This document contains concepts and instructions that may be hard to understand. This is **not a test**. This is **not a trick**.

If anything is unclear, stop reading the document then ask these questions:

* Can I clearly say what I **think** I am being asked to do?
* Have I read the entire instruction and do I understand **all** of it?
* Can I tell someone what I think I understand?
* Can I ask someone what they understand about the instruction?

If the answer is “No” to any of these questions, do not proceed further.

Instead, take the document to your co-worker, stream leader or supervisor and work through the instruction with them.

If the document is unclear, it will be updated to make things clear but **we need your feedback!**

**Stop!**

In this document:

Table of Contents

[Pre-requisites 3](#_Toc18501630)

[Reason for this Tutorial 4](#_Toc18501631)

[Expected Outcomes 4](#_Toc18501632)

[What is not covered 4](#_Toc18501633)

[Background 4](#_Toc18501634)

[Install and use a MASM IDE 6](#_Toc18501635)

[Using Visual MASM 2.00 6](#_Toc18501636)

[Using Visual Studio 2017 24](#_Toc18501637)

[Create and compile trivial programs that can run on a Windows machine 31](#_Toc18501638)

[Demo 1 31](#_Toc18501639)

[View and understand the output of the compiled code 34](#_Toc18501640)

[File Headers 34](#_Toc18501641)

[MS-DOS Stub 34](#_Toc18501642)

[Portable Executable (PE) Signature 35](#_Toc18501643)

[COFF File Format Header 36](#_Toc18501644)

[Tools and Techniques 38](#_Toc18501645)

[Why am I doing this? 38](#_Toc18501646)

[Free isn’t always the same as Good 38](#_Toc18501647)

[Mandatory tools for this “Capture the Flag” 38](#_Toc18501648)

[MASM32 DUMPPE 38](#_Toc18501649)

[MASM32 DUMPBIN 38](#_Toc18501650)

[Code De-compiler such as IDA or Ghidra 39](#_Toc18501651)

[Further Reading 39](#_Toc18501652)

[Online Tools 40](#_Toc18501653)

[Free Tools from the Internet 40](#_Toc18501654)

[Write a Report on the code found 41](#_Toc18501655)

[Appendix A: Sample Code 44](#_Toc18501656)

[Demo 1: Hey this actually works 44](#_Toc18501657)

[Demo 2: Basic Window Dialog 45](#_Toc18501658)

# Pre-requisites

The instructions in this document were written for a Windows 10 PC running Chrome or Microsoft Edge browsers. The reader should be reasonably familiar with both the Windows 10 operating system, the Windows Desktop interface and the preferred browser.

The reader should have successfully completed the YASP Assembler tutorial using [YASP.me](http://demo.YASP.me).

The reader should have local administrator permission on the computer being used.

The reader needs to already have a good understanding of how to download and install open source products from public repositories such as [Github](https://github.com/mnewbery/MASM/).

Where gaps in required knowledge are identified, view the Tools and Techniques section for more background information.

# Reason for this Tutorial

Writing and understanding assembler code then viewing the both the output of the compiler and the output of the executed code is a good way to learn how to find and understand malware.

This tutorial will guide the reader through steps to install a Microsoft Macro Assembler (MASM) integrated design environment (IDE) and use it to create then walk through a compiled program.

Skills, knowledge and experience gained through this tutorial are expected to be foundation knowledge for anyone wanting to start an exciting career as a Cyber Security Defender, Cyber Threat Emulator or Offensive Cyber Security Professional.

# Expected Outcomes

At the end of this tutorial, the reader will be able to:

1. Understand the differences between Visual MASM and Visual Studio
2. Use a MASM IDE and be able to explain why MASM is useful in understanding malware
3. Create and compile trivial programs that can run on a Windows machine
4. View and understand the output of the compiled code
5. Using the tools and techniques demonstrated in the tutorial, reverse engineer compiled code to determine then explain the intent of the code; and
6. Write a report on the code found using a [**Capture-The-Flag**](https://en.wikipedia.org/wiki/Capture_the_flag#Computer_security) report style

# What is not covered

Not all byte code comes from a Windows 32 bit compiler. Many examples of malware were written by manipulating the individual bytes in a file in order to make it harder or impossible to determine the source and intent of the malware. This tutorial provides a foundation on viewing and understanding byte code. It does not cover the various compiler versions and settings that can influence the final byte code output. Code obfuscation and self-modifying code are advanced topics not covered here.

# Background

Text below is from [Wikipedia](https://en.wikipedia.org/wiki/Assembly_language)

An assembly language (or assembler language), often abbreviated **asm**, is any low-level programming language in which there is a very strong correspondence between the program's statements and the architecture's machine code instructions.

Assembly code is converted into executable machine code by a utility program referred to as an assembler. The conversion process is referred to as assembly, or assembling the source code. Assembly language usually has one statement per machine instruction, but comments and statements that are assembler directives, macros and symbolic labels of program and memory locations are often also supported.

Each assembly language is specific to a **particular computer architecture** and sometimes to an operating system. However, some assembly languages do not provide specific syntax for operating system calls, and most assembly languages can be used universally with any operating system, as the language provides access to all the real capabilities of the processor, upon which all system call mechanisms ultimately rest. In contrast to assembly languages, most high-level programming languages are generally portable across multiple architectures but require interpreting or compiling.

Text below is from [Wikipedia](https://en.wikipedia.org/wiki/Microsoft_Macro_Assembler)

The Microsoft Macro Assembler (MASM) is an x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows. Beginning with MASM 8.0, there are two versions of the assembler: One for 16-bit & 32-bit assembly sources, and another (ML64) for 64-bit sources only.

MASM is maintained by Microsoft, but since version 6.12 it has not been sold as a separate product. It is instead supplied with various Microsoft SDKs and C compilers. Recent versions of MASM are included with Microsoft Visual Studio.

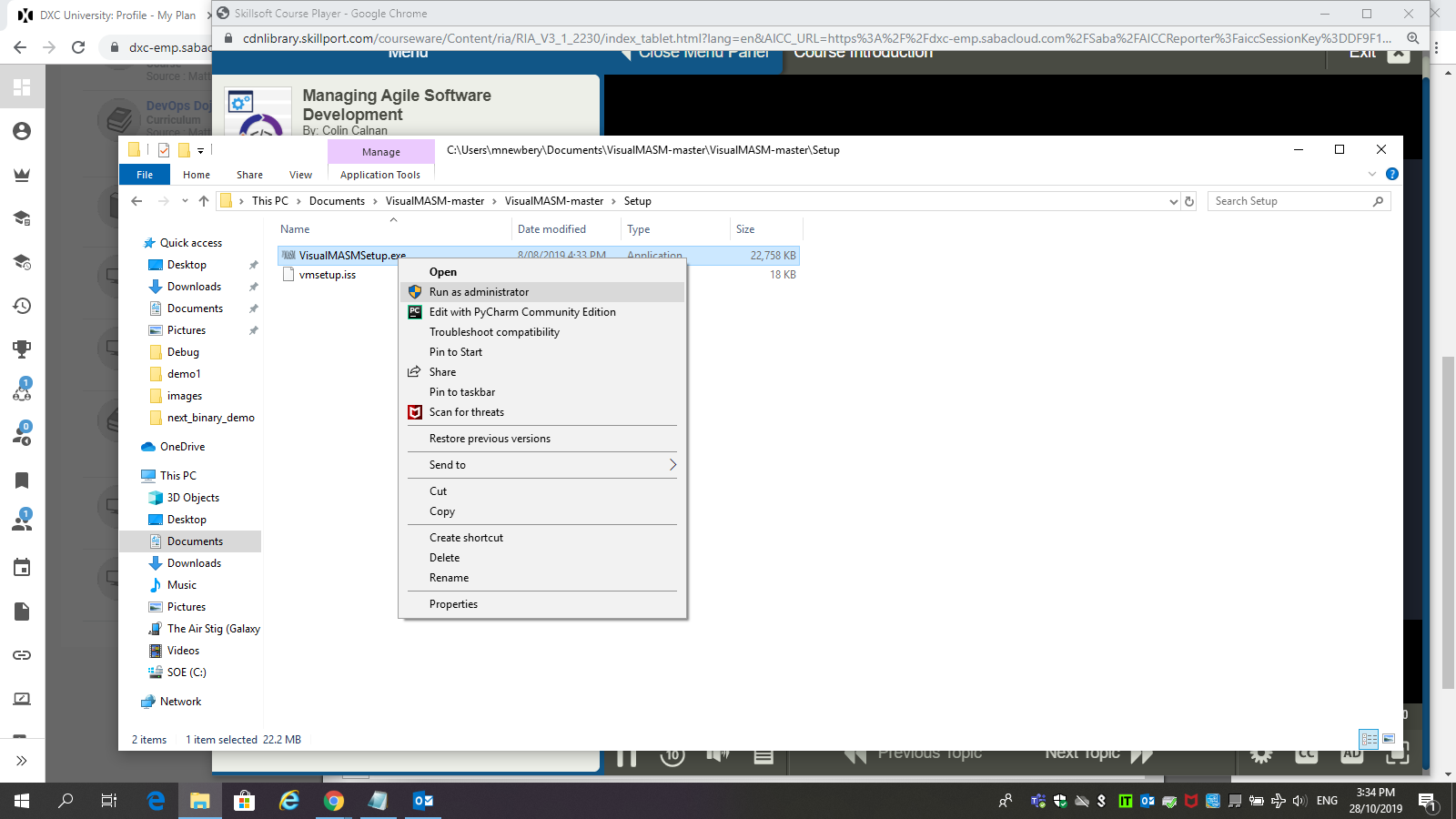
This tutorial focusses on the 32 bit Complex Instruction Set (CISC) assembly language and compiler normally associated with Windows x86 and x64.

The next section explains how to download and install a typical Integrated Design Environment (IDE) for Assembly code.

# Install and use a MASM IDE

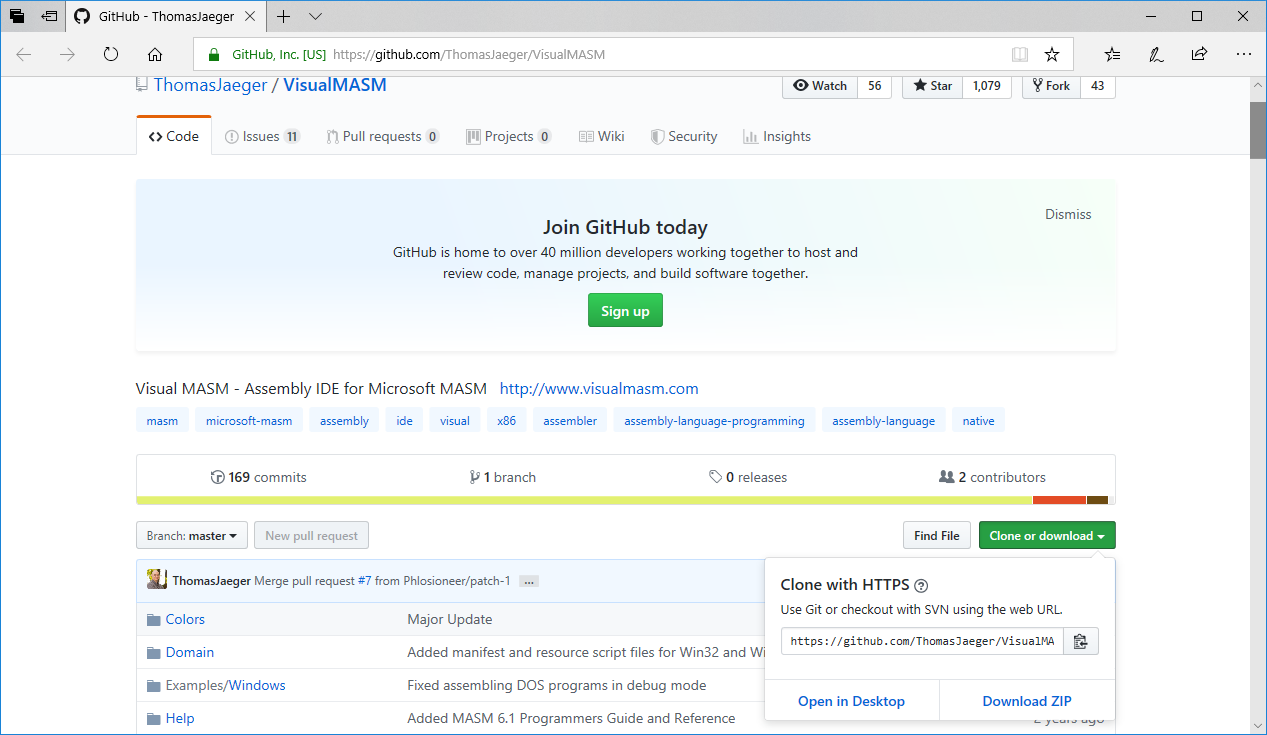
Two IDEs are provided in this tutorial. The reader only needs to install one or the other. If the target PC already has Visual Studio 2017 installed, use Visual Studio 2017 and enable C++. If Visual Studio 2017 is not installed, use Visual MASM 2.00.

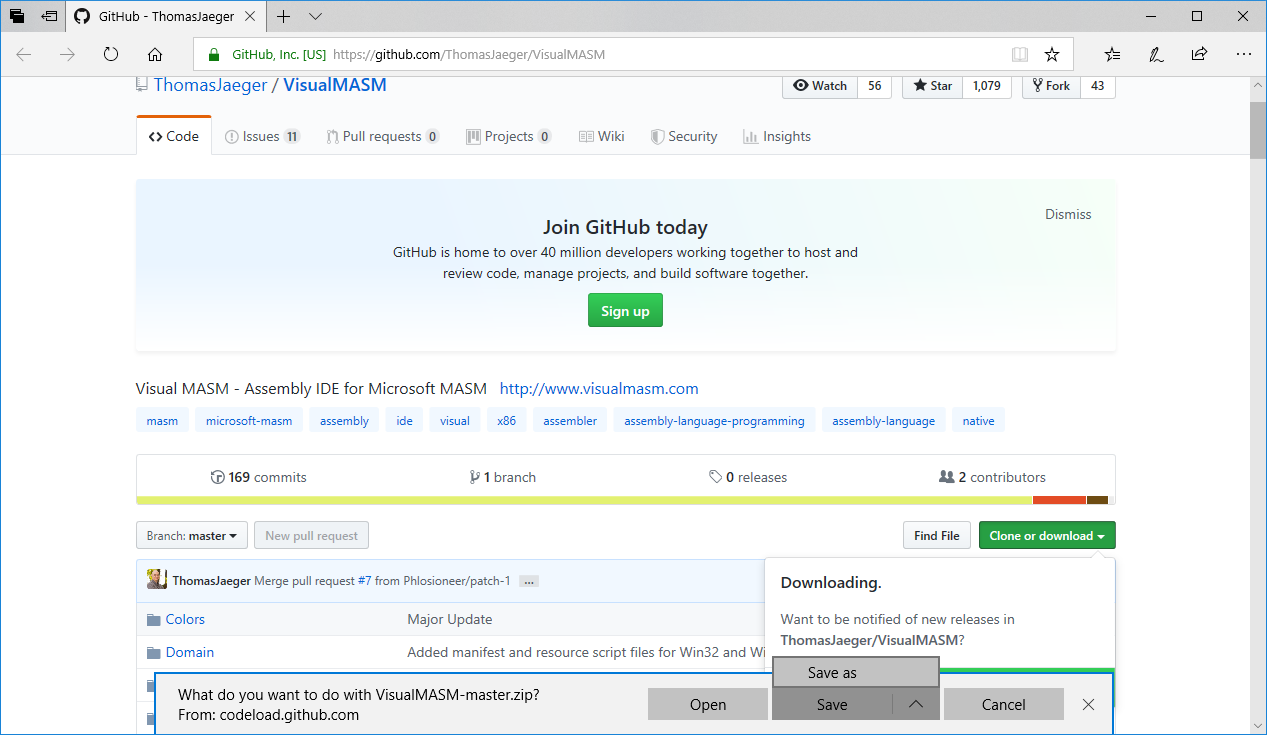
**ALWAYS RUN INSTALLERS AS ADMINISTRATOR!** Any installer may be run “as administrator” when the user right clicks on the installed then selects “Run as administrator from the menu that appears. See the image below:



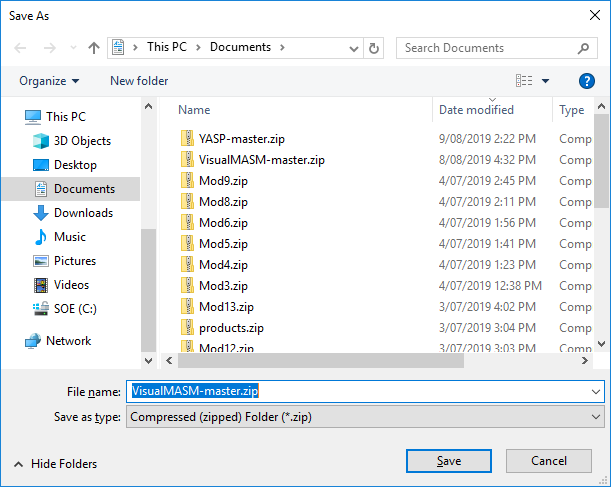
## Using Visual MASM 2.00

To get started with Visual MASM 2.0, from a web browser download and install Visual MASM from <https://github.com/ThomasJaeger/VisualMASM>. See below:

