FIRST EDITION – CHAPTER 1 REV 1

Kevin Thomas

Forward

A long time ago there existed a time and space where the 6502 processor was everywhere. There was no internet, there was no cell phone and the personal computer was that of a creation of pure majesty which had a target market of a few enthusiasts.

On November 20, 1985, Microsoft introduced the Windows operating environment which was nothing more than a graphical operating shell for MS-DOS.

I will spare you the rest of the history as we know how this game played out. Today, Windows is the most used desktop and laptop OS having a 76% share followed by Apple’s macOS at 16% and the remaining ChromeOS and other Linux variants.

Like it or not Windows is the major player and throughout the years I have focused on teaching Reverse Engineering in the Linux environment so that we could focus on a more thinner and efficient development and communication with the processor.

Today we begin our journey into the Win32API. This book will take you step-by-step writing very simple Win32API’s in both x86 and x64 platforms in C++ and then reversing them both very carefully using the world’s most popular Hey Rays IDA Free tool which is a stripped down version of the IDA Pro tool used in more professional Reverse Engineering environments.

Let’s begin...

Table Of Contents

Chapter 1: Hello World

We begin our journey with programming a very simple hello world program in Windows Assembly language. We will ONLY write in pure Assembly in this chapter as we will focus on development in C++ which almost all Windows development occurs so you have a greater understanding of how these applications are put together and THEN reversing the entire app in Assembly Language both in x86 and x64.

Let's first download Visual Studio which we will use as our integrated development environment. Select the Visual Studio 2019 Community edition at the link below. Make SURE you select all of the C++ and Windows options during the setup to ensure the build environment has all the tools necessary. When in doubt, check the box to include it during install.

https://visualstudio.microsoft.com/downloads/

Once installed, let's create a new project and get started by following the below steps.

```
Create a new project
Empty Project
Next
Project name: 0x0001-hello_world-x86
CHECK Place solution and project in the same directory
Create

RT CLICK on the 0x0001-hello_world-x86 in Solutions Explorer
Add
New Item…
main.asm
RT CLICK 0x0001-hello_world-x86
Build Dependencies
Build Customizations
CHECK masm
OK

RT CLICK on main.asm
Properties
Configuration Properties
General
Item Type: Microsoft Macro Assembler
OK
```

Now let's populate our **main.asm** file with the following.

```
.686
.model flat, stdcall
.stack 4096

extrn  ExitProcess@4: proc         ;1 param  1x4
extrn  MessageBoxA@16: proc        ;4 params 4x4

.data
    msg_txt      db "Hello World", 0
    msg_caption  db "Hello World App", 0

.code
    main:
        push  0                    ;UINT uType
        lea   eax, msg_caption     ;LPCSTR lpCaption
        push  eax
        lea   eax, msg_txt         ;LPCSTR lpText
        push  eax
        push  0                    ;HWND hWnd
        call  MessageBoxA@16

        push  0                    ;UINT uExitCode
        call  ExitProcess@4
    end main
```

Congratulations! You just created your first hello world code in x86 Windows Assembly. Time for cake!

We are going to spend the majority of our time in the Win32API documentation throughout this course.

Let's take a moment and review. To begin we designate a *.686* which means enable the assembly of non-privileged instructions for the Pentium Pro+ style architecture in 32-bit MASM.

(VISIT https://docs.microsoft.com/en-us/cpp/assembler/masm/dot-686?view=msvc-160)

We then set up a *flat* memory model which uses no combined segment or offset addressing. We also use the *stdcall* Win32 callign convention which we push args in reverse order onto the stack and then call the procedure.

Our first Win32API that we will call is the *ExitProcess* which simply exits the application and frees up the operation to the Windows OS.

(VISIT https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitprocess)

We see that the function is a void function which returns nothing and has one param UINT uExitCode which simply retrieves the process's exit value.

You might have noticed a very strange *@4* after the function. This is to designate that the function has 1 param. We multiply each param by 4 to get this designation.

Our next Win32API is the *MessageBoxA* function which simply displays a modal dialog box with a title and a message.

