A paper on:
Excel and C++ DLLs

Fredrik Gerdin Börjesson
(Dated: 2020–04–15)

A shorter version of the Microsoft Excel documentation and specifically the usage of DLLs and C++. Not intended to be fact-checked or necessarily fully accurate but to give a brief introduction to the area.

CONTENTS

I. This Paper 1
A. Collection of links 1

II. Getting started 1

III. Data types used by Excel 1
A. Volatile user-defined functions 2
B. Calculation modes 2

IV. Working with DLLs 2
A. Exporting functions and commands 2
1. Using a DEF file 3
2. Using the __declspec(dllexport) declarator 3
3. Using a #pragma preprocessor linker directive 3

V. Accessing DLLs in Excel 3
A. Argument types in C++ and VBA 4
B. Calling DLL functions directly from worksheet 4
C. Calling DLL commands directly from Excel 4
D. DLL memory and multiple DLL instances 4
E. Calling user-defined functions from DLLs 5

VI. Accessing XLL code in Excel 5

VII. Calling into Excel from the DLL or XLL 5
A. Excel4, Excel4v, Excel12, and Excel12v Functions 5

VIII. Creating XLLs 5
A. Enabling DLLs to Call Back into Excel 5

IX. Excel performance: Tips for optimizing performance obstructions 6
A. Optimize references and links 6
B. Minimize the used range 6
C. Allow for extra data – properly 6
1. Use structured table references (recommended) 6
2. Alternatively, use whole column and row references 6
3. Alternatively, use dynamic ranges 6
D. Improve lookup calculation time 6
1. Use INDEX and MATCH or OFFSET instead of VLOOKUP 6
2. Use two lookups for sorted data with missing values 7
E. Create faster VBA macros 7
1. Turn off everything but the essentials while code is running 7
2. Read and write large blocks of data in a single operation 7
3. Additional VBA performance optimizations 7
F. Additional considerations 7

I. THIS PAPER

This paper is a written summary/technical document over the Excel documentation found here.

A. Collection of links

The following list has some of the more important or interesting links that may be useful in the development.

- Excel VBA reference
- Excel performance: Performance and limit improvements
- Excel performance: Tips for optimizing performance obstructions
- Welcome to the Excel Software Development Kit

II. GETTING STARTED

There are several options of interfaces provided for use with Excel. Two such are:

- C API and XLLs: "DLLs that are integrated with Excel. These DLLs provide the most direct and fastest interface for the addition of high-performance worksheet functions, although at the cost of some complexity compared with later technologies."
- VBA: "Visual Basic code objects that are associated with Excel workbook objects. VBA allows event trapping, customization, and the addition of user-defined functions and commands. VBA is the most commonly used and most easily available of the extensibility options."

The primary reason for writing XLLs and using the C API is to create high-performance worksheet functions.

III. DATA TYPES USED BY EXCEL

The following are a list of ANSI C/C++ data types in Excel.
Numbers:

- 8-byte double

- [signed] short [int] – used for Boolean values and integers
- unsigned short [int]
- [signed long] int

Strings:

- [signed] char* – null-terminated byte strings of up to 255 chars
- unsigned char* – length-counted byte strings of up to 255 chars
- unsigned short* – Unicode strings of up to 32,767 chars, null-terminated or length-counted

Note: "All worksheet numbers in Excel are stored as doubles so that it is not necessary (and in fact introduces a small conversion overhead) to declare add-in functions as exchanging integer types with Excel."

A. Volatile user-defined functions

A volatile user-defined function (UDF) is a function "that causes recalculation of the formula in the cell where it resides every time Excel recalculates." As such, they are to be avoided whenever so is possible. The following set of functions are volatile: `NOW`, `TODAY`, `RANDBETWEEN`, `OFFSET`, `INDIRECT`, `INFO` (depending on its arguments), `CELL` (depending on its arguments), `SUMIF` (depending on its arguments).

When writing an UDF, the function may be declared volatile using `Application.Volatile (MyBool)` where `MyBool` toggles the volatility with either FALSE or TRUE (i.e. 0 or 1).

B. Calculation modes

Excel has three calculation modes:

- Automatic,
- Automatic Except Tables,
- Manual.

The mode may be changed from within VBA to prevent updates mid-macro, which is beneficial for large workbooks.

In VBA, `Range.Calculate` may be used to force the recalculation of a specific Range, regardless of the cells being tagged dirty or not. An alternative is `Range.CalculateRowMajorOrder`, calculating the range cell-by-cell from top-left to bottom-right in row-major order.