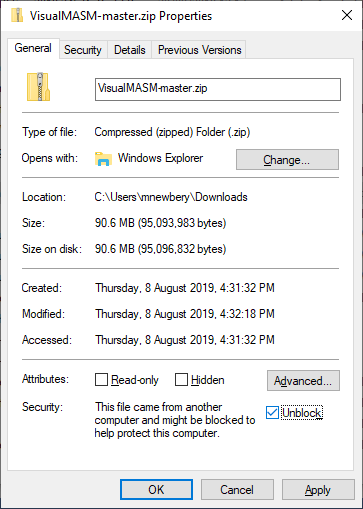




Click on the up arrow (it is actually a chevron but who cares). Save the zip file in the “My Documents” folder on the current computer. If the zip file is saved in the Downloads folder, it won’t unzip properly there! See the next image for an example:

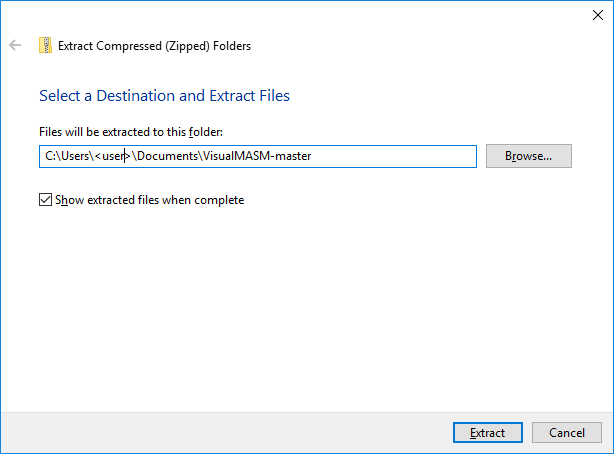


**Always remember to UNBLOCK files that were dowloaded before attempting to unzip or run them locally! See below:**

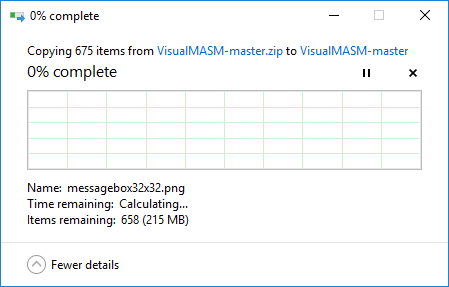


Once the zip file is downloaded to the documents folder, right click on the file to extract the files into the current folder.

Click “Extract all” to continue. See the new dialog below then click “Extract” to continue:

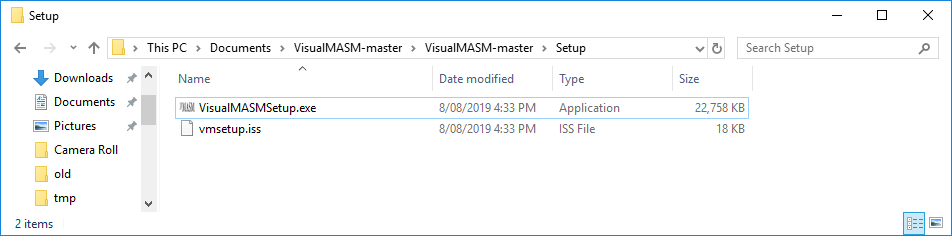


Wait for the extraction to complete. See below:



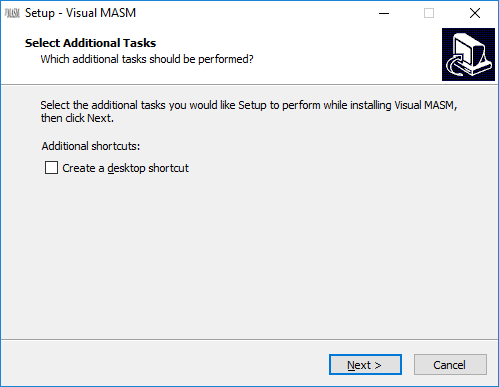
**This step could take up to an hour!**

Once the files are extracted, navigate to the Setup folder and **right-click** on the file then choose run **VisualMASMSetup.exe** **AS ADMINISTRATOR**. See below:

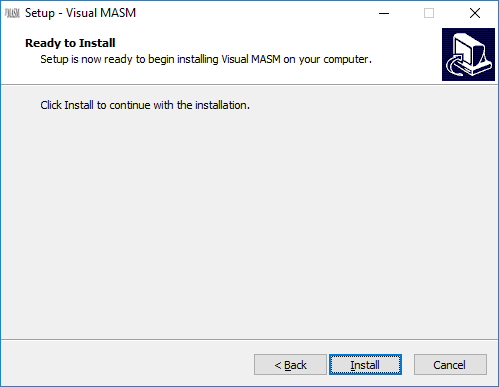


Click “Yes” on the security dialog to begin the installation.

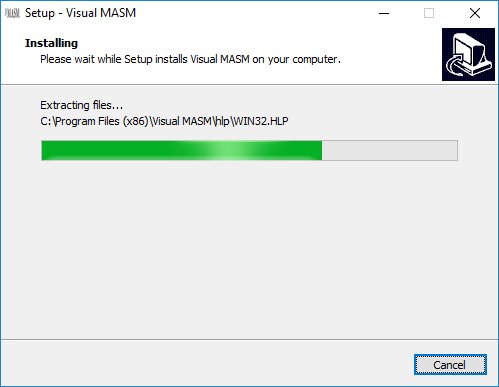
Click “Next” on the dialog below to continue installation:



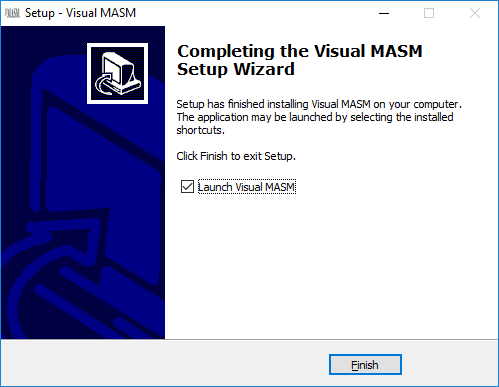
Click “Install” on the dialog below to continue the installation:



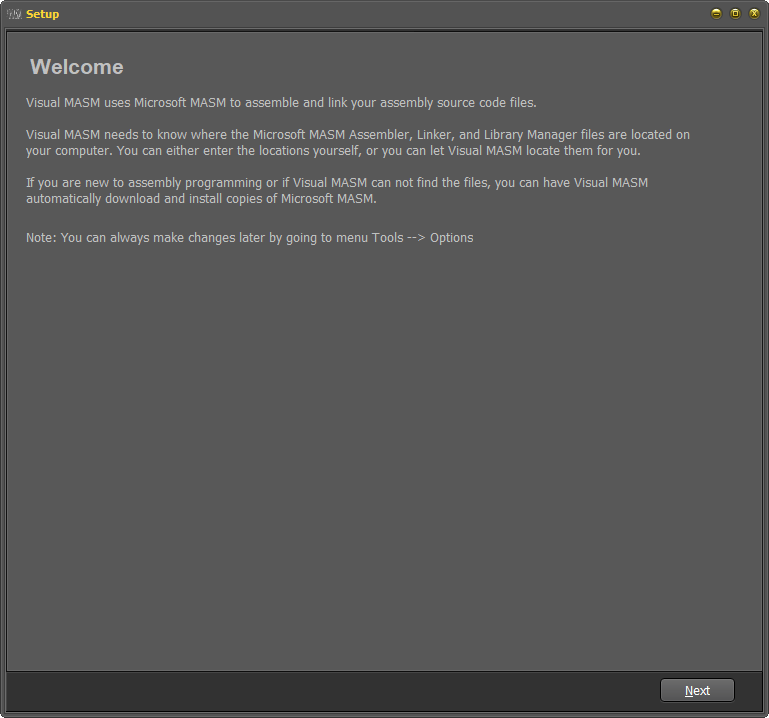
Wait for the installation to complete. See below:



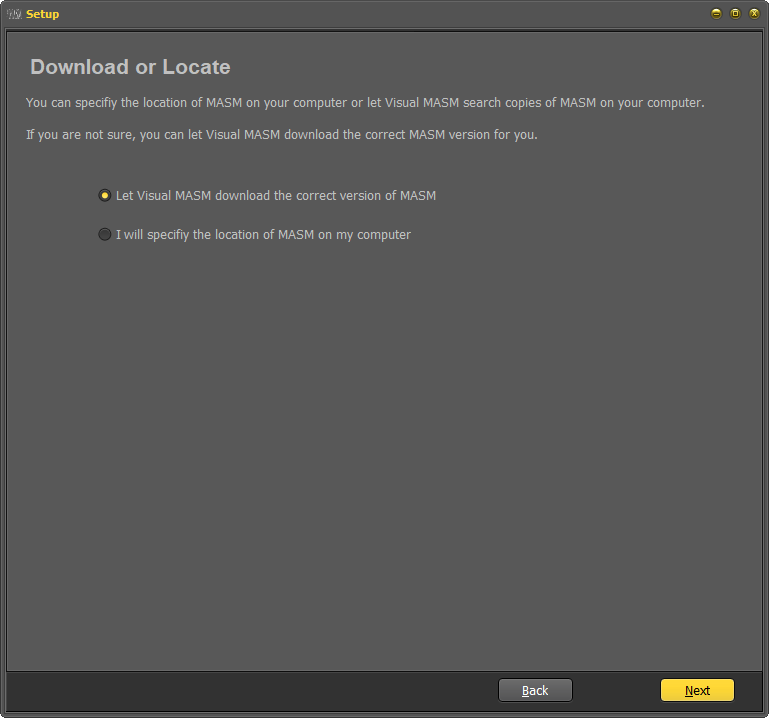
This is the final dialog of the installation. Click Finish to launch Visual MASM:



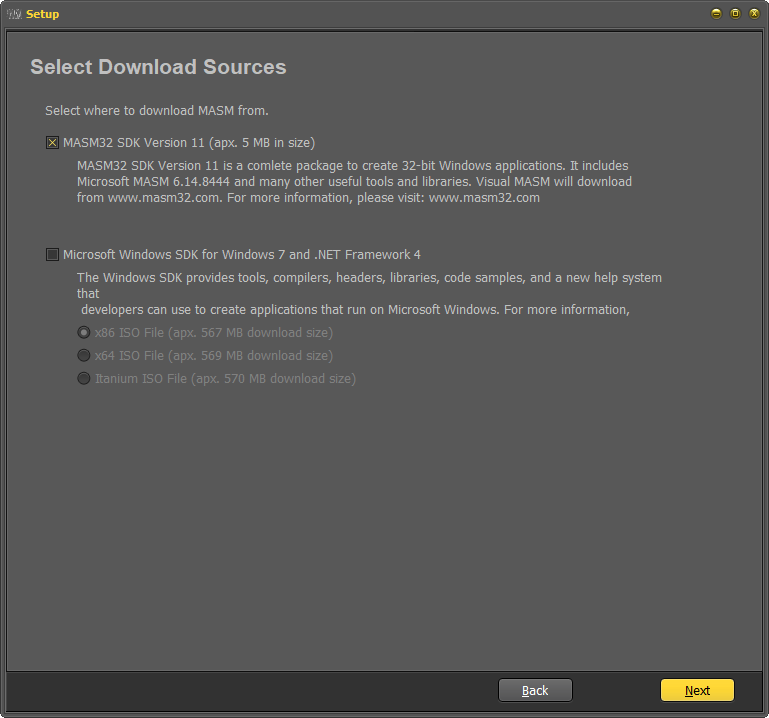
Below is the first screen of the Visual MASM Integrated Design Environment. The first thing it will do is download the correct framework for the current installation. Click Next to continue:



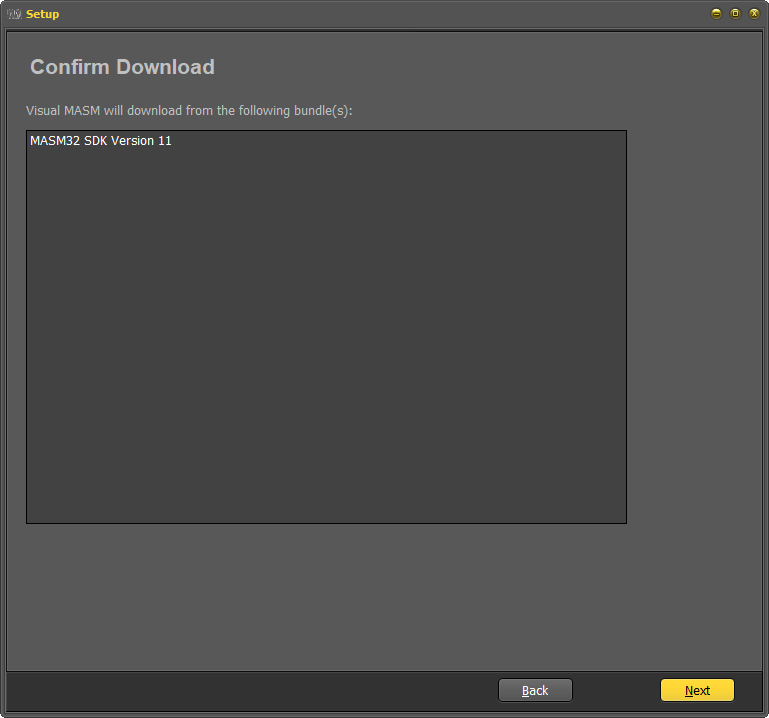
On the screen below, use the default settings to continue the setup then click next to continue. See below:



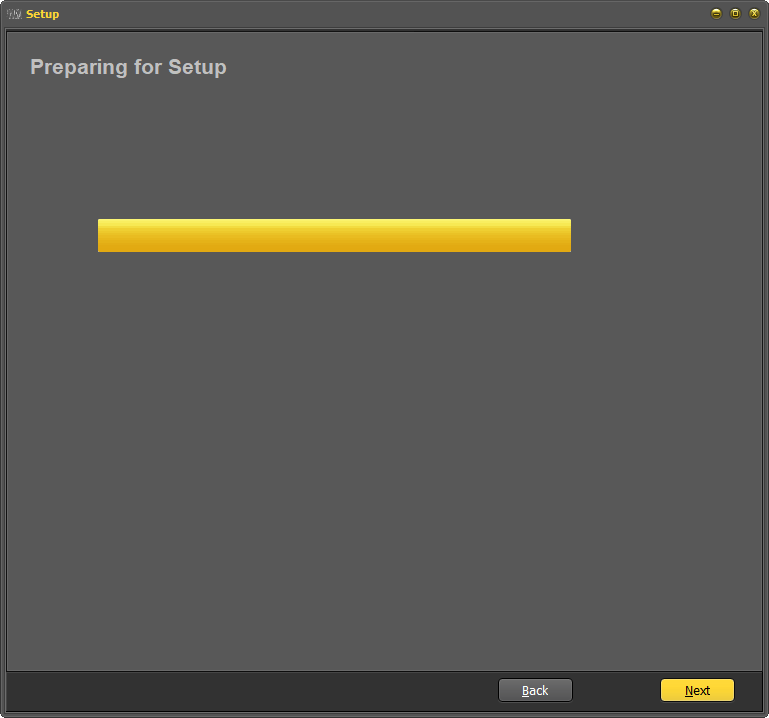
In the screen below, use the default settings if the target machine has an internet connection. Click “Next” to continue. See below:



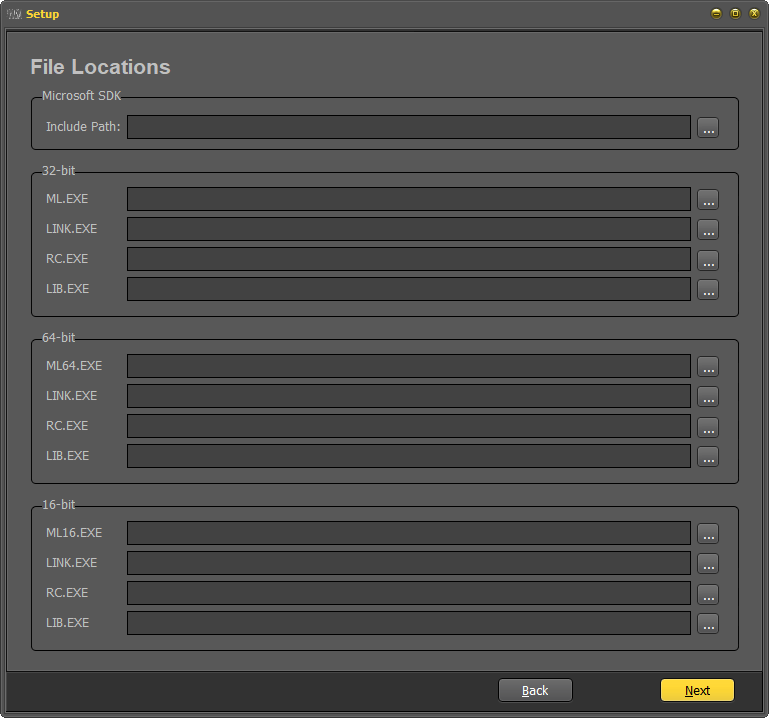
Click “Next” to confirm and continue. See below:



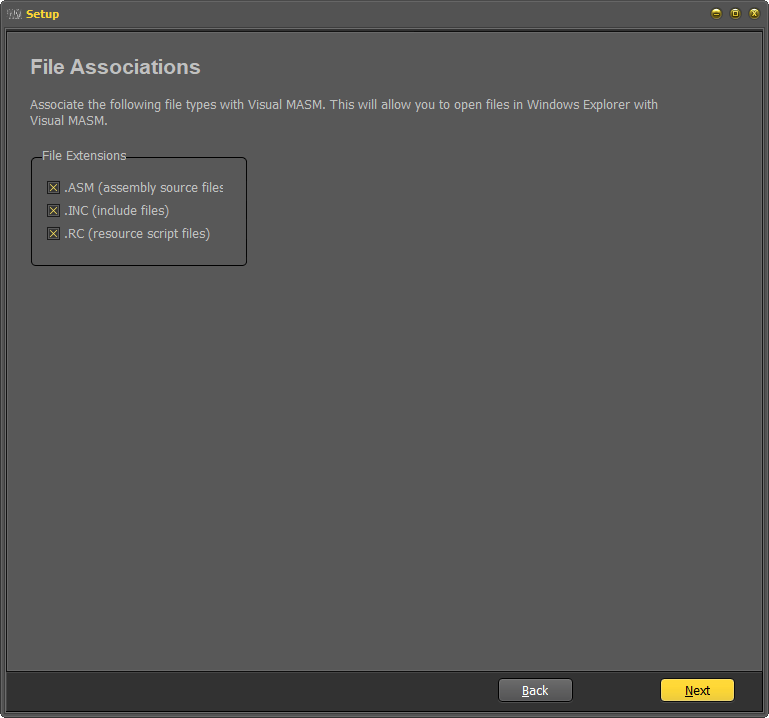
Click “Next” to continue and accept all prompts (that is, click “Next”, “Yes” or “No” as appropriate) until the setup is complete.



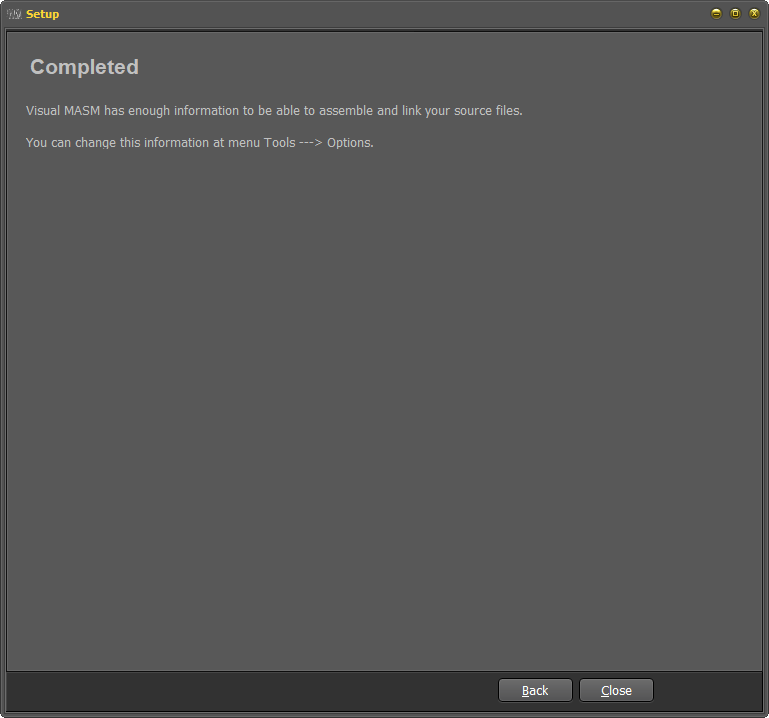
These values to not need to be populated straight away. Click Next to continue. See below:



Use the default selections in the example below then click “Next” to continue setup. See below:



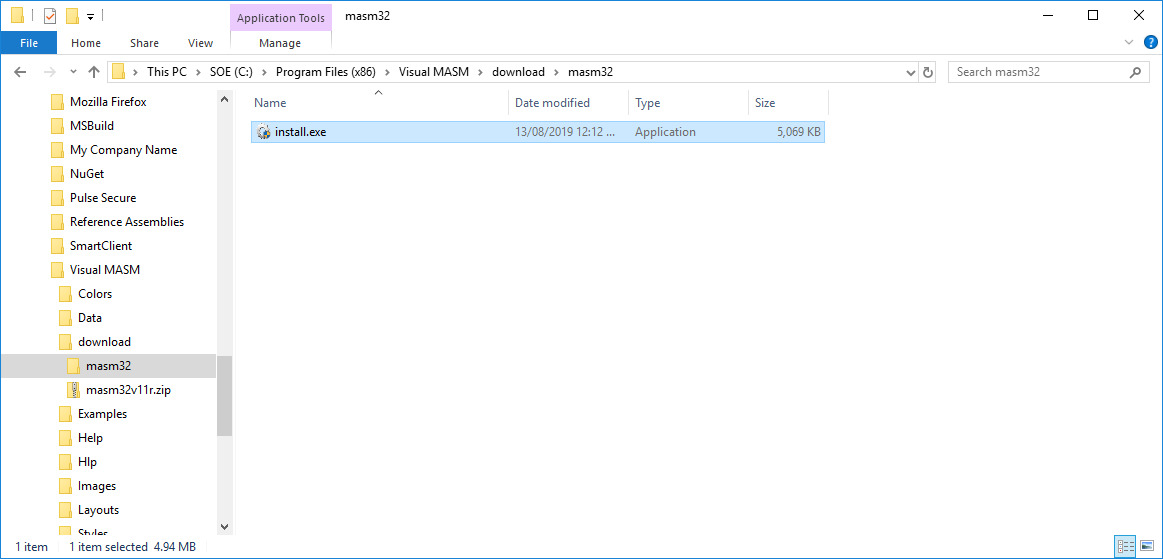
In the screen below, click “Close” to complete the setup:



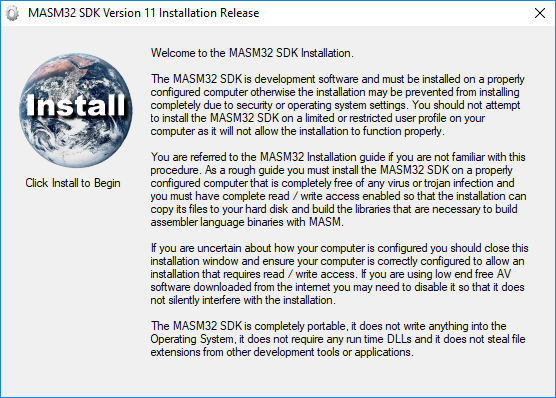
Visual MASM will now launch! This may fail if the launcher does not **RUN AS ADMINISTRATOR!**



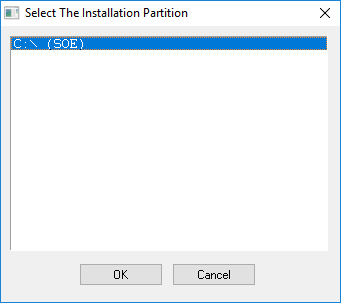
Extract the samples from the MASM32 demo code and MASM32 Software Development Kit (SDK). Right click the file then choose **RUN install.exe AS ADMINISTRATOR** to start the installation. See below:



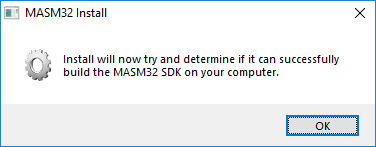
The button in this image looks like a globe. Click the button to continue. See below:



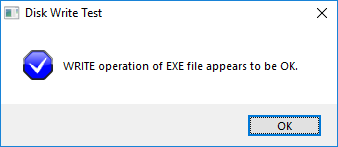
Use the default selection by clicking on the line “**C:\ (SOE)”**. This will install to c:\MASM32. Click “OK” to continue. See below:



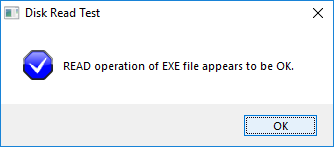
In the screen below, click “OK” to continue:



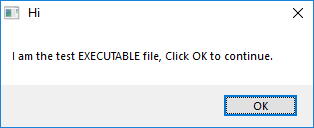
In the screen below, click “OK” to continue:



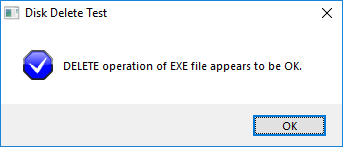
In the screen below, click “OK” to continue:



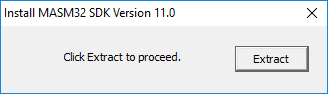
In the screen below, click “OK” to continue:



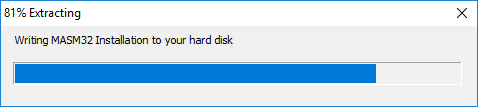
In the screen below, click “OK” to continue:



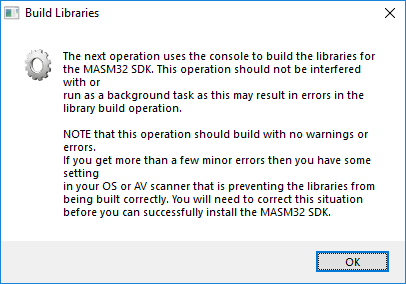
In the screen below, click “Extract” to continue:



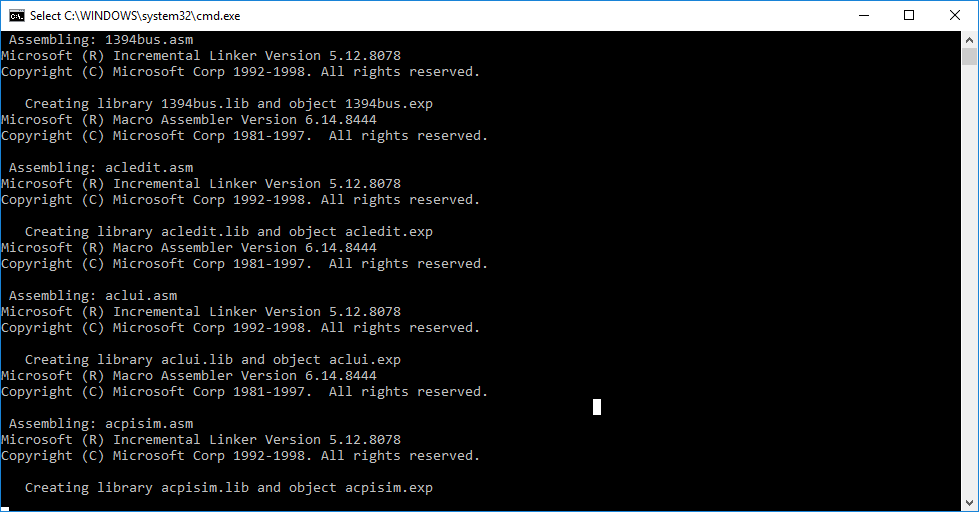
…then wait a minute…



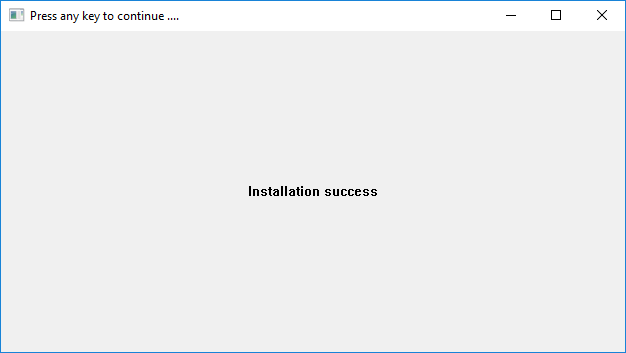
In the screen below, read the warning then click “OK” to continue:



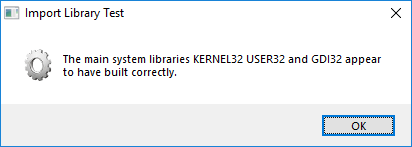
… then wait up to an hour while the libraries are created from their assembler source… If the process pauses, set focus to the window below then tap the space bar.



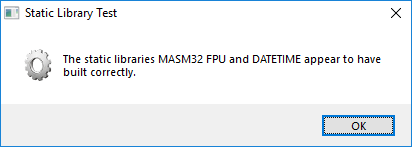
Once this dialog appears, tap any key to dismiss it



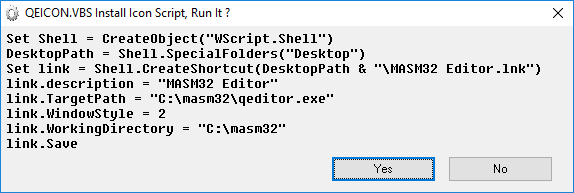
In the screen below, click “OK” to continue:



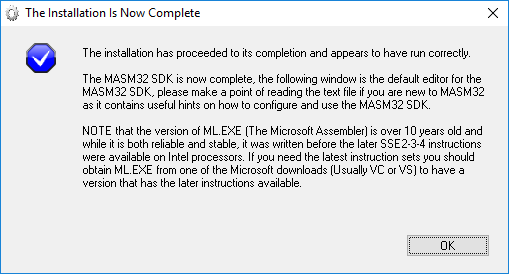
In the screen below, click “OK” to continue:



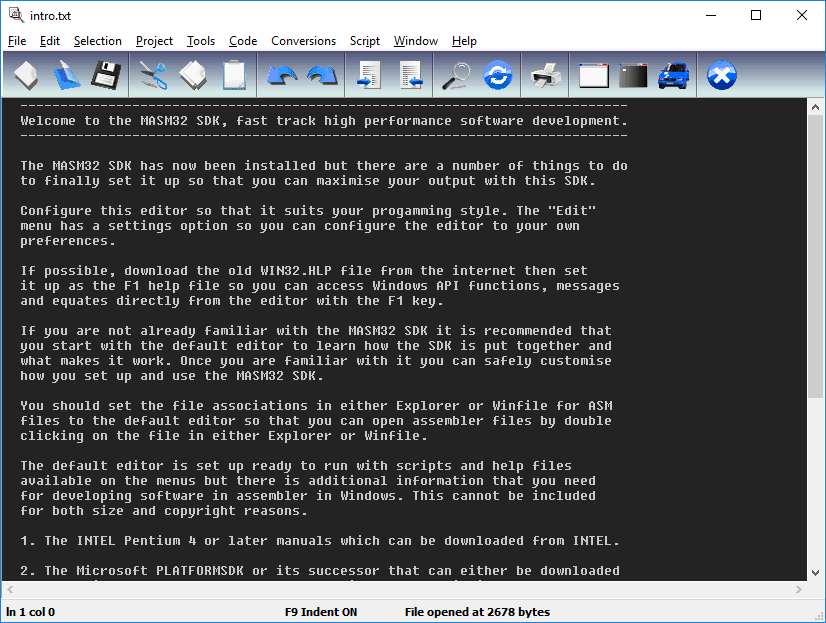
In the screen below, click “Yes” to execute the script:



In the screen below, click “OK” to continue:



This last step will *finally* launch the **simple** MASM editor. See below:



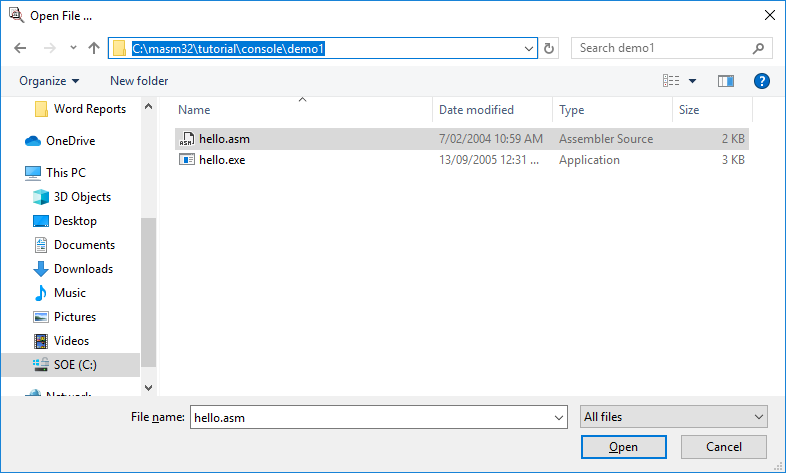
Note: At this time the **MASM32 editor** is open so it is very easy to navigate back to the application window. Once the application is closed, there should be a shortcut for the **MASM32 editor** on the Desktop.

