WASM Deep Dive

By Geoff Darst

Introduction

The purpose of this document is to take a deep dive into WASM. We will explore the complete end-to-end scenario of compiling C/C++ code to WASM and then loading and executing WASM in Microsoft Flight Simulator. We will perform our investigation from the system level with the goal being to have as complete of an understanding as possible of how WASM works (down to the hardware level) in Microsoft Flight Simulator.

This document assumes understanding of how to build WASM in Microsoft Visual Studio for Microsoft Flight Simulator as well as create and deploy the package. For anyone who has not yet built their first WASM, I would recommend doing so prior to continuing with this document. The Platform Toolset instructions can be found here: SDK Documentation (flightsimulator.com).

Background

What is WASM?

WASM stands for Web ASseMbly. It is a specification for an abstract LLVM (Low Level Virtual Machine). The complete specification can be found here: WebAssembly Specification — WebAssembly 1.1 (Draft 2021-05-17). Unlike architectures such as x86-64, which represent physical chips, LLVMs are an abstraction that allows imagining computation without tying it to a specific hardware implementation.

The WASM LLVM architecture is that of a stack machine. There are no defined registers in the instruction set. Instead, operations generally involve pushing and popping values from the stack. For example, a binary operation such as Add would pop two values from the stack (the left and right operands) and then push the result of the operation back onto the stack. The byte-code instruction set for WASM is defined here: Binary Format — WebAssembly 1.1 (Draft 2021-05-17).

Compilers that support WASM map higher level languages to WASM byte code. The Microsoft Flight Simulator SDK provides the Clang compiler, which will generate WASM from C/C++ code. However, any toolset which correctly implements the WASM specification should be capable of generating WASM modules which could be used by Microsoft Flight Simulator. This would imply that the possibility exists of building add-ons in other languages.

In addition to the byte-code specification, there is also a specification for a human-readable form of the WASM instruction set. This is known as WAT (Web Assembly Text) and can be found here: Text Format — WebAssembly 1.1 (Draft 2021-05-17). WAT is somewhat analogous to an AST (Abstract Syntax Tree). The basic unit of WAT is the S-expression, which enables WAT to encode the structure of the underlying code within its syntax. Like assembly, WAT can be directly compiled into WASM (though the Microsoft Flight Simulator SDK does not provide the tools with which to do that).

The important thing to understand about WASM is that because it only defines the specification for a virtual machine, WASM code by itself is not useful. No existing hardware can directly execute WASM byte code. To consume WASM, additional software must take the WASM byte-code and convert it to the

appropriate byte-code for the specific hardware it is being run on. There is no specification that dictates how this conversion must be done, so WASM consumers are completely free to implement the backend (known in WASM parlance as the “embedder”) however they would like.

Why WASM?

While the idea of targeting a virtual machine and then taking the extra step to convert WASM to the byte-code of a physical machine may seem strange and unnecessary, there are two primary advantages to taking that approach.

The first advantage is portability. By targeting a LLVM, WASM code can execute on any platform that supports it. That includes not only hardware, but any application (such as a web browser) which implements a WASM back-end and runtime. In the case of Microsoft Flight Simulator, an C/C++ code which is compiled to WASM is guaranteed to run on X-Box.

The second advantage is security. The WASM LLVM architecture does not include any virtualization of hardware beyond the CPU and the instruction set is extremely limited in what it can do. This is by design; WASM code necessarily is sandboxed and only has access to system and hardware to the extent that the runtime allows it. As a result, it should be completely safe to run untrusted WASM code. The downside is that in order to enable untrusted code to run safely, such code cannot be allowed to do very much. The WASM runtime implementation in Microsoft Flight Simulator imposes significant limitations on add-on functionality. It should be mentioned that this is a design choice. The WASM specification does not impose any requirements on the implementation of the runtime execution environment. While it certainly makes sense for web browsers to provide a robust security model for WASM, whether such an approach makes sense for Microsoft Flight Simulator is a more open question.

What is WASI?

WASI stands for Web Assembly System Interface. It is an attempt to standardize how system calls might be enabled from within WASM. The idea is to use POSIX (Portable Operating System Interface) as a basis for providing a portable means with which WASM can make system calls. As it turns out, WASI isn’t really relevant to Microsoft Flight Simulator since its WASM runtime implementation is only intended to execute code written specifically as a Microsoft Flight Simulator add-on.

What is EmScripten?

EmScripten is a compiler-toolchain assembly for building WASM. Initially, WASM was developed for use in the web browser space to complement JavaScript. Browser runtimes which support WASM enable both calls from JavaScript to WASM and WASM to JavaScript. In the browser environment, WASM is instantiated from within JavaScript. As such, EmScripten generates much of the JavaScript glue code that simplifies using WASM in the web environment.

In Microsoft Flight Simulator, WASM is loaded directly by the runtime itself, rather than through JavaScript. As such, EmScripten is not the right toolset for building WASM that is intended to run in Microsoft Flight Simulator. Understand that EmScriptem documentation with respect to WASM may not be accurate in the context of development for Microsoft Flight Simulator.

WASM End-to-End

The rest of this document will concern itself with the complete end-to-end scenario of building and running a WASM module. For purposes of our investigation, we will use a simple stand-alone WASM as it is the easiest package to create and get loaded in Microsoft Flight Simulator.

A Simple WASM

For purposes of our investigation, we will create a simple stand-alone WASM module. In Visual Studio, that means creating a project using the "MSFS WASM Module" project template (which can be used for both stand-alone and gauge modules.

