LAB 4

PART I: STEP INTO/OVER, AND THE WATCH & CALL STACK WINDOWS

In Part I, you will learn (or review) the difference between the Step Into and Step Over commands in the debugger, and you'll learn how to use the Watch and Call Stack windows in Visual Studio. These will be useful in the future when writing programs with several procedures.

In Part II, you will build a Windows application that does not use the Irvine32 library. Instead, you will use Windows' application programming interface ("the Win32 API") directly. You'll see that Win32 API functions are called using *stack parameters*... and you'll see that using the Win32 API is more complex than using Kip Irvine's library.

Directions. Turn in answers to ? questions. Submit an .asm file in Canvas for the last exercise.

```
INCLUDE Irvine32.inc

.data
msgPressKey BYTE  "Press any key to exit...", 0
result      DWORD ?

.code
main PROC
    ; Do some very inefficient math
    mov eax, 3
    call Double        ; Breakpoint (2) will be set here
    call Quadruple
    mov result, eax    ; Breakpoint (1) will be set here
    call WriteDec
    call Crlf

    ; Press any key to exit
    mov edx, OFFSET msgPressKey
    call WriteString
    call ReadChar
    exit
main ENDP

Double PROC
    add eax, eax       ; Breakpoint (3) will be set here
    ret
Double ENDP

Quadruple PROC
    call Double
    call Double
    ret
Quadruple ENDP
END main
```

1. Getting Started

☐ **Enter the above program in Visual Studio.** It's easiest to start with Irvine's Project_sample and modify it, as you have done previously. (Do **not** set breakpoints yet; you will do that later.) If you don't want to type the code, it is posted in Canvas as a text file (Lab4Part1.txt) so you can copy and paste it... but do read through the code to understand what it does before you move on.

☐ **Run the program to make sure you entered it correctly.** It should display the number 24 and ask you to press any key. After you press a key, it should exit.

2. The Watch Window

☐ **Set a breakpoint on the line that reads "Breakpoint (1) will be set here."**

☐ **Start debugging. The debugger will stop at the breakpoint.**

☐ **In Visual Studio, open a Watch window.** Click Debug > Windows > Watch > Watch 1.

☐ **Add `msgPressKey` to the set of watched variables.** Click in the Watch 1 window. The first (empty) row will be highlighted. In this first, highlighted row, click in Name column. A text box will appear, allowing you to type the name of a variable. Type `msgPressKey` and then press Enter. The Value and Type columns should be populated.

☐ **Add `result` to the set of watched variables** in the same way**.** Note that `result` was uninitialized (`result DWORD ?`), and because the breakpoint is *on* the line that moves a value into `result`, it is still uninitialized. So, the value will be whatever (garbage) is currently in memory.

☐ ? **What is the current value of `result`?** ______________
(If it is zero, consider yourself lucky. Uninitialized variables are not guaranteed to have a zero value.)

☐ **Step to the next instruction.** Click Debug > Step Over, or press F10. The instruction `mov result, eax` will execute, setting the value of `result`. The value of `result` in the Watch window will be updated accordingly and will be displayed in red to indicate that it changed.

☐ ? **What is the new value of `result`?** ______________

☐ **Terminate the program, and remove the breakpoint.**

3. STEP INTO/STEP OVER & THE CALL STACK WINDOW

- ☐ **Set a breakpoint on the line that reads "Breakpoint (2) will be set here."**

- ☐ **Start debugging. The debugger will stop at the breakpoint.**

- ☐ **Step *over* the statement `call Double`.** Click Debug > Step Over, or press F10. Note that the current line indicator moves to the next instruction, **`call Quadruple`**.

- ☐ **Step *into* the statement `call Quadruple`.** Click Debug > Step Into, or press F11. Note that the current line indicator moves to the first instruction of the Quadruple function.

For most assembly language instructions, the Step Into and Step Over commands behave identically. They differ when the current line indicator is on a `call` instruction: Step Into steps the debugger to the the first instruction *inside* the called procedure, while Step Over steps the debugger to the next instruction *after* the `call`.