The path in the shortcut is **c:\masm32\qeditor.exe**

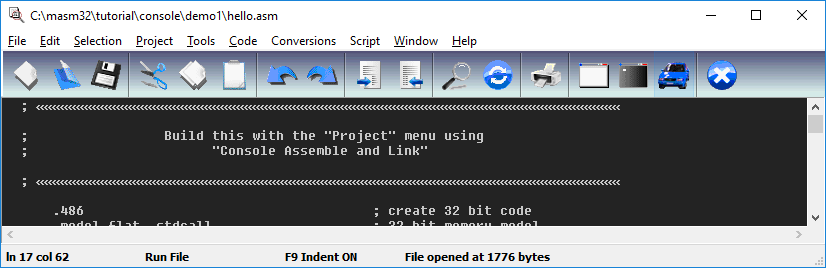
Visual MASM 2.00 does not install its own desktop shortcut but that isn’t being used yet.

**Stop!**

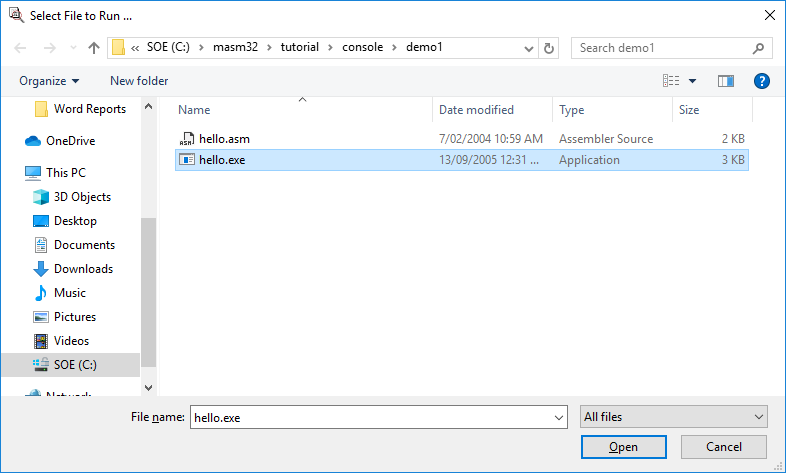
To test out the IDE **in the previous image**, open a sample file. Below is a simple demo:



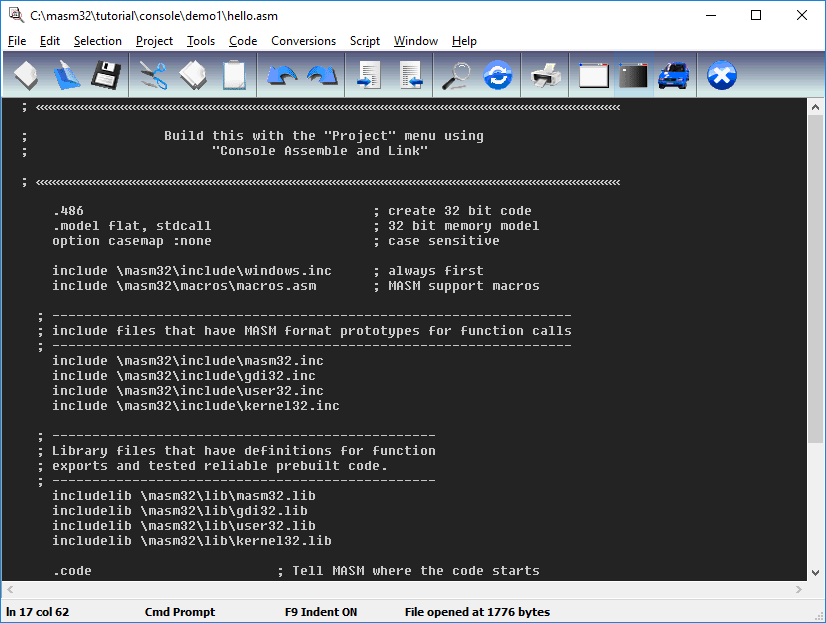
Once the ASM file is opened, it can be compiled and/or run from the editor. Click on the **Car** icon to run the executable file (the executable file already exists in the demo). See below:



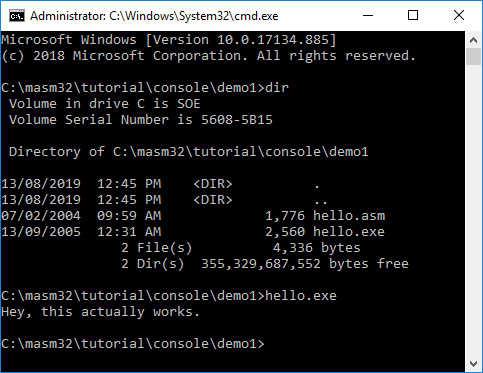
Select the .exe file to run. This will cause a terminal window to flash open momentarily then close. **Try it now**. See below:



To make things easier, click on the Command Prompt icon to open a command window in the path of the executable. Note how the title of the icon appears in the status bar at the bottom of the dialog. See below:

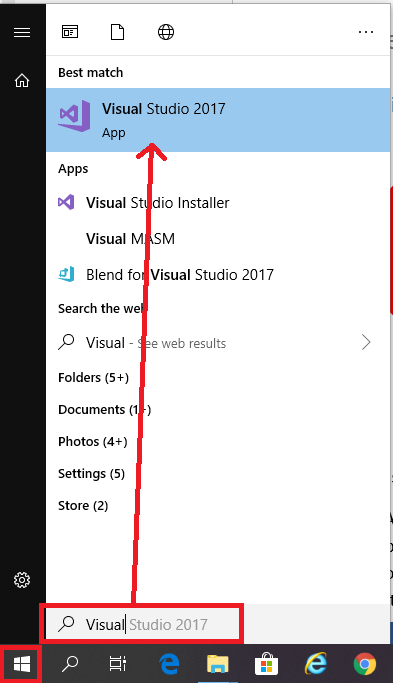


Below is the output of the command window. A number of commands are executed **in the example below**, along with the output. Note the output from the executable below then go back to the ASM file and find the text!



## Using Visual Studio 2017

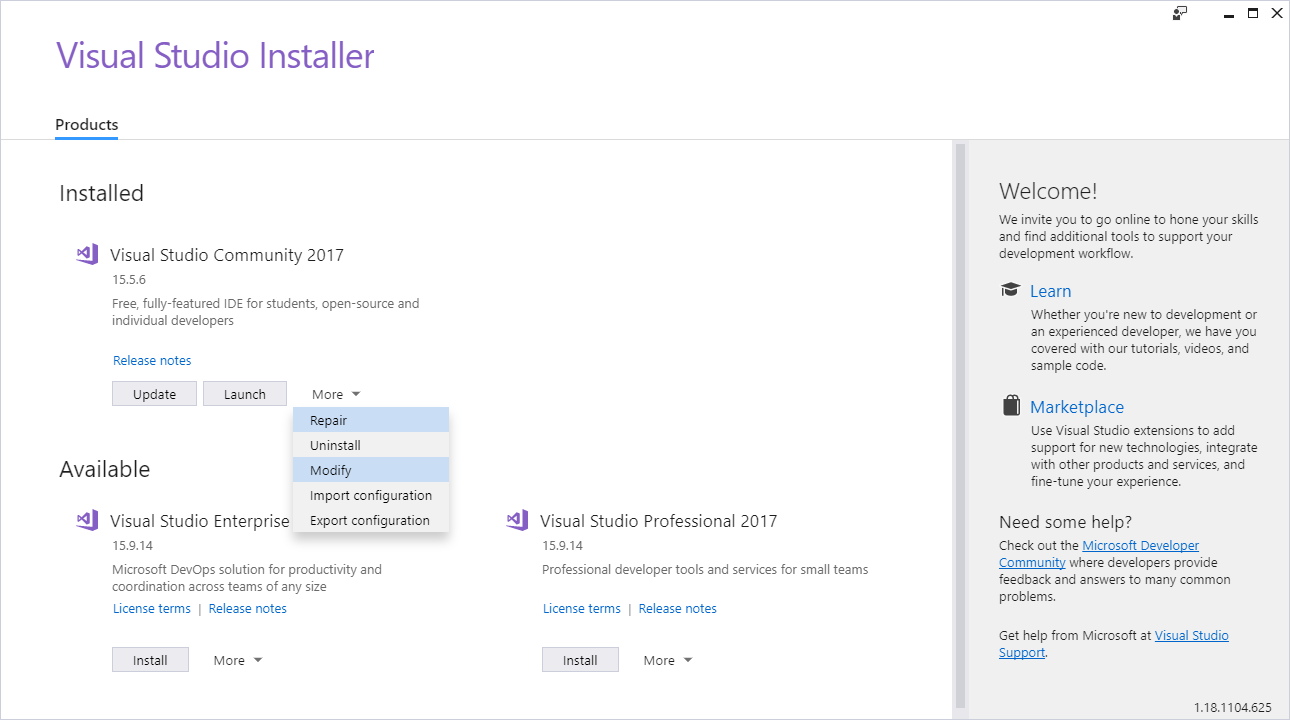
Note: Visual Studio 2017 may not have a desktop shortcut. In order to start an application from the desktop, click on the Windows icon (or tap the Windows key on the keyboard) then start typing the name of the application of interest. If it is installed it will appear in the list results. See below. Also note Visual MASM appears in these search results:



**Stop!**

This section is for readers who already have Visual Studio installed.

MASM is still and has always been an extension of the Visual Studio C/C++ toolset. So there is no separate project system for it. Just ensure that when installing Visual Studio for the first time, C++ projects are selected in the installation options. Alternatively if Visual Studio is already installed, run the installer again with the updated selection. Below is the Visual Studio updater/installer window:

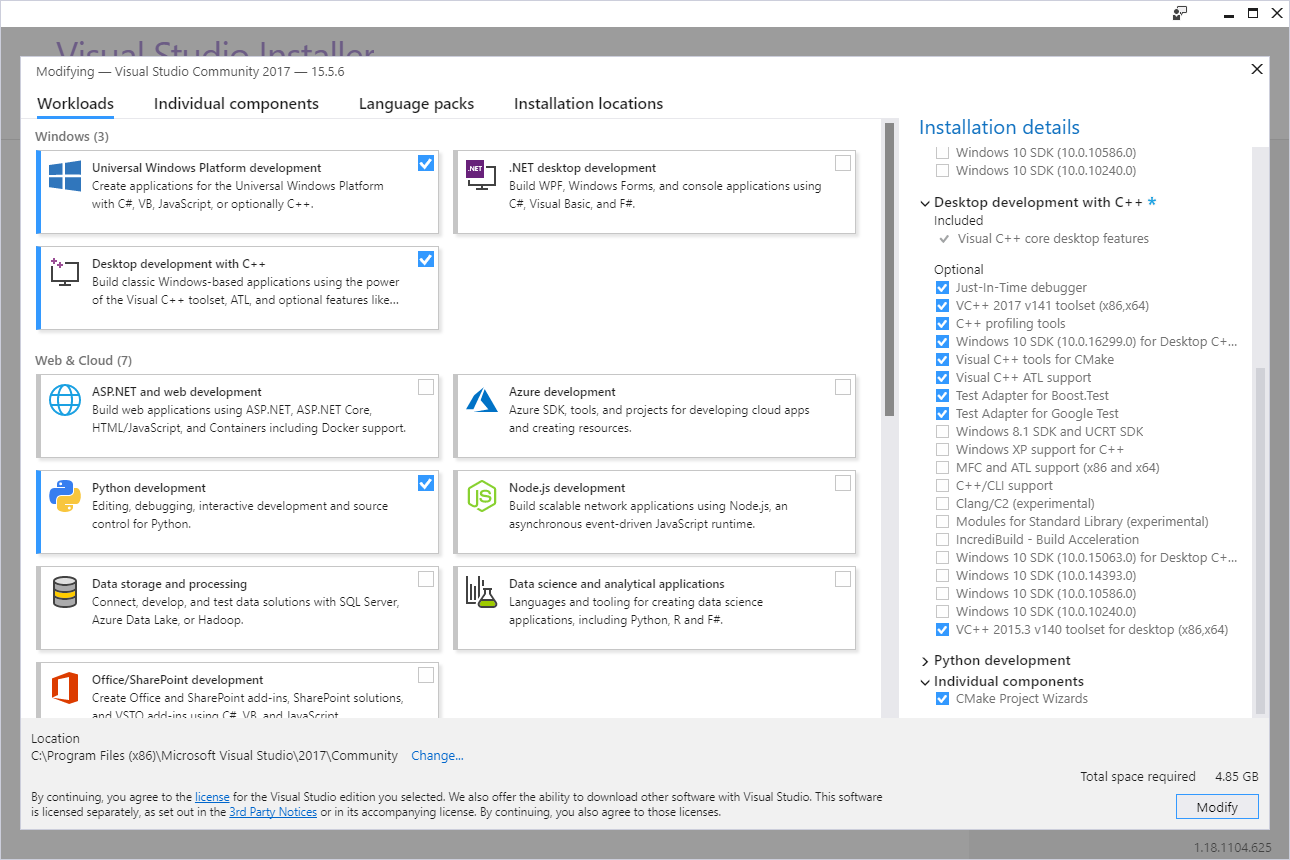


To get support for it in a Visual Studio project, install the C/C++ support as per the installer settings in the next image, create a new C/C++ project and when it has been created you must then do one more step. Open up Solution Explorer, right click on your project, find Build Dependencies hover over it and then select Build Customisations. In the window that appears, check MASM and select ok. This setting is covered later in the document.

This will now build any .asm file in the project with ml/ml64 depending on your project configuration. But there is still no project item for .asm files. To add these you can simply select new C++ file (.cpp) in the add new item window and when you name it, **make sure you give it a “.asm” extension.**

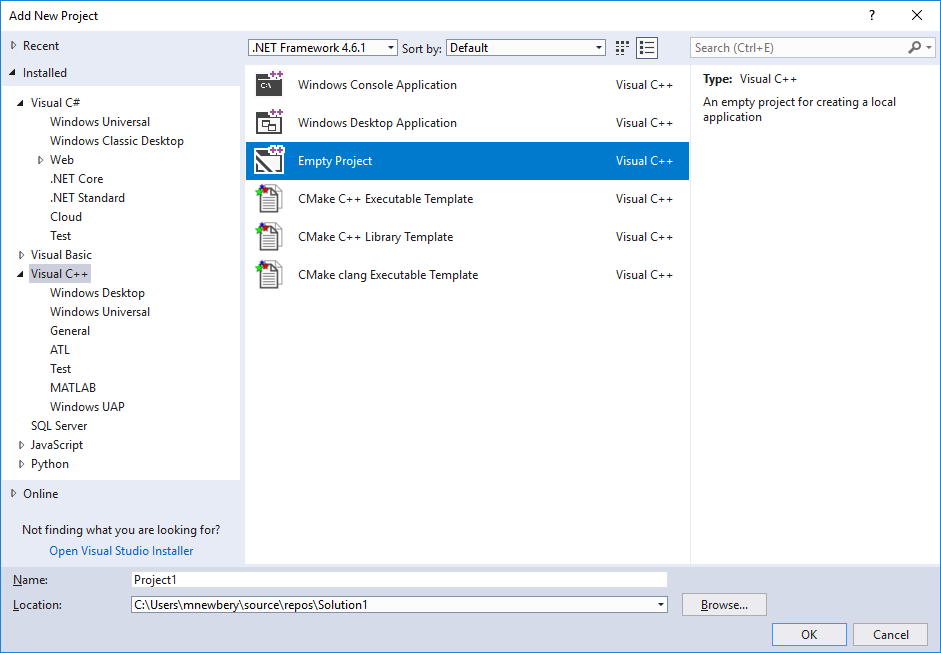
If you added any .asm files to the project before you enabled MASM support, these .asm files will not automatically be changed to build with MASM, you will need to change them over yourself. The simplest way of doing this is to just remove them from the project (being careful not to delete them) and then add them again.