The shortcut `Shift + F9` forces calculation of the active Worksheet. Similarly, `Ctrl + Alt + Shift + F9` forces calculation of the full book, and `F9` forces calculation of all open workbooks.

IV. WORKING WITH DLLS

"A library is a body of compiled code that provides some functionality and data to an executable application. Libraries can be either statically linked or dynamically linked, and they conventionally have the file name extensions .lib and .dll respectively. Static libraries (such as the C run-time library) are linked to the application at compilation and so become part of the resulting executable. The application loads a DLL when it is needed, usually when the application starts up. One DLL can load and dynamically link to another DLL."

We use the DLLs to add functionality to our worksheets and can often improve speed and reduce file size by implementing them.

The following are required to create a DLL:

1. A source code editor
2. A compiler to turn source code into object code compatible with our hardware
3. A linker to add code from statuc libraries and to create the executable DLL file

For this purpose, we use Microsoft Visual Studio (2019), which supports all of the above using the Microsoft Visual C++ (MSVC) compiler.

A. Exporting functions and commands

"When compiling a DLL project, the compiler and linker need to know what functions are to be exported so that they can make them available to the application." When exporting a function, we give it a chosen name to make it recognizable by the application loading the DLL. This is called "name decoration." The name may either be the same as the original name in the source code, or something different.

This decoration depends on the language and the compiler's instructions, where the standard inter-process calling convention for Windows used by DLLs is known as the WinAPI convention. This is defined in Windows header files as `WINAPI`, being defined using the Win32 declarator `__stdcall`. The DLL-export function to be used by Excel should always use one of these two calling conventions. Otherwise, the default `__cdecl` convention (`INAPIV`) is used.

The linker may be told that a function is to be exported by one of the three following methods.

1. Place the function in a DEF file after the `EXPORTS` keyword, and set your DLL project setting to reference this file when linking.
2. Use the `__declspec(dllexport)` declarator in the function's definition.
3. Use a `#pragma` preprocessor directive to send a message to the linker.

1. Using a DEF file

See Listing 1 for an example of an exported function with a DEF file.

Listing 1. C++ example of when using DEF file

```
double WINAPI my_export(double x)
{
    // Modify x and return it.
    return x * 2.0;
}
```

Then, the DEF file would contain the following line:

```
EXPORTS my_export = _my_export@8 my_export
```

The general version of this syntax is:

```
entryname[=internalname] [@ordinal[NONAME]]
[DATA] [PRIVATE]
```

"For 32-bit Windows API function calls, the convention for the decoration of C-compiled functions is as follows: `function_name` becomes `_function_name@n` where `n` is the number of bytes expressed as a decimal taken up by all the arguments, with the bytes for each rounded up to the nearest multiple of four."

Note: "All pointers are four bytes wide in Win32. The return type has no impact on name decoration."

We can also force the C++ compiler to expose undecorated names of C++ functions by preceding the function implementation with `extern "C"` in Listing 1.

2. Using the `__declspec(dllexport)` declarator

See the code example in Listing 2.

Listing 2. C++ example of when using `__declspec(dllexport)`

```
__declspec(dllexport) double WINAPI
    my_export(double x)
{
    // Modify x and return it.
    return x * 2.0;
}
```

Alternatively, to avoid the C++ function being made available with the C++ decoration, declare it with `extern "C"`, such as in Listing 3.

Listing 3. C++ example of when using `__declspec(dllexport)` and extern/undecorated function

```
extern "C"
__declspec(dllexport) double WINAPI
    my_undecorated_export(double x)
{
    // Modify x and return it.
    return x * 2.0;
}
```

Note: The `__declspec(dllexport)` declaration must be placed **furthest left** in the function declaration.

3. Using a #pragma preprocessor linker directive

In later Microsoft Visual Studio versions, there are two predefined macros that, when used with a `#pragma` directing, allow for instructing the linker to export a function directly from within the function code. These macros are `FUNCTION` and `FUNCDNAME`.

These lines can be incorporated into a common header file as follows.

Listing 4. Implementing the predefined macros in C++

```
#if _MSC_VER > 1200 // Later than VS
    6.0
#define EXPORT comment(linker, "/EXPORT
    :"__FUNCTION__"="__FUNCDNAME__)
#else // Cannot use this way of
    exporting functions.
#define EXPORT
#endif // else need to use DEF file or
    __declspec(dllexport)
```

We may then implement our C++ function as in Listing 5, without having to use a DEF file or `__declspec(dllexport)`.