- ☐ **Set a breakpoint on the line that reads "Breakpoint (3) will be set here."**

- ☐ **Continue execution.** The debugger will stop at the breakpoint you just set in the Double function.

Now, the current line indicator is in the Double procedure. Unfortunately, the function Double is called at three different places in your program: it's called from main, and it's called twice from the Quadruple function. So, we just hit a breakpoint in Double... but where was Double called from? To figure it out, do this.

- ☐ **Open the Call Stack window.** Click Debug > Windows > Call Stack. The top entry in the call stack is the procedure that is currently executing (Double). The next entry is the procedure that called it. The third entry is the procedure that called the second. And so forth.

- ☐ **Double-click on the second entry in the call stack.** A green arrow will appear next to the instruction *after* the call to Double.

- ☐ ? **At the current point in the program's execution, was `Double` called from from `main` or `Quadruple`?** ____________________

- ☐ **Step over instructions repeatedly until you are back in the `main` procedure.** Pay special attention to what happens after you step over a RET instruction: the next instruction to execute is the one following the CALL to that procedure.

- ☐ **Terminate the program, and remove the breakpoints.**

4. Procedure Errors

Now, you are going to intentionally introduce some procedure-related errors, just to see what happens. (Hopefully, having seen what happens will make it easier to catch these errors in the future.)

A. Missing RET Instruction

- ☐ **Comment out one of the RET instructions.**
- ☐ **Start debugging.**
- ☐ ⓘ **What error occurs?** ______________________________
- ☐ **Terminate the program.**
- ☐ **Uncomment the RET instruction, and make sure the program runs normally.**

B. Incorrect Procedure Name

- ☐ **Misspell the procedure name in one of the CALL instructions.**
- ☐ ⓘ **Attempt to start debugging. Assembly will fail. What error does MASM report?** ______________________________
- ☐ **Fix the spelling of the procedure name, and make sure the program runs.**
- ☐ **Misspell the procedure name in one of the ENDP directives, so that it does not match the name in the corresponding PROC directive.**
- ☐ ⓘ **Attempt to start debugging. Assembly will fail. What error does MASM report?** ______________________________
- ☐ **Fix the spelling of the procedure name, and make sure the program runs.**

C. Mismatched PUSH/POP

- ☐ **Add some PUSH instructions to the Double procedure *without* matching POPs.**
- ☐ **Start debugging.**
- ☐ ⓘ **What error occurs?** ______________________________
- ☐ **Terminate the program.**
- ☐ **Remove the PUSH instructions, and make sure the program runs normally.**

PART II: GO AWAY, KIP IRVINE

1. CREATING A NEW WIN32 CONSOLE APPLICATION[1]

☐ Open Visual Studio 2010. If it is already open, click **File > Close Solution**. (You are going to create a new project that *doesn't* use Kip Irvine's library.)

☐ In Visual Studio, click **File > New > Project** to open the New Project dialog.

☐ In the list on the left, under Installed Templates, select **Visual C++ > Win32**.

☐ In the center panel, select **Win32 Console Application**.

☐ In the **Name** text box, enter Win32ConsoleAsm

☐ In the **Location** text box, select a location on a non-network drive (such as your local documents folder – C:\Users\yourid\Documents\ – or a folder on a USB drive).

☐ Click **OK**. The Win32 Application Wizard dialog will open.

☐ Click **Next** to proceed to the next page of the wizard.

☐ Under **Additional options**, check **Empty Project**.

☐ Click **Finish** to close the wizard.

☐ In the Solution Explorer (on the right side of the Visual Studio window), right-click on the Win32ConsoleAsm project, and from the context menu, select **Build Customizations**. This will open the Visual C++ Build Customization Files dialog.

☐ Check the box labeled **masm(.targets, .props)**.

☐ Click **OK** to close the dialog.

