt so you will have find out what Microsoft means with WORD, a quick googling for "WORD typedef" turns out that WORD is actually an unsigned 16 bit integer aka "ushort" in AutoIt (check the help file entry for DllCall). So the complete translation of the struct (and the simple call) is as follow:

```
$SYSTEMTIME=DllStructCreate("ushort wYear;ushort wMonth;ushort wDayOfWeek;ushort
wDay;ushort wHour;ushort wMinute;ushort wSecond;ushort wMilliseconds")
DllCall("Kernel32.dll","none","GetSystemTime","ptr",DllStructGetPtr($SYSTEMTIME))
MsgBox(0,"Current
time:",DllStructGetData($SYSTEMTIME,"wHour")&":"&DllStructGetData($SYSTEMTIME,"wMinut
e"))
```

Puh, lots of new info there, try playing around with it and when you feel like you understand it try doing this exercise:

- Change the clock of the local system to 6 pm tomorrow and then set it back to the old time using the SetLocalTime function.