Below is a screen grab of the Visual Studio 2017 Installer settings required to build executable content from MASM files, as in the image below:

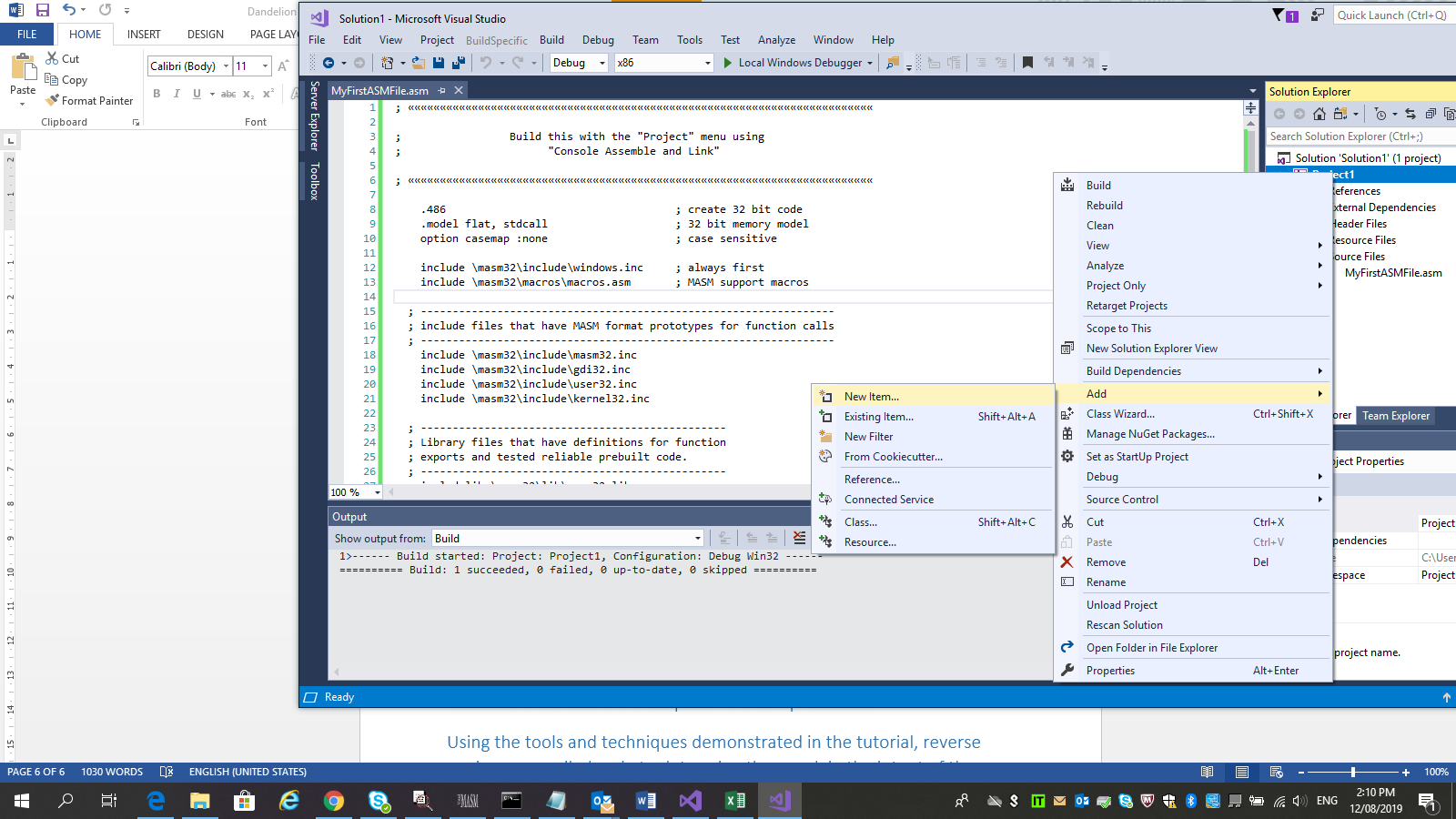


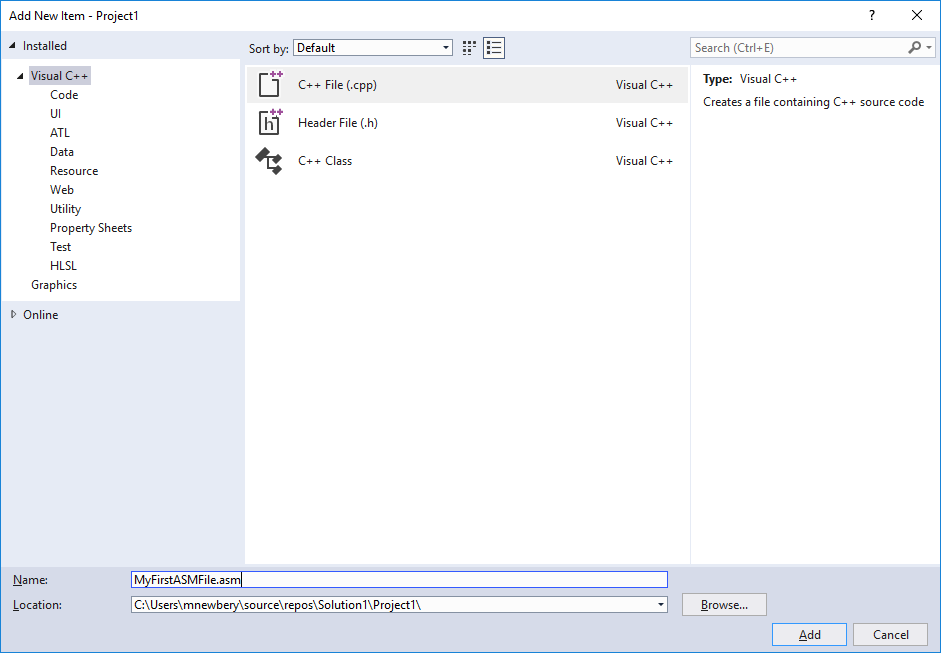
The section below assumes Visual Studio 2017 is being **RUN AS ADMINISTRATOR.** If not, run Visual Studio **as Administrator now**

Below is the window that shows a new project being added before a MASM file is created or included:

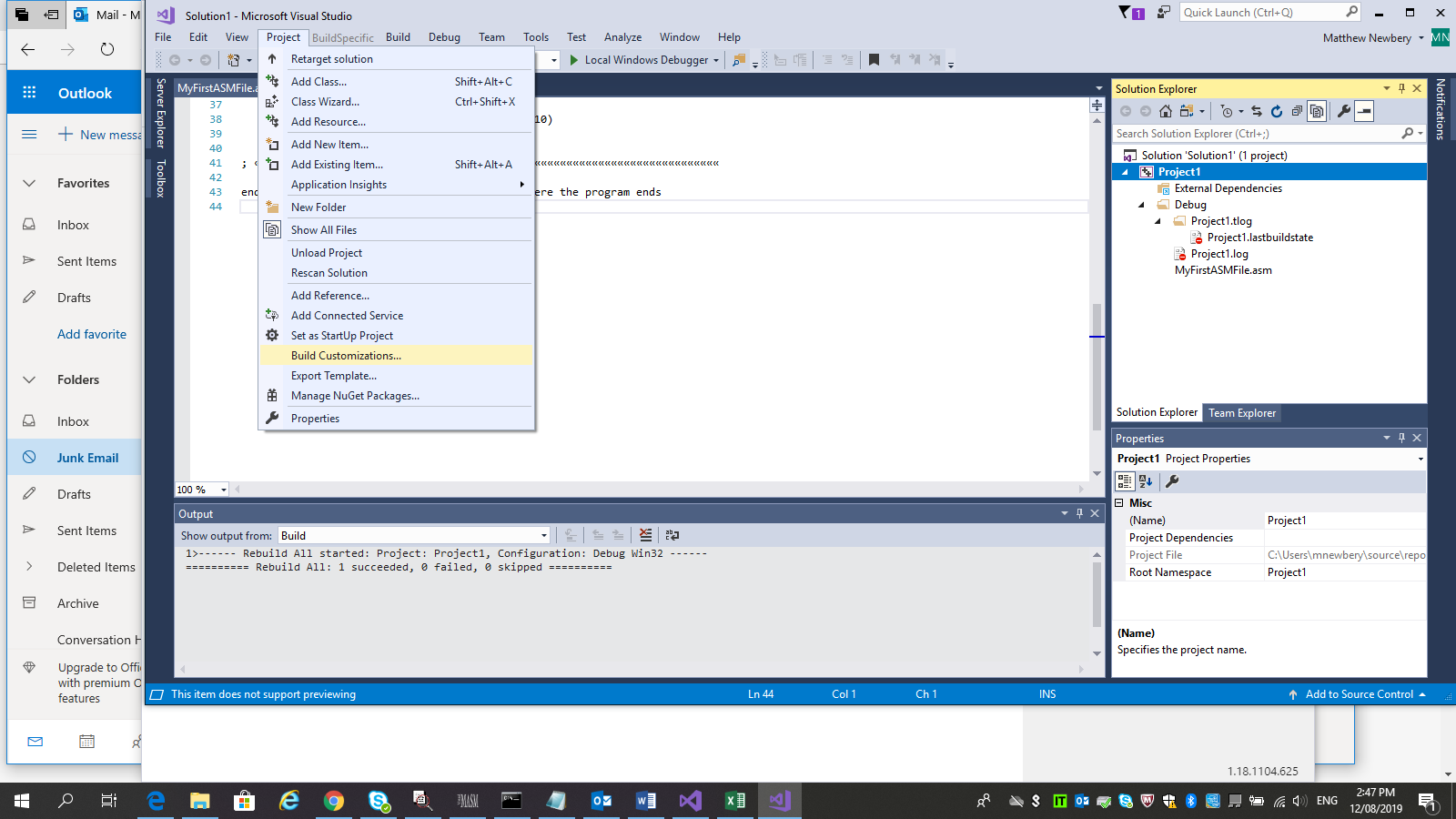


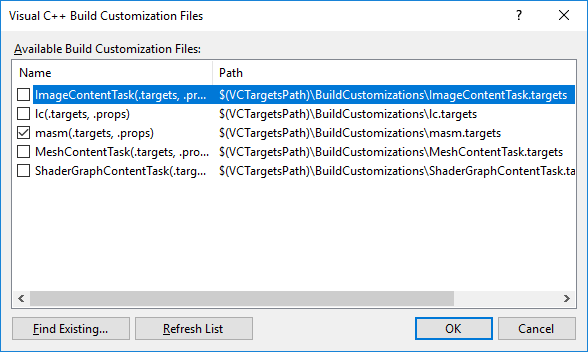
Below is the “Add New Item window” with the first ASM file being added to the project. Create a new project item in the project with the Project->Add New Item menu. Follow these instructions to add a new file with the .asm file extension as in the example below:



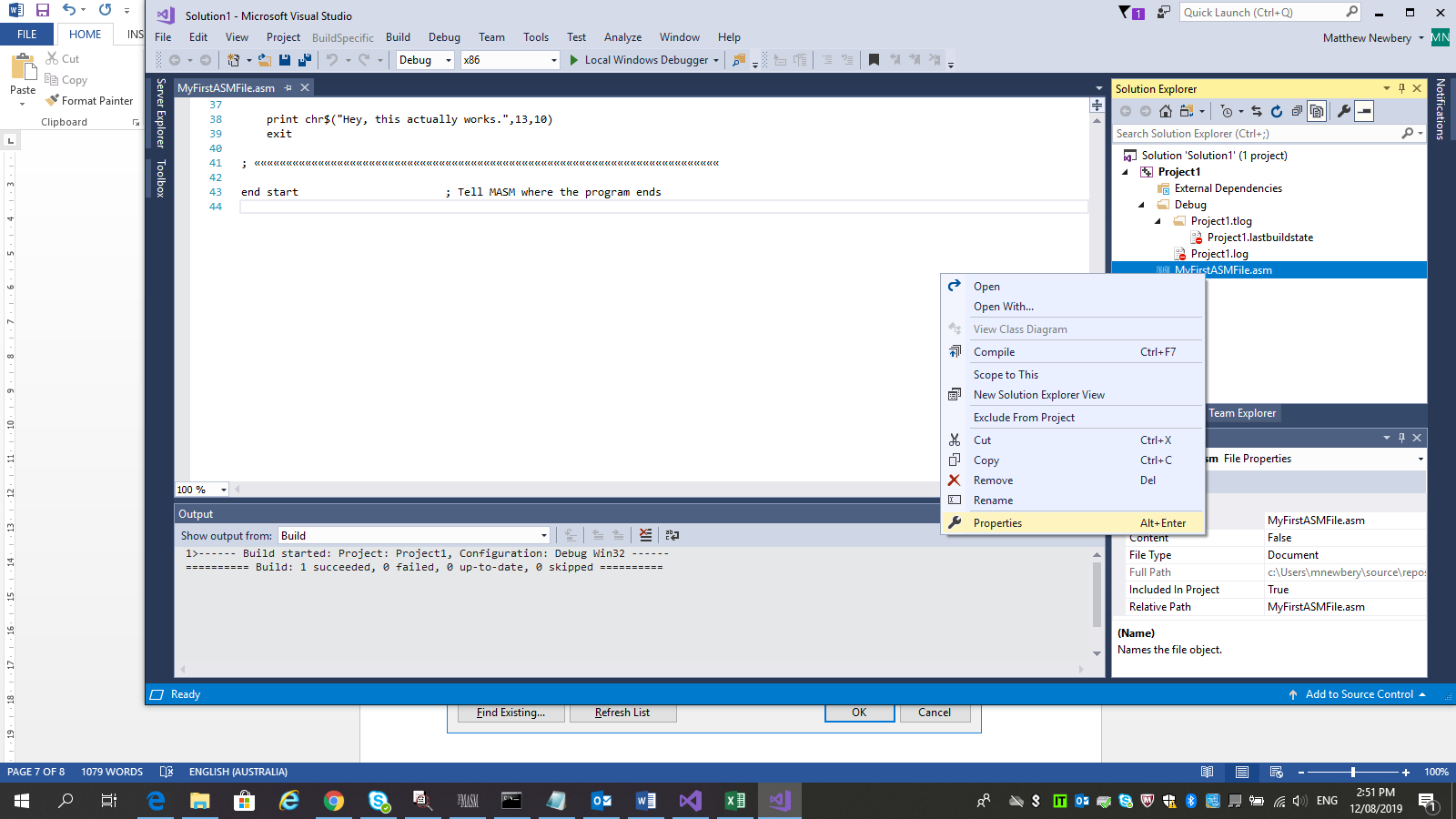


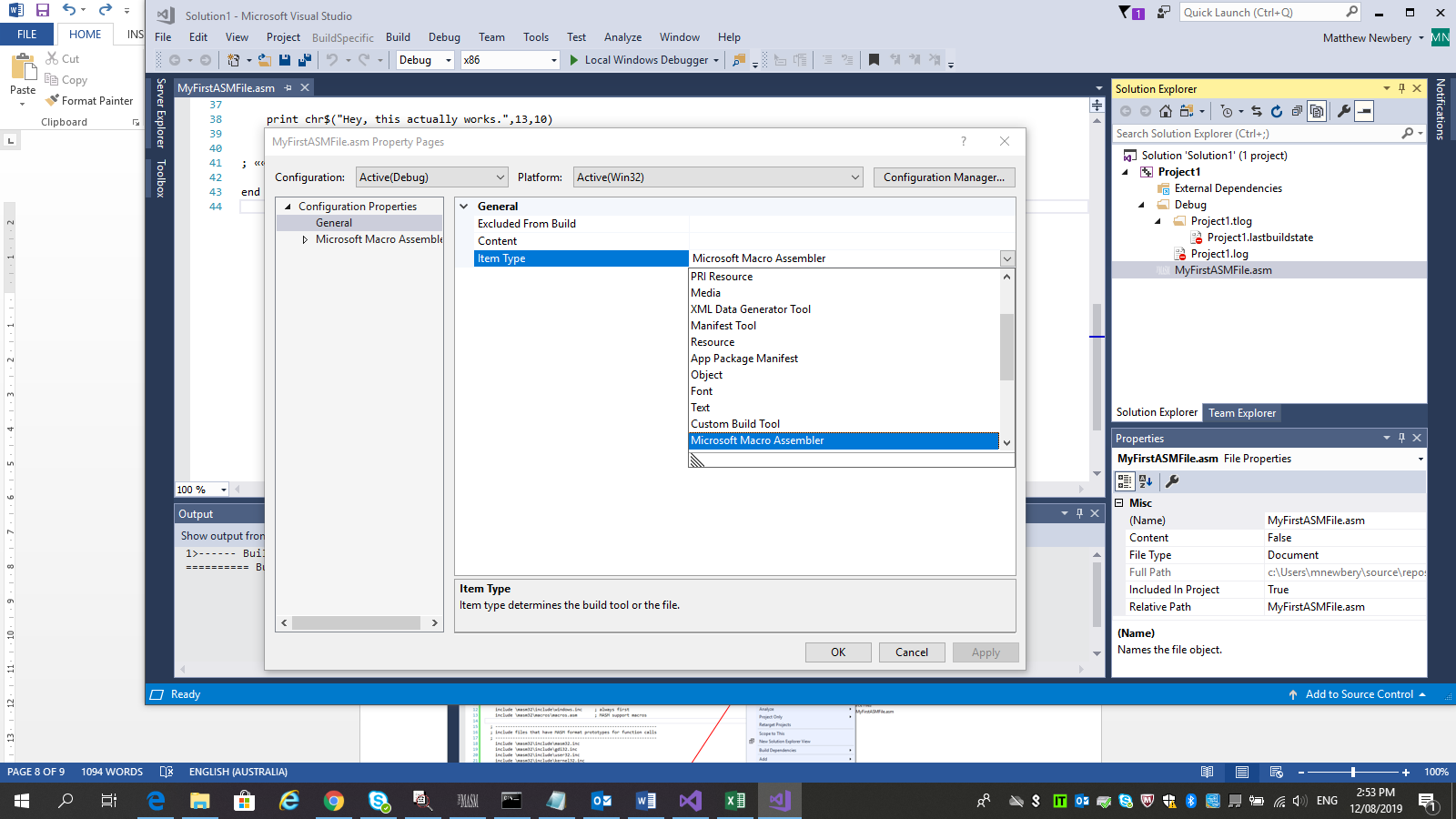
Follow these graphical instructions to set the build customisations to include the correct processing of ASM files. See the image below:





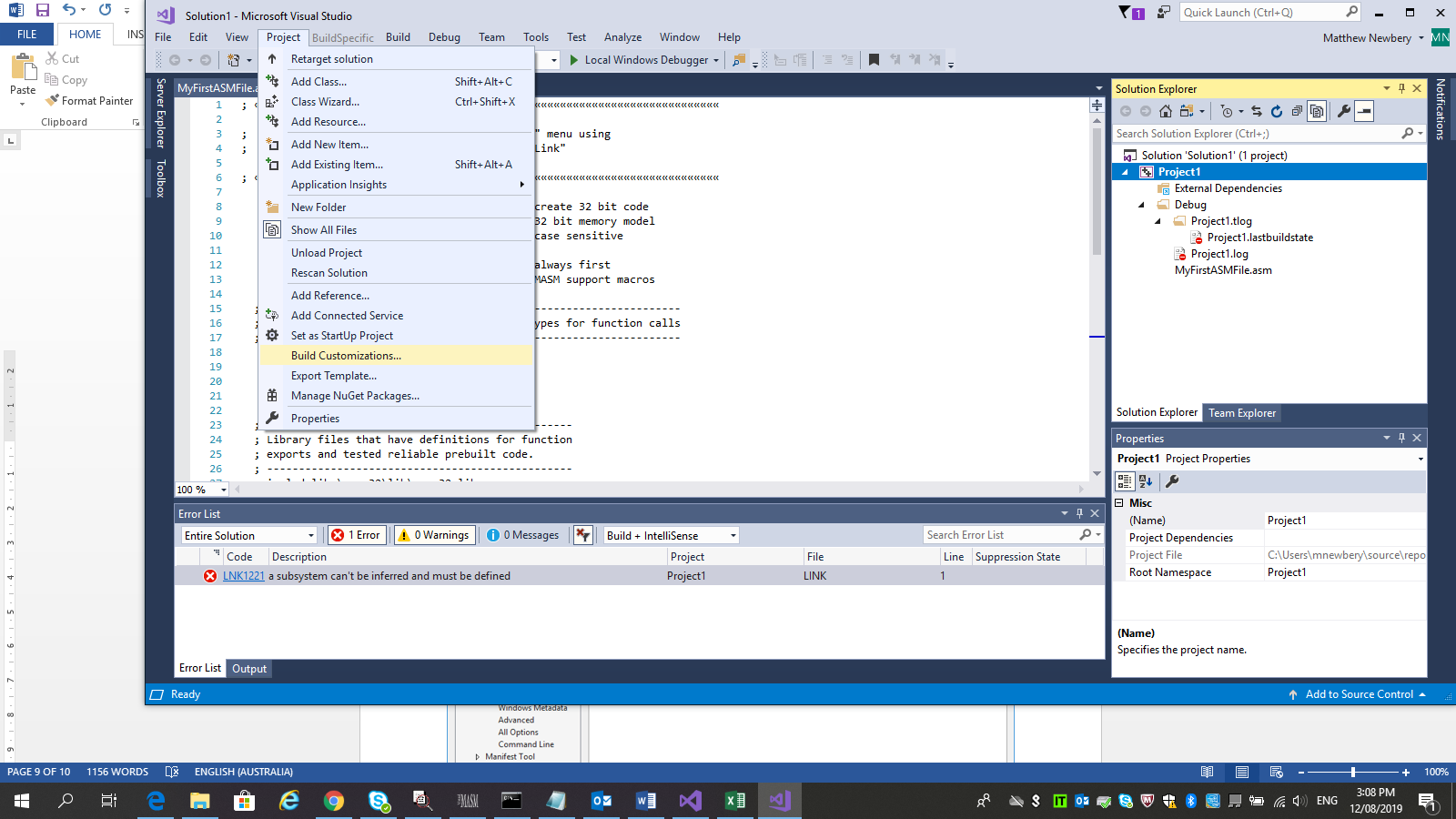
This image shows how to change the way the project processes the new ASM file. Start by right clicking on the new ASM file then selecting Properties from the menu that appears:

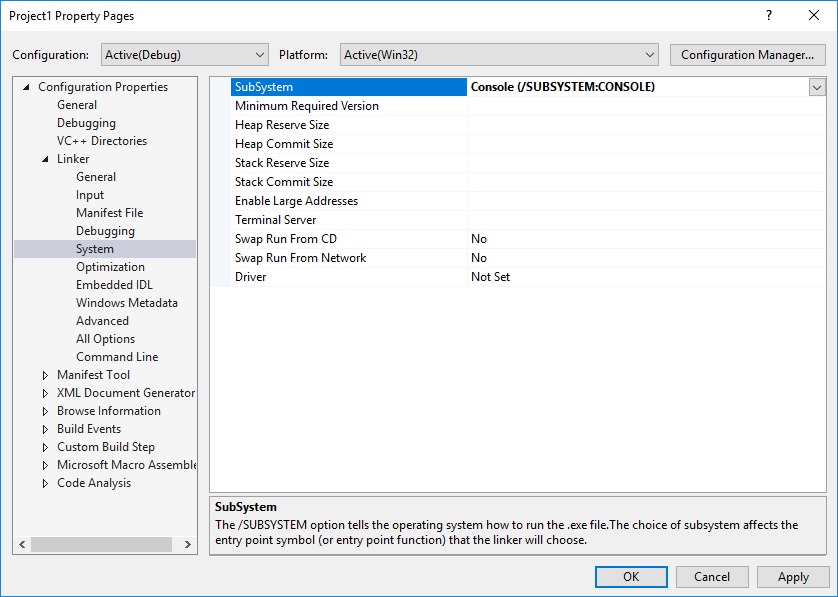




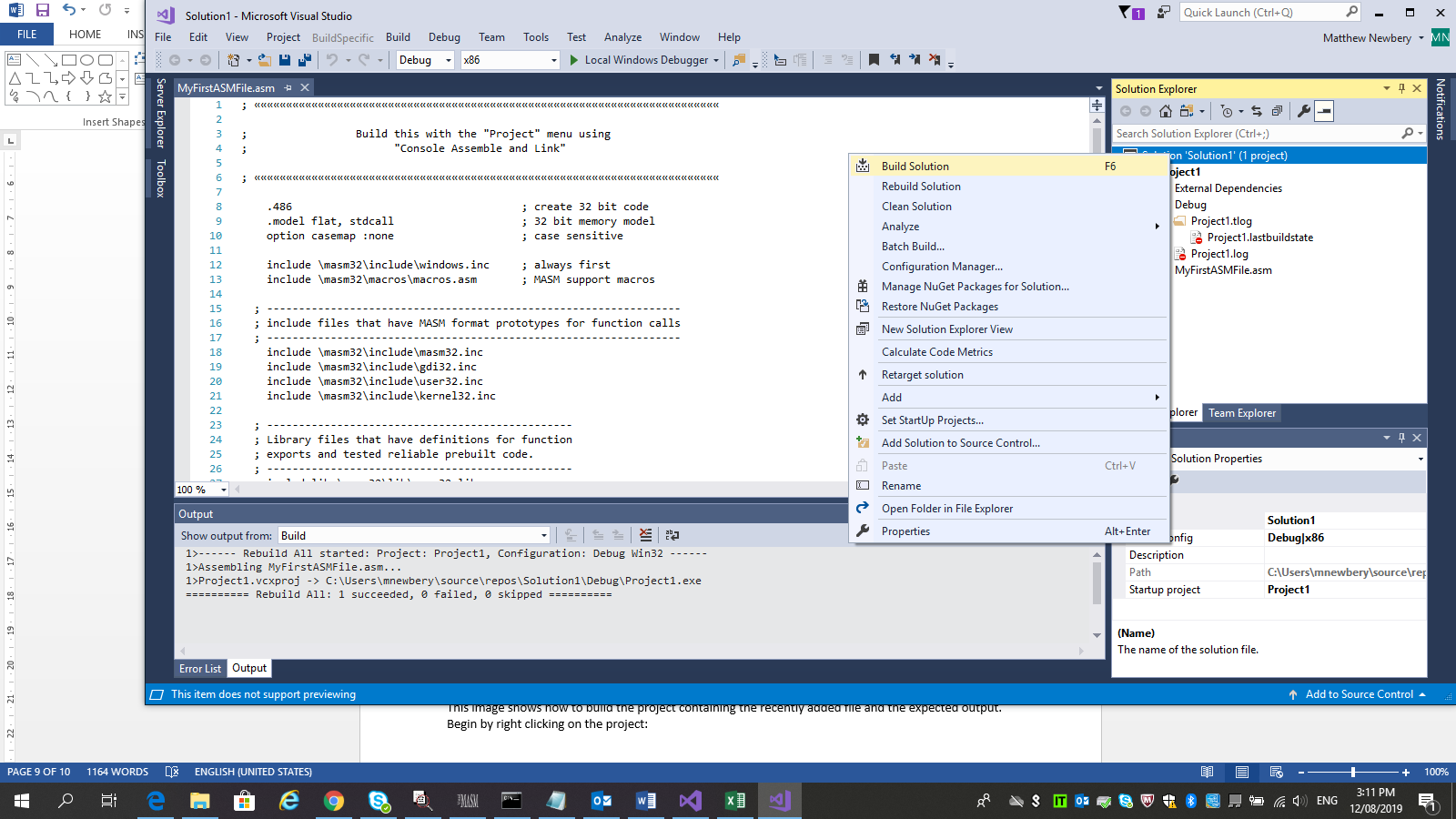
Note: If there is a requirement to **not** compile an assembly code file, setting the file to be text is one easy way to achieve this. See the image above for the **text** option!

This image shows how to change the compiler subsystem target (Note: this step is done automatically when the user chooses a new C++ console or Windows project). Start by viewing the main menu at the top of Visual Studio, select Project from the main menu then select Properties from the menu that appears as in the images below:





This image shows how to build the project containing the recently added file and the expected output. Begin by right clicking on the project or the solution then selecting “Build Solution” or “Build project” from the menu that appears. Note, this image appears when the solution explorer is viewing the project (this is the default), not the folder structure. Note the expected text in the output window in the image below:



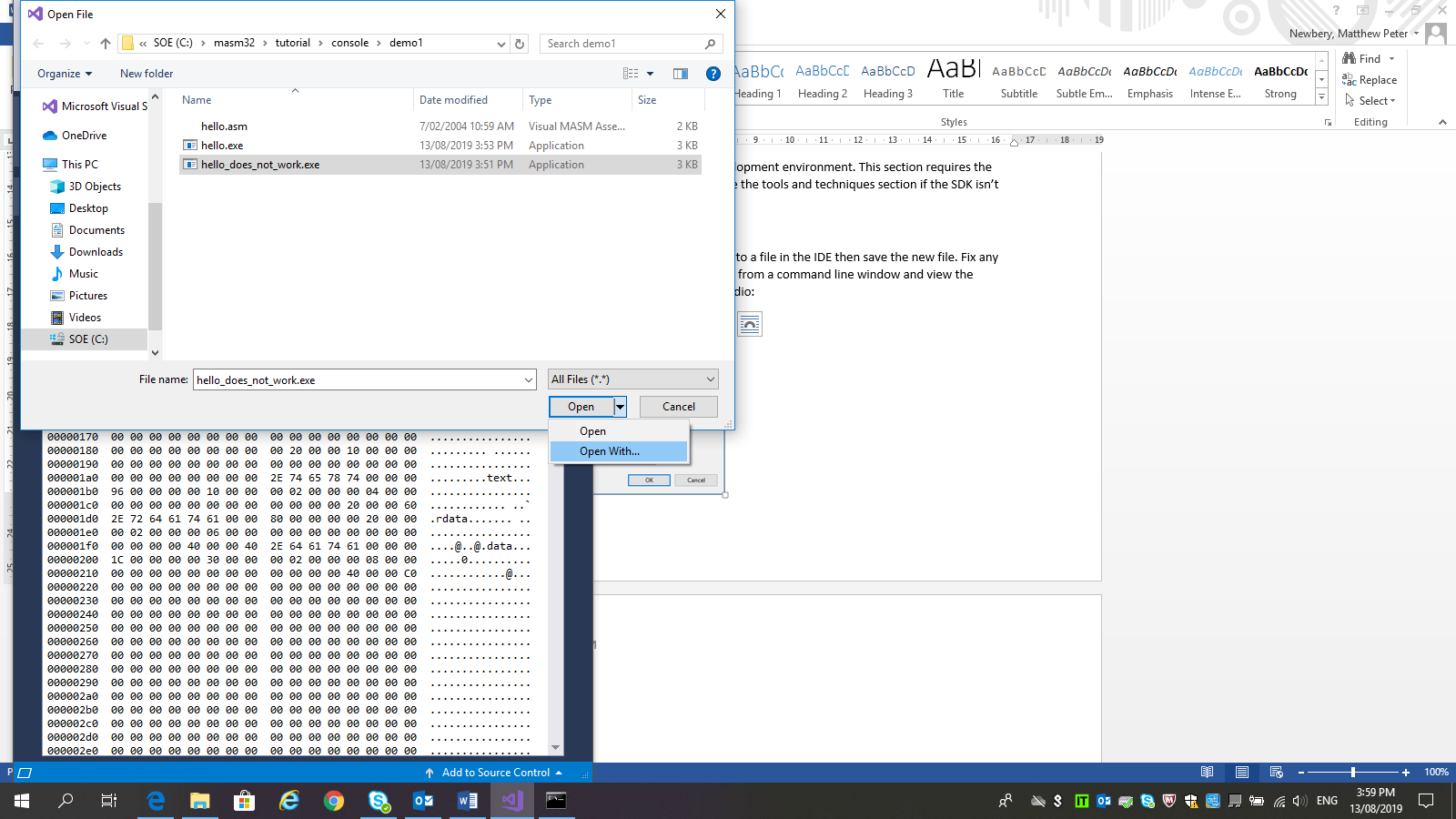
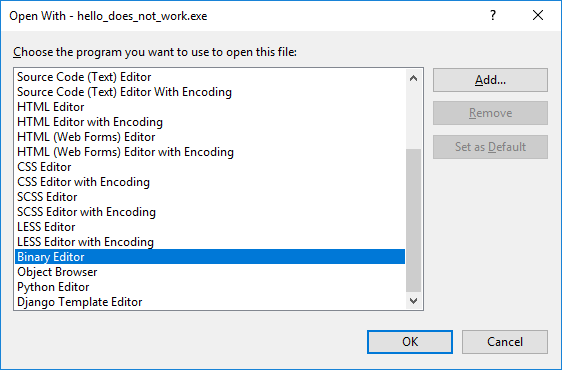
Once the code is compiled, the .exe file can now be found in the file system. If desired, run the executable from a command window. Sample code appears in Appendix A

# Create and compile trivial programs that can run on a Windows machine

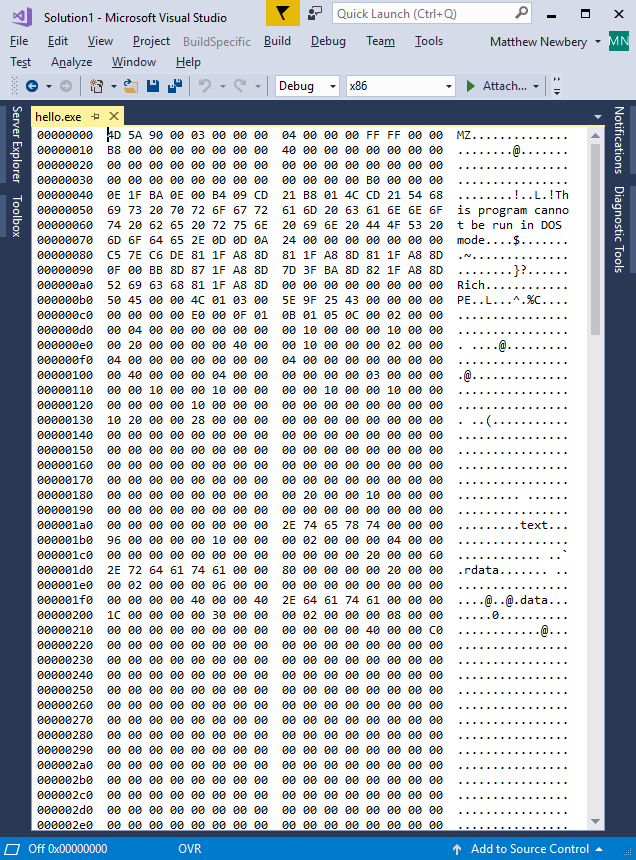
Instructions in this section are the same for any development environment. This section requires the MASM SDK to already be installed in c:\MASM32. See the tools and techniques section if the SDK isn’t already installed with Visual MASM 2.00.