[1] The instructions here are partially based on
http://scriptbucket.wordpress.com/2011/10/19/setting-up-visual-studio-10-for-masm32-programming/
A StackOverflow discussion of compiling assembly language code in Visual Studio 2010 is at
http://stackoverflow.com/questions/4548763/compiling-assembly-in-visual-studio
which in turn references instructions for configuring build customizations at
http://connect.microsoft.com/VisualStudio/feedback/details/538379/adding-macro-assembler-files-to-a-c-project

- ☐ In the Solution Explorer, right-click on **Source Files,** and select **Add > New Item**. This will open the Add New Item dialog.

- ☐ In the list on the left, select **Visual C++**.

- ☐ In the center panel, choose **Text File (.txt)**.

- ☐ In the **Name** text box, enter main.asm

- ☐ Click **Add** to close the dialog.

- ☐ Type the following into the main.asm file, and then save it.

```
.model flat, stdcall

ExitProcess PROTO, dwExitCode:DWORD

.code
main PROC
    push 0
    call ExitProcess
main ENDP
END main
```

- ☐ In the Solution Explorer, right-click the Win32ConsoleAsm project, and select **Properties** from the context menu.

- ☐ In the tree on the left, select **Configuration Properties > Linker > Advanced**.

- ☐ In the list on the right, click on the text area to the right of **Entry Point**. Type *main* and press the Enter key. (This tells the linker that the entrypoint for the generated executable file will be a procedure named *main*. This must match the name in the END directive at the end of your .asm file.)

- ☐ Click **OK** to close the dialog.

- ☐ Click **Build > Build Solution**. If the linker crashes, click **Build > Clean Solution**, then click **Build > Build Solution** again; usually, it will succeed the second time even if it crashes the first time.

- ☐ Start debugging. Your program does not do anything yet—all it does is exit cleanly (i.e., exit without an error)—so a console window should open and then close immediately.

The first part of this file looks from what you're used to, since we're no longer including Irvine32.inc. Here are the new parts:[2]

- The **.model** directive indicates to the assembler what *memory model* is being used. For 32-bit programs, the **flat** memory model is always used. For 16-bit programs (e.g., for older versions of Windows or MS-DOS), there are several alternative memory models, which we might discuss later in the course.

- Since we need to call Windows API functions, we also provide the **stdcall** option to the .model directive. This indicates the *naming and calling convention* that is being used. In a few lectures, we'll talk about more calling conventions in general and the STDCALL convention in particular, so don't worry too much about this yet.[3]

- *ExitProcess* is a Windows API function that terminates your program; it is located in kernel32.dll. When you want to call a procedure in an external library (a static or dynamically linked library), you need to give the assembler some information about that procedure, so the procedure calls in your program can be linked to the correct procedures in kernel32.dll. We're using the **PROTO** directive to tell the assembler what the function's name is, how many arguments it expects, and what their types are:

  ```
  ExitProcess PROTO, dwExitCode:DWORD
  ```

  This is known as a *function prototype* (which is why the directive is called "PROTO"). Wikipedia has a good definition <http://en.wikipedia.org/wiki/Function_prototype>: "A function prototype ... is a declaration of a function that omits the function body but does specify the function's return type, name, arity [i.e., number of arguments] and argument types. While a function definition specifies HOW the function does what it does, a function prototype merely specifies its interface, i.e., WHAT data types go in and come out of it."

  (Contrary to the Wikipedia definition, MASM's PROTO directive does *not* specify the return type, because the assembler does not need to know about it. If a value is returned, it will be returned in EAX.)

---

[2] The official documentation for these directives is in the *Microsoft Macro Assembler Reference*, which is online at <http://msdn.microsoft.com/en-us/library/afzk3475.aspx> under the section "Directives Reference." Unfortunately, it's remarkably cryptic and, in some cases, incomplete.