Listing 5. C++ example of exporting function with `#pragma`

```
double WINAPI my_export(double x)
{
    #pragma EXPORT
    // Modify x and return it.
    return x * 2.0;
}
```

V. ACCESSING DLLS IN EXCEL

Two possible ways of accessing a DLL function or command in Microsoft Excel are:

- Through a VBA code module in which the DLL function or command is made available by a `Declare` statement
- Directly from the worksheet or a customized item in the user interface

A command is accessed as follows,

```
[Public | Private] Declare Sub name Lib
    "libname" [Alias "aliasname"] [([
    arglist])]
```

and a function is accessed by

```
[Public | Private] Declare Function
    name Lib "libname" [Alias "
    aliasname"] [([arglist])] [As type]
```

The keywords `Public` and `Private` specify whether the function is available to the entire VB project or only the VB module itself. The name is the name to be used for the function or command in Excel, whereas the `Alias` may be used if the desired name differs from that of the exported function (in which `"aliasname"` is set to the exported function name).

Commands return void while functions should return data types that may be recognized by VBA's `ByVal`. Thus, some data types such as strings, arrays, user-defined types, and objects may be more easily returned by modifying them in place.

The function or command arguments must be superseded by `ByVal` unless passed by reference or pointer. If the arguments are received in the C++ function by reference or pointer, the argument should be superseded by `ByRef`, although it is the default and may therefore be omitted in VBA.

A. Argument types in C++ and VBA

Table I presents common VBA data types and their equivalent types in C++.

The following is an example of a VBA user-type definition.

Listing 6. VBA example of `Type` definition

```
Type VB_User_Type
    i As Integer
    d As Double
    s As String
End Type
```

The C++ equivalent `struct` is defined as follows.

Listing 7. C++ equivalent of the above user-type definition

```
#pragma pack(4) // See \textbf{Note}
struct C_user_type
{
    short iVal;
    double dVal;
    BSTR bstr; // VBA String type is a
       byte string
}
#pragma pack() // Restore default
```

Note: The first `#pragma pack (4)` is required since VBA by default packs user-defined data types to 4-byte boundaries whilst in Visual Studio they are packed to 8-byte boundaries. The latter pack statement restores the default in Visual Studio.

Variant and string arguments

The next paragraphs quote this section of the documentation.

> "Excel works internally with wide-character Unicode strings. When a VBA user-defined function is declared as taking a String argument, Excel converts the supplied string to a byte-string in a locale-specific way. If you want your function to be passed a Unicode string, your VBA user-defined function should accept a Variant instead of a String argument. Your DLL function can then accept that Variant BSTR wide-character string from VBA.
>
> To return Unicode strings to VBA from a DLL, you should modify a Variant string argument in place. For this to work, you must declare the DLL function as taking a pointer to the Variant and in your C/C++ code, and declare the argument in the VBA code as ByRef varg As Variant. The old string memory should be released, and the new string value created by using the OLE Bstr string functions only in the DLL.
>
> To return a byte string to VBA from a DLL, you should modify a byte-string BSTR argument in place. For this to work, you must declare the DLL function as taking a pointer to a pointer to the BSTR and in your C/C++ code, and declare the argument in the VBA code as ' ByRef varg As String'.
>
> You should only handle strings that are passed in these ways from VBA using the OLE BSTR string functions to avoid memory-related problems."

B. Calling DLL functions directly from worksheet

One way of accessing DLL functions directly from the worksheet is:

- Declare the function in VBA as described previously and access it via a VBA user-defined function.

C. Calling DLL commands directly from Excel

One way of accessing DLL commands directly from Excel is:

- Declare the command in VBA as described previously and access it via a VBA macro.

D. DLL memory and multiple DLL instances

"When an application loads a DLL, the DLL's executable code is loaded into the global heap so that it can be run, and space is allocated on the global heap for its data structures. Windows uses memory mapping to make these areas of memory appear as if they are in the application's process so that the application can access them."

"DLL developers do not have to be concerned about static and global variables and data structures being accessed by more than one application, or more than one instance of the same application. Every instance of every application gets its own copy of the DLL's data."

TABLE I. Table of data types and their equivalents

VBA	C++ comment
`String`	Passed as pointer to `BSTR` structure when passed `ByVal`, and as pointer to a pointer when passed as `ByRef`
`Variant` (of string)	Passed as pointer to a Unicode wide-character string `BSTR` structure when passed `ByVal`, and as pointer to pointer when passed `ByRef`
`Integer`	16-bit equivalent to a signed short
`Long`	32-bit equivalent to signed int
`Type`	User-defined and equivalent to `struct`
`Variant`	Found in Windows OLE/COM header files as `VARIANT`
`SafeArrays` (VBA arrays)	Found in Windows OLE/COM header files as `SAFEARRAY`

E. Calling user-defined functions from DLLs

"Calling user-defined functions (UDFs) from a worksheet is as simple as calling built-in functions: You enter the function via a cell formula." See here.