## Demo 1

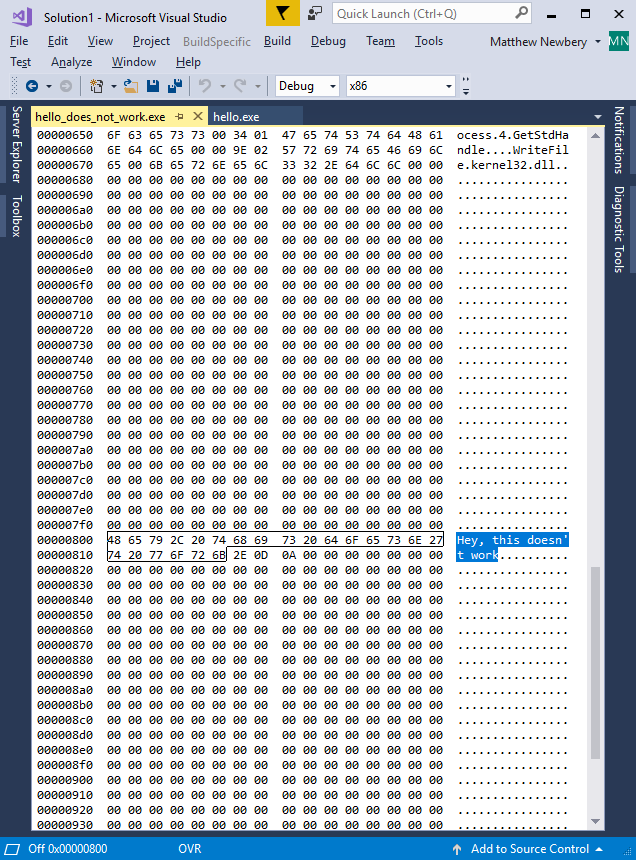
Copy and paste the code from Appendix A, “demo 1” into a file in the IDE then save the new file. Fix any issues then compile the code. Run the executable file from a command line window and view the expected output. In the next section, the executable will be viewed as bytecode. Below is the converter being used in Visual Studio:

Using a binary editor such as the one noted, open the executable file. View the byte code which will look something like this:



View the binary code and find the text “Hey this actually works” in the text interpretation. Using the binary editor, replace the text “Hey this actually works” with “Hey, this doesn’t work”. Do this by tying over the top of the highlighted words in the editor. See below:



Save the changes as a **new** file “hello\_does\_not\_work.exe”.

Note: The byte code was changed as the text was changed.

**Attempt to run** the changed executable and note that it now (probably) won’t run. This is because the new file has a different total length and Windows expects the file to finish at byte 09ff hex or 2,559 decimal. Also note, the value 2E hex is a dot or “.”, the value 0D hex is 13 decimal and the value 0A hex is 10 decimal. Looking at the demo code again…

print chr$("Hey, this doesn't work.",13,10)

This command translates to:

* Print the text “Hey this doesn’t work.”
* Then print chr(**13**) which is a carriage return
* Then print chr(**10**) which is a line feed

So every time the byte code contains the sequences **2E 0D 0A** … It’s a full stop followed by the end of a line! Note that the compiler silently adds a zero at the end, known as a null byte (0 hex).

From the command line, now compare the files. See the command below:

**fc hello.exe hello\_does\_not\_work.exe**

The output will look like this:

c:\masm32\tutorial\console\demo1>fc hello.exe hello\_does\_not\_work.exe

Comparing files hello.exe and HELLO\_DOES\_NOT\_WORK.EXE

0000080A: 61 64

0000080B: 63 6F

0000080C: 74 65

0000080D: 75 73

0000080E: 61 6E

0000080F: 6C 27

00000810: 6C 74

00000811: 79 20

00000812: 20 61

00000813: 77 63

00000814: 6F 74

00000815: 72 75

00000816: 6B 61

00000817: 73 6C

00000818: **2E** 6C

00000819: **0D** 79

0000081A: **0A** 20

0000081B: 00 77

0000081C: 00 6F

0000081D: 00 72

0000081E: 00 6B

0000081F: 00 **2E**

00000820: 00 **0D**

00000821: 00 **0A**

c:\masm32\tutorial\console\demo1>

In order to make this altered code executable, the user needs to remove some of the padding (the zeroes at the end of the file) until the length of the file again is a multiple of 16 bytes. This example is 159 \* 16 byte blocks. If you look at the file system, both files are 2560 bytes in size. Note that the byte counter starts at zero and finishes at 2559 (09ff hex) but the file length is 2560 (0a00 hex).

# View and understand the output of the compiled code

This section uses the demo code from hello.exe in the MASM32 tutorials. This file was created in 2005!

In the previous section, an introduction to the output was provided with a demonstration of where and how to view the byte code produced by the compiler. This output won’t vary much between compilers however this section will highlight some of the outputs specific to a Microsoft x86 targeted compiler.

From the PE File format guide, the byte code of a compiled file can be broken down into the following sections, not all of which appear in every byte code file:

* File Headers
  + MS-DOS Stub
  + Signature
  + COFF File Header
* Machine Type (see the first line of the code samples in Appendix A)
* Characteristics
* Optional Header (think “Magic Number”)
* Optional Header Standard Fields
* Optional Header Window Specific Fields
* Windows Subsystem
* Optional DLL Characteristic Fields
* Optional Header Data Directories
* Section Table
* Section Flags
* Grouped Sections; and
* Other Contents of the File (think “this is where everything else goes”)

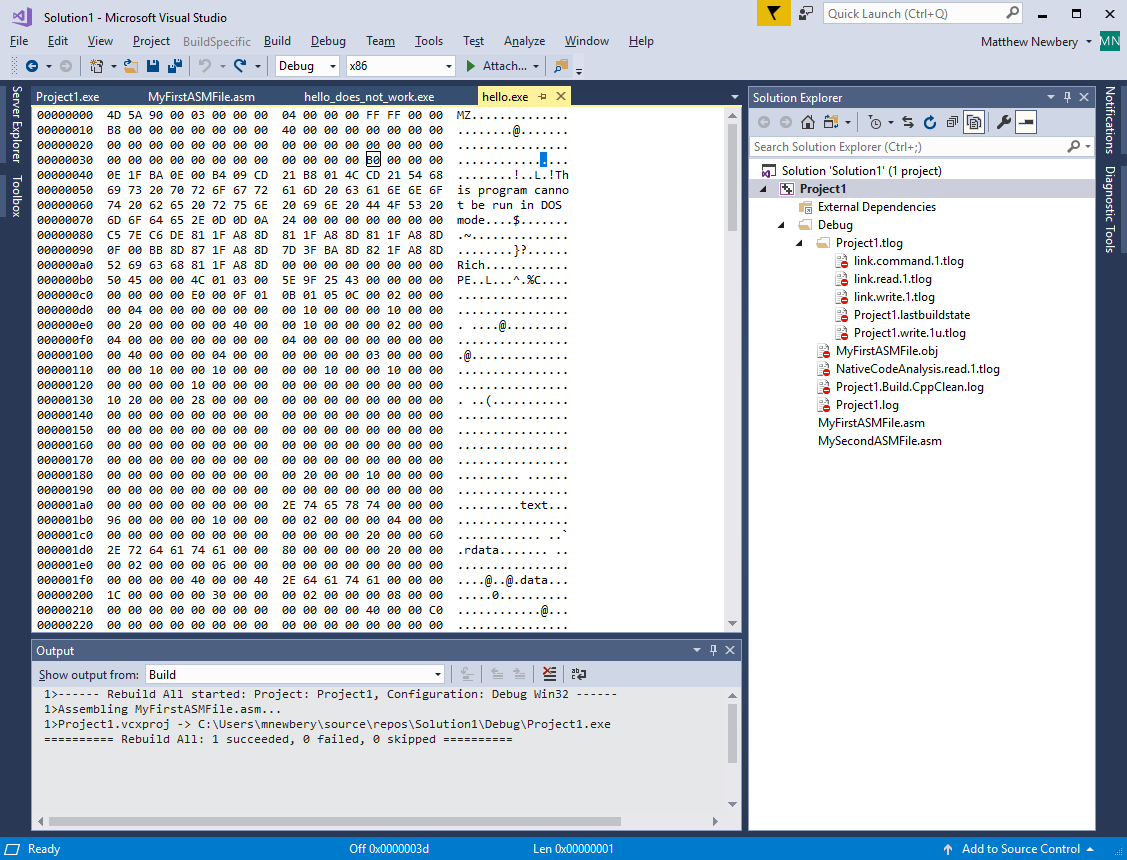
## File Headers

The “PE file headers” consist of a Microsoft **MS-DOS stub**, **the PE signature**, the **COFF file header**, and an optional header. A COFF object file header consists of a COFF file header and an optional header. In both cases, the file headers are followed immediately by section headers.

### MS-DOS Stub

“This program cannot be run in DOS mode’

This isn’t the first thing that the compiler outputs but it is the most obvious when the byte code is first viewed. See below:



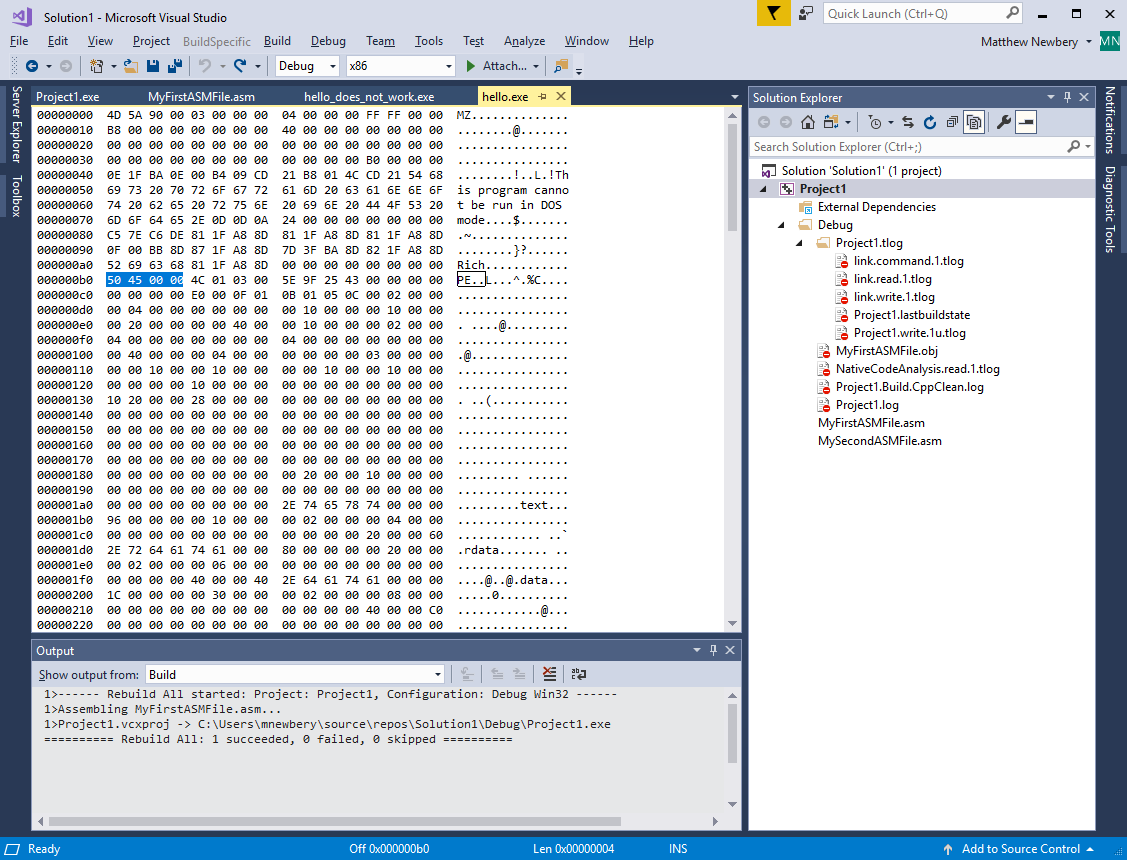
The [Portable Executable format](https://docs.microsoft.com/en-us/windows/win32/debug/pe-format) specification states the following:

The MS-DOS stub is a valid application that runs under MS-DOS. It is placed at the **front** of the EXE image. The linker places a default stub here, which prints out the message “**This program cannot be run in DOS mode.”** when the image is run in MS-DOS. The user can specify a different stub by using the /STUB linker option.

At location 0x3c (highlighted in the image above), the stub has the file offset to the PE signature. This information enables Windows to properly execute the image file, even though it has an MS-DOS stub. This file offset is placed at location 0x3c during linking. In the image above the offset is B0 hex (176 decimal) from the start. That is to say, the signature starts at 000000b0 hex in the image above.

### Portable Executable (PE) Signature

After the MS-DOS stub, at the file offset specified at offset 0x3c, is a 4-byte signature that identifies the file as a “PE format” file. This signature is "PE\0\0" (the letters "P" and "E" followed by two null bytes). This signature says “I am a file in PE format”. See below:



### COFF File Format Header

The Common Object File Format (COFF) Specification can have up to 96 sections that explain the structure of the rest of the file. [View the PE Format document](https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#coff-file-header-object-and-image) section on COFF for a complete explanation. The COFF header always appears after the PE signature. In the sample code in Appendix A and in the previous image, the COFF File Format Header is:

| **Offset** | **Size** | **Field** | **Value** |  |
| --- | --- | --- | --- | --- |
| 0 | 2 | Machine | From file locations b5 and b6 hex we get **4C** and **01** respectively. | This appears as low-byte first ([little-endian](https://en.wikipedia.org/wiki/Endianness#Atomic_element_size_8-bit)) so the actual value is 014c hex. The value corresponds to “Intel 386 or later processors and compatible processors” |
| 2 | 2 | Number Of Sections | Locations b7 and b8 hex are **03** and **00** | Same as above, the value is 0003 hex which is 3 sections |
| 4 | 4 | Time Date Stamp | Locations b9 to bb hex contain values **5e 9f 25 43**. This translates to Monday, 12 September 2005 3:31:31 PM (because it’s an old demo) | Number of seconds since 00:00 January 1, 1970. Swapping the order gives **43 25 9f 53**. See the epoch converter in the tools section! |
| 8 | 4 | Pointer To Symbol Table | Locations bc to bf hex contain all zeroes! | The file offset of the COFF symbol table, or zero if no COFF symbol table is present |
| 12 | 4 | Number Of Symbols | Locations c0 to c3 hex contain all zeroes! | The number of entries in the symbol table |
| 16 | 2 | Size Of Optional Header | Locations c4 and c5 hex contain E0 and 00. This corresponds to 00e0 hex or 224 bytes. In this example the header is blank! | The size of the optional header, which is required for executable files but not for object files |
| 18 | 2 | Characteristics | Locations c6 and c7 hex contain 0F and 01. Swapping these around yields two flags:  0x0100 and 0x000F | The flags that indicate the attributes of the file. For specific flag values, see [Characteristics](https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#characteristics). The first flag means “The computer supports 32-bit words”. The second flag is actually flags 1,2,4, and 8 combined |

This section has been just a taste of what is contained in the first 255 bytes of a simple byte code file.

From this section, it is clear that working through byte code to determine even a little useful information is quite tedious and prone to error. This is why some very clever people created automated tools to do the job quickly and accurately.

If this demo was fun, view Demo 2 and see if the byte code emitted by the compiler can be changed to output a different message.

The next section covers tools and techniques. These tools allow the developer (that’s you) to create, read, change and understand byte code as part of a cybersecurity defence, emulation or offence team.