[3] You could also add another line with "option casemap:none" to specify the naming convention, but it's not necessary since that is implied by STDCALL. From https://www.xlsoft.com/jp/products/intel/cvf/docs/vf-html_e/pg/pgwadmsm.htm: In MASM .... specifying the C or STDCALL naming convention in PROC and PROTO statements preserves case sensitivity if no CASEMAP option exists. The MASM OPTION CASEMAP directive (and the command line option /C) also sets case sensitivity and overrides naming conventions specified within PROTO and PROC statements:
• CASEMAP: NONE (equivalent to /Cx) preserves the case of identifiers in PUBLIC, COMM, EXTERNDEF, EXTERN, PROTO, and PROC declarations.
• CASEMAP: NOTPUBLIC (equivalent to /Cp) preserves the case of all user identifiers; this is the default.
• CASEMAP: ALL (equivalent to /Cu) translates all identifiers to uppercase.

2. Stack Parameters[4]

Look at the prototype for the *ExitProcess* function above, and notice that it takes one argument: a DWORD, which we called *dwExitCode.* To call the *ExitProcess* function, we used this instruction sequence:

```
push 0
call ExitProcess
```

Up to this point in the course, we have only used registers to pass arguments to procedures. We can say that the procedures used *register parameters*. Register parameters are optimized for program execution speed and they are easy to use. Unfortunately, register parameters tend to create code clutter in calling programs. Existing register contents often must be saved before they can be loaded with argument values.

Stack parameters offer a more flexible approach. Just before the subroutine call, the arguments are *pushed on the stack*. To call a procedure that uses stack parameters:

- Push the arguments onto the stack *from right to left.* (Yes, push them in *reverse* order.)
- Always push 32-bit values onto the stack. If the procedure requires a BYTE or WORD, zero- or sign-extend it to 32-bits before pushing it.

Example 1. Let's suppose you have a procedure named *subtract*. If you wanted to call it with two arguments, in a language like C/C++, you'd write:

```
subtract(7, 1)     // Returns 7 - 1 = 6
```

In assembly language, you would push the arguments in the reverse order:

```
push 1
push 7
call subtract      ; Returns 7 - 1 = 6
```

Example 2. Suppose you have a function *doSomething*. You want to pass the values in EAX, EBX, and ECX as the first three arguments and 100 as the fourth argument:

```
doSomething(eax, ebx, ecx, 100)
```

In assembly language, the equivalent call is:

```
push 100
push ecx
push ebx
push eax
call doSomething
```

Remember: push the arguments *in the reverse order.*

[4] This text is adapted from Section 8.2.1 "Stack Parameters" from your textbook.

3. WriteConsole

Remember the *WriteString* procedure you've been calling from Kip Irvine's library? Now, you'll see how it's actually implemented. Add the lines in bold to your program:

```
.model flat, stdcall

STD_INPUT_HANDLE = -10
STD_OUTPUT_HANDLE = -11
STD_ERROR_HANDLE = -12

GetStdHandle PROTO,  ; Get handle for stdin/stdout/stderr
    nStdHandle:DWORD

WriteConsoleA PROTO,  ; Display output on the console
    hConsoleOutput:DWORD,             ; Console handle
    lpBuffer:PTR BYTE,                ; Output buffer
    nNumberOfCharsToWrite:DWORD,      ; Size of buffer
    lpNumberOfCharsWritten:PTR DWORD, ; Num bytes written
    lpReserved:DWORD                  ; (Not used)

ExitProcess PROTO, dwExitCode:DWORD

.data
message         BYTE "Hello, world!"
hStdout         DWORD ?
numCharsWritten DWORD ?

.code
main PROC
    ; Get a handle for standard output
    push STD_OUTPUT_HANDLE
    call GetStdHandle
    mov hStdout, eax

    ; Write hello to standard output
    push 0
    push OFFSET numCharsWritten
    push LENGTHOF message
    push OFFSET message
    push hStdout
    call WriteConsoleA

    ; Exit
    push 0
    call ExitProcess
main ENDP
END main
```

What is all this?