VI. ACCESSING XLL CODE IN EXCEL

Functions and commands in an XLL *must be exported by the XLL* and *registered with Excel* to be accessible in Excel. The registration gives the following information about a DLL entry point, among other things:

- Whether it is hidden or, if a function, whether it is visible in the Function Wizard.
- What its XLL/DLL export name is, and what name you want Excel to use.
- If it is a function:
  - Return data type and arguments.
  - Whether it returns its result by modifying an argument in place.
  - Whether it is volatile.
  - Whether it is thread safe.
  - ...

This would be done using `xlfRegister` function if using the C API.

VII. CALLING INTO EXCEL FROM THE DLL OR XLL

"Microsoft Excel enables your DLL to access built-in Excel commands, worksheet functions, and macro sheet functions. These are available both from DLL commands and functions called from Visual Basic for Applications (VBA), and from registered XLL commands and functions called directly by Excel."

A. Excel4, Excel4v, Excel12, and Excel12v Functions

"Excel enables your DLL to access the commands and functions through the callback functions Excel4, Excel4v, Excel12, and Excel12v." The latter two, denoted 12, were introduced in Excel 2007 and use the `XLOPER12` data structure instead of the `XLOPER` structure. `Excel12` and `Excel12v` are found in the SDK C++ source file `Xlcall.cpp`

For other details, see here.

VIII. CREATING XLLS

If your DLL needs to access Excel functionality (for example, to get the contents of a cell, to call a worksheet function, or to interrogate Excel to obtain workspace information), your code must be able to call back into Excel.

The Excel C API provides several functions that enable DLLs to call back into Excel. To access these, the DLL must be linked statically at compile time with the Excel 32-bit library, xlcall32.lib. The static library is downloadable from Microsoft as part of the Microsoft Excel 2013 XLL SDK, which includes both 32-bit and 64-bit versions of this library.

A. Enabling DLLs to Call Back into Excel

"For a DLL to be able to access the functionality in Excel and get or set workspace information, it must first obtain the addresses of the Excel callback functions Excel4, Excel4v, Excel12, and Excel12v." The file to be `#included` is `Xlcall.h`, and `Xlcall32.lib` and `Xlcall.lib` must be linked. See the documentation for more information regarding the creation of XLL add-ins for Excel.

IX. EXCEL PERFORMANCE: TIPS FOR OPTIMIZING PERFORMANCE OBSTRUCTIONS

The following are a set of tips for increasing the performance of Excel.

A. Optimize references and links

The following are pointers and tips regarding the proper design of formulas in Excel.

Do not use forward referencing and backward referencing: Avoid formulas that refer forward, i.e. to the right or below, to other formulas or cells. In extreme cases this may increase the time of formulating the book's calculation sequence.

Do not use forward referencing and backward referencing: Avoid circular references as they require iterations.

Avoid links between workbooks: Avoid inter-workbook links as they are often slow and wasy to break.

Minimize links between worksheets: "Using many worksheets can make your workbook easier to use, but generally it is slower to calculate references to other worksheets than references within worksheets."

B. Minimize the used range

"Sometimes various editing and formatting operations extend the used range significantly beyond the range that you would currently consider used. This can cause performance obstructions and file-size obstructions."

You can check the visible used range on a worksheet by using `Ctrl + End`. If excessive, consider deleting all the rows and columns below and to the right of the real last used cell, and then saving the workbook. Create a backup copy first. If you have formulas with ranges that extend into or refer to the deleted area, these ranges will be reduced in size or changed to `#N/A`.

C. Allow for extra data – properly

When frequently adding rows or columns of data to your worksheets, you need to find a way of having your Excel formulas automatically refer to the new data area, instead of trying to find and change your formulas every time. This may be dodged by referencing larger (than otherwise necessary) ranges, although inefficient and more difficult to maintain.

There are three options to solving this issue.

1. Use structured table references (recommended)

Using these tables ensure that the references automatically expand and contract with the changes in the referenced table.

Three advantages are:

- Fewer performance versus the next two alternatives of whole column referencing and dynamic ranges.
- Easy to have multiple pages in one worksheet.
- Formulas embedded in the table also expand/contract with the table.