# Tools and Techniques

## Why am I doing this?

This page gives some insights into why it’s a good idea to periodically dive into a tool stack useful for white hat hacking and offensive cyber security even if it is not part of a typical cybersecurity defender role:

<https://www.cbtnuggets.com/blog/training/exam-prep/how-to-prepare-for-a-capture-the-flag-hacking-competition>

## Free isn’t always the same as Good

This section explains and provides links to various tools that reduce or remove the need for manual interpretation of byte code. A solid understanding of how the byte code files are structured **is still required** for the following reasons:

* The tools don’t always work perfectly and it is important to know how they work
* The byte code can still have actionable information (as in “I can use it to explain something”) stored in the padding but this information may never be referenced as the file is executed and never shown by the tools; and
* Some byte code is crafted to deliberately obfuscate its intention or cause errors to be thrown in order to cause vulnerability that can be exploited by other means, after an error.

More tools can be found in the [Kali Linux](https://www.kali.org/) suite however that topic is beyond the scope of this tutorial.

The most important tools for automation are listed here and their locations appear further down in the next section.

## Mandatory tools for this “Capture the Flag”

### MASM32 DUMPPE

This utility is downloaded as part of the MASM32 tool set and outputs all of the header information in a human readable format. This tool is to be used to **answer some of the questions in the Capture-The-Flag report**. The location of this utility is mentioned later in this section.

### MASM32 DUMPBIN

This utility is downloaded as part of the MASM32 tool set and outputs all of the section, linker and dependency information in a human readable format. This tool is to be used to **answer some of the questions in the Capture-The-Flag report**. Note that when a body of code is compiled, the sections will be limited to (for example) 1000h or 4,096 byte chunks or whatever size is set by the compiler. The location of this utility is mentioned later in this section. Note here there is a difference between the physical location(s) of the data in each section as is sits in the byte code and the “virtual” location of the data once it appears in memory during code execution. **Both** pieces of information are important. This tool will show the virtual locations plus a number of other details, depending on the commands used.

### Code De-compiler such as IDA or Ghidra

Code de-compilers do all of the heavy lifting when it comes to showing the intent of a byte code file and they are indispensable with regard to quickly displaying execution paths graphically. De-compilers can also attempt to convert the byte code back into assembler. Code de-compilers typically **do not do** these things:

* Output some PE File header information, compiler details or the date stamp in the file; and
* Display information hidden in the padding.

Both IDA version 7 and Ghidra are free to anyone for non-commercial use.

Ghidra was developed by the United States National Security Agency (NSA). The binaries were released at RSA Conference in March 2019, the sources were published one month later on GitHub. Ghidra is seen by many security researchers as a competitor to IDA Pro and JEB De-compiler.

This tutorial recommends and is focussed on IDA version 7 free ware. The location of the IDA download appears later in this section.

MASM32 tools and IDA will be used to complete the Capture-The-Flag report.

## Further Reading

This primer document has a background on dis-assembly of various x86 byte code files

<https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files>

This Wikipedia article provides an introduction to little- and big-endianness. To remember the byte-order, referring to little-endianness “you crack the egg **first**, **at the little end**”. See below:

<https://en.wikipedia.org/wiki/Endianness>

This page links to the Portable Executable (PE) format which will become the reference source for all byte code analysis

<https://docs.microsoft.com/en-us/windows/win32/debug/pe-format>

## Online Tools

This web page will allow a developer to paste in some text and covert it to or from a human readable format

<https://www.rapidtables.com/convert/number/ascii-hex-bin-dec-converter.html>

This epoch converter is handy to see what “seconds since 1970” looks like in a human readable format. The epoch converter will accept both big-endian hex and integers once punctuation is removed. Note here that time stamps in byte code are Little-Endian and may need to be converted to Big-Endian before being input to the tool

<https://www.epochconverter.com/>

## Free Tools from the Internet

Get the MASM 32 Bit SDK, version 11 zip file here

<http://www.masm32.com/download.htm>

Once MASM32 is downloaded, look at the bin folder for useful utilities such as

* Dumppe.exe; and
* Dumpbin.exe

Free De-compiler to help with writing Capture-The-Flag reports

<https://www.hex-rays.com/products/ida/support/download_freeware.shtml>

…and its tutorial pages

<https://www.hex-rays.com/products/ida/support/idadoc/>

If working with Unix, get familiar with 010 Editor. This tool also works on Windows

[https://www.sweetscape.com/010editor/](https://www.sweetscape.com/010editor/%20)

# Write a Report on the code found

Note: **The “Flag” that is being captured in this report** is the successful modification of the binary file in a way that makes it do the job it was originally intended to do. The report will document “The captured Flag” in the following way:

* What the code originally did and why this was wrong
* What the intent appeared to be in the opinion of the report writer; and
* What did the report author change (if anything) to make the code more useful?

Follow the steps below to get the information required to complete the Capture-The-Flag report:

Step 1: Learn how to use Ghidra and IDA

View the following YouTube video on solving Capture-The-Flag with Ghidra and IDA:

<https://www.youtube.com/watch?v=S06pgk4DjFQ>

Step 2: Download the sample binary file from the Github repository to your local machine

<https://github.com/mnewbery/MASM/blob/master/output1.bin>

Step 3: Using the MASM32 tool **DumPE.exe** or by any other means, analyse the binary file. Provide answers to the following questions and note the answers in a new, blank document:

1. What address does the Portable Executable Signature start at?
2. What is the value of the Portable Executable Signature?
3. What is the target Machine value?
4. How many sections does the binary say it has?
5. What is the Time Date Stamp in hex and the human readable value?
6. Where is the Symbol Table?
7. How many symbols are in the symbol table?
8. What is the Optional header size?
9. What are the Characteristics flags that were set?
10. What is the Magic number?
11. What is the Linker Version?
12. What is the Target Subsystem?

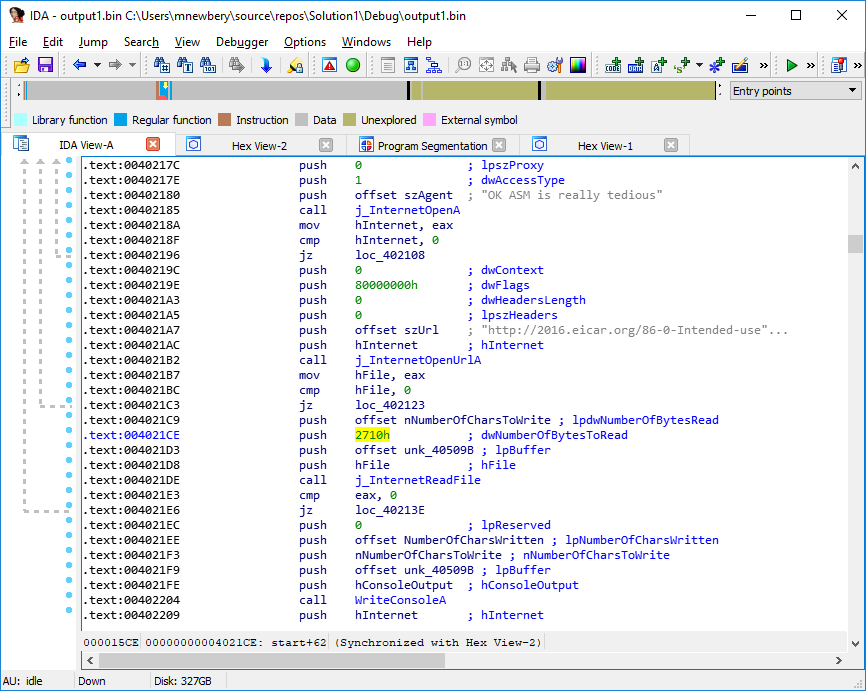
Step 4: Answer these extra questions and note the answers in the same document

1. What else can be determined from the binary file, such as information about the machine that did the compiling? Is there any “odd” human readable text in the binary that might help identify who or what created this file and what their motivation is? Put this information in the report using less than 150 words
2. In 25 words or less, describe what was the motivation of the originator, in the opinion of the report author

Step 5: Using IDA, Ghidra or by any other means, change the code to do something more useful. Document any changes made to the byte code and the new output.

## Hints

View the IDA and Visual Studio screen grabs below and consider what they are showing:



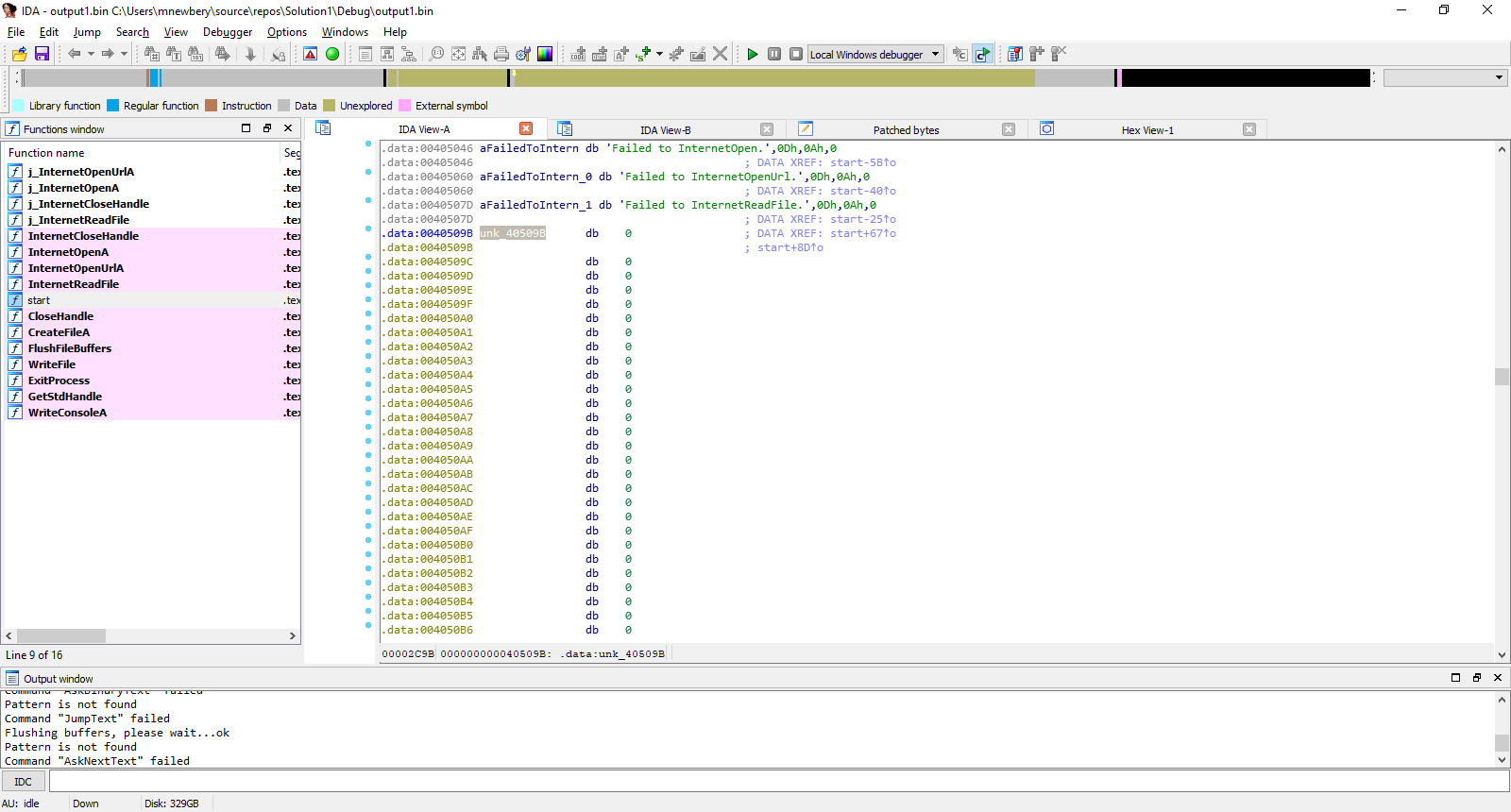
What does this number mean?

The original version of output1.bin contained a segment from address 2C9B hex to 53AB hex that contained nulls. This section was used in memory while the binary was being executed to store something. It turns out the space allocated was too small. The compiler output co-incidentally put in a few more nulls so that the available space was eventually 3001 hex (12,289 decimal bytes) in size.

The most recent compiled version has allocated “at least” 14,440 bytes of space in the binary between the addresses 2C9B hex and 5ECB hex. The image above identifies the handle to the memory chunk as “unk\_40509B”.

Your hint is to determine the size of the old reserved space and the new reserved space.

There is NOTHING in the decompiler that will explain what is going on with allocation of reserved space. Now have a look at this image below:



The reserved segment starts in the binary at 2C9B hex. Move the yellow arrow to see where it ends

Find and highlight this line in IDA View-A, or just move the yellow arrow

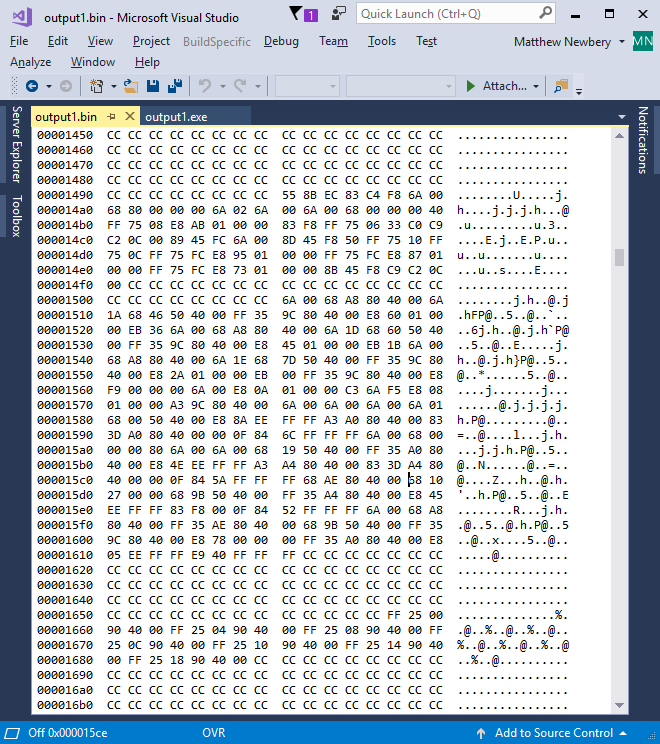
Yellow Arrow here!

Your hint will be to find out how wide the reserved segment is and compare it with the value for dwNumberOfBytesToRead. What would happen if, in the case of the old binary, the value of dwNumberOfBytesToRead was bigger than the reserved space?

Hint: In the binary, increase the value of this variable and see what else appears on the screen.

Happy hunting!

Here is another view of the byte code:



Step 5: Download the report template from <https://github.com/mnewbery/MASM/blob/master/Capture%20the%20Flag%20report%20template.docx> and complete the report using the findings from the previous steps.

# Appendix A: Sample Code

Sample code here will run on any compiler if the demo libraries are installed in c:\masm32. If not using the Visual MASM installer, view the Tools and Techniques section in this document for an alternative source

## Demo 1: Hey this actually works

.486 ; create 32 bit code

.model flat, stdcall ; 32 bit memory model

option casemap :none ; case sensitive

include \masm32\include\windows.inc ; always first

include \masm32\macros\macros.asm ; MASM support macros

include \masm32\include\masm32.inc

include \masm32\include\gdi32.inc

include \masm32\include\user32.inc

include \masm32\include\kernel32.inc

includelib \masm32\lib\masm32.lib

includelib \masm32\lib\gdi32.lib

includelib \masm32\lib\user32.lib

includelib \masm32\lib\kernel32.lib

.code ; Tell MASM where the code starts

start: ; The CODE entry point to the program

print chr$("Hey, this actually works.",13,10)

exit

end start ; Tell MASM where the program ends

## Demo 2: Basic Window Dialog

Here some more demo code. This demo shows some commented out code where an alternate source of the library files may be found. The “include” files are just header files that describe what can be expected in the **lib** files. The code will not compile without the lib files being successfully referenced

.386 ; Enables assembly of non-privileged instructions for the 80386 processor

; … disables assembly of instructions introduced with later processors.

.model flat, stdcall

option casemap:none

include C:\masm32\include\WINDOWS.INC

include C:\masm32\include\kernel32.inc

include C:\masm32\include\user32.inc

;includelib "C:\Program Files (x86)\Windows Kits\8.1\Lib\winv6.3\um\x86\kernel32.lib"

;includelib "C:\Program Files (x86)\Windows Kits\8.1\Lib\winv6.3\um\x86\user32.lib"

includelib "C:\masm32\lib\kernel32.lib"

includelib "C:\masm32\lib\user32.lib"

.data

MsgCaption DB "First Steps", 0

MsgBoxText DB "This is a bare bones application", 0

.code

start: ; The CODE entry point to the program

Invoke MessageBox, NULL, Offset MsgBoxText, Offset MsgCaption, MB\_OK

Invoke ExitProcess, NULL

end start ; Tell MASM where the program ends