- A *handle* is a 32-bit unsigned integer used to refer to a particular file, network stream, console, or other "resource" supplied to you by the operating system.
- To display output, you'll need to write to *standard output*, sometimes abbreviated *stdout.* But to write to standard output, you'll need to get a *handle* to it.
- To get a handle for standard output, call `GetStdHandle(-11).` The value –11 indicates that you want a handle for standard output (as opposed to *standard input*, which you use to *read* input, e.g., from the keyboard) or *standard error*, which you typically use only to display error messages).
- Having magic numbers like –11 floating around in your program is extremely poor style—not even seasoned Windows programmers remember that –11 means "get the handle for standard output"—so we've made a symbolic constant:
  ```
  STD_OUTPUT_HANDLE = -11
  ```
- *GetStdHandle* returns the handle in EAX. We're using a `mov` to save it in memory, so it's easy to access later. (This isn't really necessary, but when you write longer programs, you'll find that storing values like this in memory is easier than trying to keep them in registers for an arbitrarily long time.)
- Finally, we can call *WriteConsoleA* to display a message. It takes five arguments:
  - *hConsoleOutput* – the handle to standard output returned by *GetStdHandle*
  - *lpBuffer* – the memory address of the first character of the output message
  - *nNumberOfCharsToWrite* – the number of characters in the message to display
  - *lpNumberOfCharsWritten* – the memory address of a DWORD where Windows can store the number of characters it *actually* displayed (which could be less than the number you asked it to display)
  - *lpReserved* – this is not used in current versions of Windows (pass 0 to be safe)
- The parameters to *WriteConsoleA* have the form *name*:*type*. The type is as follows:
  - DWORD indicates that a DWORD *value* will be passed for that argument.
  - PTR DWORD indicates that the *memory address* of a DWORD will be passed.
  - PTR BYTE indicates that the *memory address* of a BYTE will be passed.

The difference between DWORD and PTR DWORD is extremely important. When we call *WriteConsoleA*, you must be sure to distinguish *values* from *memory addresses*.

- The first parameter is `hConsoleOutput:DWORD`. Since its type is DWORD, the function expects a DWORD *value* for that argument, so for that argument, we `push hStdout`—in other words, we read the value of *hStdout* from memory and push that value onto the stack.
- The second parameter is `lpBuffer:PTR BYTE`. Since its type is PTR BYTE, the function expects the *memory address* of a BYTE variable. So for that argument, we `push OFFSET message`—i.e., we pass the *memory address* where the first byte of `message` is stored.

Now, answer these questions:

- ☐ ⓘ Step through the program and use the Watch window to answer this question: **What is the value of numCharsWritten *before* you call *WriteConsoleA?* _____ What is its value *after* you call *WriteConsoleA?* _____**

- ☐ ⓘ **Change the third argument from `LENGTHOF message` to `8`. What does the program output? ______________________________**

- ☐ ⓘ **Change the second arugment from `OFFSET message` to `OFFSET message + 2`. What does the program output?**

  ____

4. SLEEP

The Win32 API includes a function called *Sleep*, which has the following prototype:

```
Sleep PROTO,  ; Suspends execution for a period of time
    dwMilliseconds:DWORD     ; Number of milliseconds
```

- ☐ ⓘ **How would you call the *Sleep* function in assembly language if you wanted your program to sleep for 5 seconds?**

5. READCONSOLE (EXERCISE: SUBMIT IN CANVAS)

Now, remember the *ReadString* procedure you've been calling from Kip Irvine's library? It words by calling a Win32 API function called *ReadConsoleA*.

```
ReadConsoleA PROTO,    ; Read input from the console
    handle:DWORD,              ; Input handle
    pBuffer:PTR BYTE,          ; Pointer to buffer to store input
    maxBytes:DWORD,            ; Max number of bytes to read
    pBytesRead:PTR DWORD,      ; Pointer to store number of bytes read
    unused:DWORD               ; Unused (set to 0)
```

For the *handle*, you will need to use *GetStdHandle* to retrieve the handle for standard *input* (STD_INPUT_HANDLE).

- ☐ ⓘ **Write a Win32 program that uses *ReadConsoleA* to read a string (up to 100 characters) from the user, and then uses *WriteConsoleA* display the string the user entered.** When you call *WriteConsoleA*, it should only display as many characters as the user *actually entered*. Submit your .asm file in Canvas.