2. Alternatively, use whole column and row references

For example, referencing the column reference `$A:$A` may be used instead of a fixed column range reference. This, however, has difficulties when there are multiple tables in a sheet and uses unnecessary computation power when it fails to recognize the last row in a column.

3. Alternatively, use dynamic ranges

"By using the `OFFSET` or `INDEX` and `COUNTA` functions in the definition of a named range, you can make the area that the named range refers to dynamically expand and contract." The `OFFSET` function has the disadvantage of being volatile and may therefore not be a beneficial alternative. `COUNTA` also decreases performance as it must examine a large number of rows in the specified column. Another option is the `INDIRECT` function used to construct dynamic ranges, though it, too, is volatile and single-threaded.

D. Improve lookup calculation time

On unsorted data, the functions `VLOOKUP`, `HLOOKUP`, and `MATCH` have been made much faster since Office 365 version 1809 (released Oct 2018), but prior versions have large performance issues. As such, math-type variants should be preferred over range-lookups.

On sorted data, however, all three functions are fast and use a binary search algorithm, such that the computation time is barely affected by increases in the lookup range.

1. Use `INDEX` and `MATCH` or `OFFSET` instead of `VLOOKUP`

The `VLOOKUP` function is approximately 5 % faster than the `INDEX` and `MATCH` combination, if only used on its own. The flexibility offered by `MATCH`, however, by storing and reusing the `MATCH` value in a secondary cell for example, may greatly increase the performance versus `VLOOKUP`.

Another option is to sort the data pre-search, using `SORT` which is a fast function. Keeping the lookups and data one a single sheet should also speed up the computation.

When an exact match is required, the range of searched cells should be restricted to a minimum whenever possible. By using tables and structured references

2. Use two lookups for sorted data with missing values

"Two approximate matches are significantly faster than one exact match for a lookup over more than a few rows. (The break-even point is about 10-20 rows.)"

"If you can sort your data but still cannot use approximate match because you cannot be sure that the value you are looking up exists in the lookup range, you can use this formula:"

```
IF(VLOOKUP(lookup_val, lookup_array,1,
    True)=lookup_val, _
     VLOOKUP(lookup_val, lookup_array,
        column, True), "notexist")
```

The first part of the formula works by doing an approximate lookup on the lookup column itself. You check if the answer from the lookup column is the same as the lookup value (in which case you have an exact match). If this formula returns True, you have found an exact match, so you can do the approximate lookup again, but this time, return the answer from the column you want.

E. Create faster VBA macros

1. Turn off everything but the essentials while code is running

- `Application.ScreepUpdating`: Turn off screen updating.
- `Application.DisplayStatusBar`: Turn off the status bar.
- `Application.Calculation`: Switch to manual calculation during the period of the macro running.
- `Application.EnableEvents`: Turn off events. If there are add-ins listening for Excel events, disabling this may save resources when executing a macro.
- `ActiveSheet.DisplayPageBreaks`: Turn off page breaks. Disabling this avoids the recalculation of page breaks during code runs.

2. Read and write large blocks of data in a single operation

Instead of reading and writing single cells by iterating, perform the entire operation at once by first reading the cells into a `Variant` range and then write the entire range to the sheet. See below.

```
Dim DataRange As Variant

DataRange = Range("A1:C10000".Value2
' Compute something...'
Range("A1:C10000").Value2 = DataRange
```

Also note: Use `.Value2` over `.Text`(returns the formatted cell value and may accidentally return ### if zoomed) and `.Value`(which returns a VBA currency or VBA date if range is formatted as Currency or Date – which is slow) since it does not alter the data retrieved from Excel.

3. Additional VBA performance optimizations

- Return results by assigning an array to a `Range`.
- Declare variables explicitly to save the overhead of determining the data type. (Toggle `Option Explicit` to force this.)

F. Additional considerations

- **Conditional formats and data validation:** Conditional formats and data validation are great, but using a lot of them can significantly slow down calculation. If the cell is displayed, every conditional format formula is evaluated at each calculation and when the display of the cell that contains the conditional format is refreshed. The Excel object model has a `Worksheet.EnableFormatConditionsCalculation` property so that you can enable or disable the calculation of conditional formats.
- **Defined names:** Because names are calculated every time a formula that refers to them is calculated, you should avoid putting calculation-intensive formulas or functions in defined names. In these cases, it can be significantly faster to put your calculation-intensive formula or function in a spare cell somewhere and refer to that cell instead, either directly or by using a name.