(VISIT https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa)

We have 4 params here so we know we will have an @16 at the end of the function.

The first param is *HWND hWnd* which is a handle to the owner of the window of the message box to be created and in our case it is *NULL* meaning the message bhas has no owner.

We then have the *LPCSTR lpText* which will display our text inside the message box.

We then have the *LPCSTR lpCaption* which will be the caption text on the message box.

Finally we have the UINT uType which is simply the combo of flags from the table located in the docs. In our case it will be NULL.

Remember in stdcall we push the params in REVERSE order onto the stack as you see in the code above.

At this point we can run our code by clicking on the green arrow next to the Local Windows Debugger.

HOORAY our hello world modal dialog box pops up.

Let’s now create our x64 version of this code.

```
Create a new project
Empty Project
Next
Project name: 0x0001-hello_world-x64
CHECK Place solution and project in the same directory
Create

RT CLICK on the 0x0001-hello_world-x64 in Solutions Explorer
Add
New Item…
main.asm
RT CLICK 0x0001-hello_world-x64
Build Dependencies
Build Customizations
```

CHECK masm
OK

RT CLICK on the 0x0001-hello_world-x64 in Solutions Explorer

RT CLICK on main.asm
Properties
Configuration Properties
Linker
Advanced
Entry Point: main
OK

Select x64 to the right of Debug and the left of Local Windows Debugger menu bar

Now let’s populate our **main.asm** file with the following.

```
extrn MessageBoxA: proc
extrn ExitProcess: proc

.data
      msg_txt     db 'Hello World', 0
      msg_caption db 'Hello World App', 0

.code
      main proc
            sub    rsp, 20h            ;shadow stack

            mov    r9, rax             ;UINT uType
            lea    r8, msg_caption     ;LPCSTR lpCaption
            lea    rdx, msg_txt        ;LPCSTR lpText
            xor    rcx, rcx            ;HWND hWnd
            call   MessageBoxA

            add rsp, 20h               ;restore shadow stack

            mov rcx, rax               ;UNIT uExitCode
            call ExitProcess

            ret
      main endp
end
```

Congratulations! You just created your first hello world code in x64 Windows Assembly. Time for cake, again!

Let’s take a moment and review. We first need to understand the x64 calling convention.

(VISIT https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160)

What we see here under the *Parameter passing* section is by default, the x64 calling convention passes the first four arguments to a function in registers. The registers used for these arguments depend on the position and type of the argument. Remaining arguments get pushed on the stack in right-to-left order.

Integer valued arguments in the leftmost four positions are passed in left-to-right order in RCX, RDX, R8, and R9, respectively. The fifth and higher arguments are passed on the stack as previously described. All integer arguments in registers are right-justified, so the callee can ignore the upper bits of the register and access only the portion of the register necessary.

Any floating-point and double-precision arguments in the first four parameters are passed in XMM0 - XMM3, depending on position. Floating-point values are only placed in the integer registers RCX, RDX, R8, and R9 when there are varargs arguments. For details, see Varargs. Similarly, the XMM0 - XMM3 registers are ignored when the corresponding argument is an integer or pointer type.

According to the x64 calling convention we need to provide a shadow stack for memory cells for each QWORD and the stack has to be aligned to 16 bytes for the next instruction.

The shadow space is the mandatory 32 bytes (4x8 bytes) which we must reserve for the called procedure. We provide 32 bytes on the stack before calling. This space can be left uninitialized.

In this calling convention, arguments after the 4th are pushed on the stack, which are on top of this shadow space (pushed before the 32 bytes).

We then setup and call our *MessageBoxA* Win32API again. We do not need to review the params as we have handled this earlier in our x86 example.

We then restore the shadow stack and then call *ExitProcess*.

At this point we can run our code by clicking on the green arrow next to the Local Windows Debugger.

HOORAY our hello world modal dialog box pops up.

This will be the only example where we write in all Assembly as I want to teach using the official Win32API which is natively in C++ however I wanted to first show you EXACTLY what is going on under the hood when it is in fact compiled.