The project will contain only a single .cpp file, which will look like this (line numbers added for reference):

```
// WASM_Module1.cpp

1. #include <stdio.h>
2. #include <MSFS/MSFS.h>
3. #include "WASM_Module1.h"


4. extern "C" MSFS_CALLBACK void module_init(void)
5. {
       a. printf("Initialization called!");
6. }
```

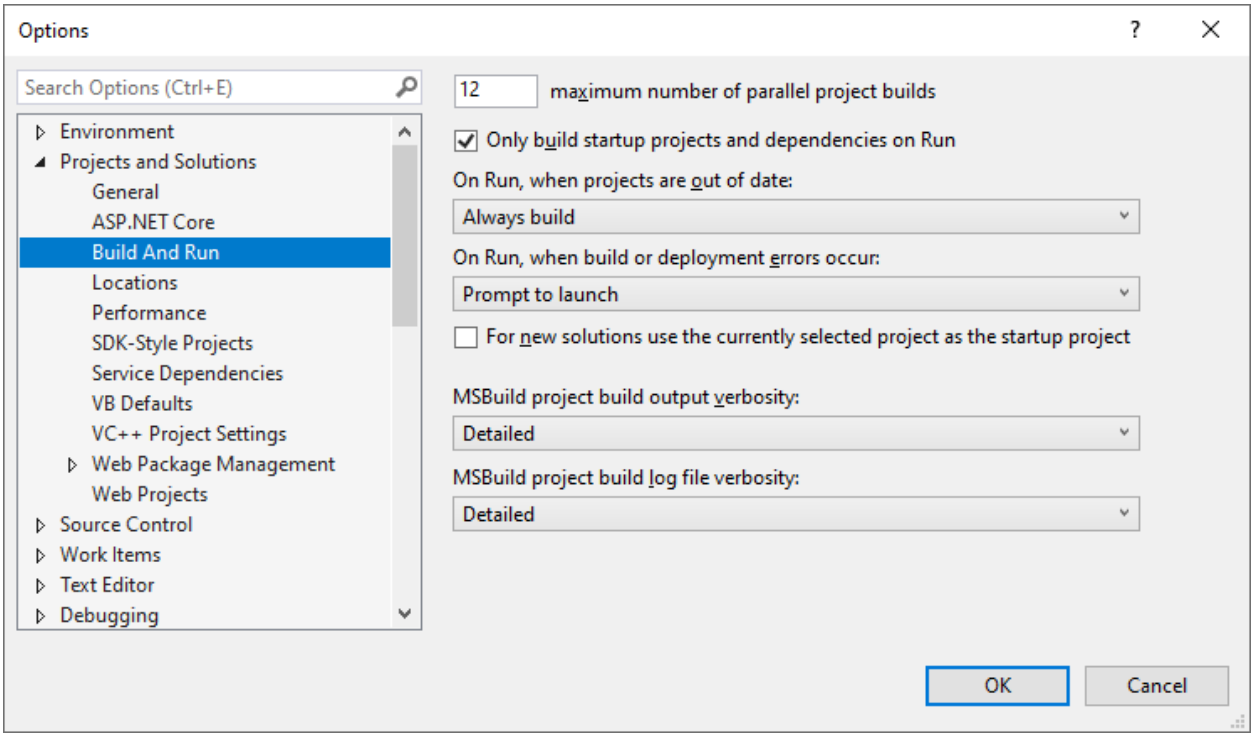
Hopefully, this code is easily understood, but here is what each line means:

1. We include stdio.h from the c runtime since we will be calling printf in our module_init function below. We will discuss where we are importing the c runtime as well as how it is implemented shortly.
2. We include MSFS.h in order to resolve the MSFS_CALLBACK calling convention. MSFS.h can be found in the ..\ MSFS SDK\WASM\include\MSFS folder of the Microsoft Flight Simulator SDK. If we examine the declaration for MSFS_CALLBACK, we see a rather mysterious line of code for the case where __INTELLISENSE__ is not defined (which would be the case where we were compiling outside of Visual Studio). The code in question is: __attribute__((export_name(""))). Jumping ahead of ourselves somewhat, we search the Clang documentation and discover that this has to do with how symbols are exported. Presumably, we don't need to care too much about this beyond understanding that callbacks need to be declared as MSFS_CALLBACK in order for linkage to work properly.
3. This is just the include for the automatically generated header file that is paired with our source file. Examining the contents of that header, we see a couple of macros that are turned on when compiling within Visual Studio, again affecting symbol generation. Otherwise, several macros are defined to be no-ops.
4. This is the function declaration for module_init. As we will see this will be called by Microsoft Flight Simulator when the WASM is loaded. We note that it is declared as: extern "C" which exports the function using C style linkage. In other words, this function can be called externally and the caller is expected to use the C calling convention.

5. (a) This is just a call into the cruntime printf function to (hopefully) allow us to explore both how the C runtime is implemented as well as how system calls from the Microsoft Flight Simulator WASM runtime work.

Building WASM

Visual Studio makes building WASM simple, but in the process it also hides all of the details from us. To start with, we are going to go to the "Tools" menu and choose "Options." We will expand the "Projects and Solutions" node and select "Build and Run, " at which point we should see the following:



We will set the "MSBuild project build output verbosity" and "MSBuild project build log file verbosity" to "Detailed." Doing this will give us complete visibility into the build process.

While this isn't a tutorial on the Visual Studio Build process, but the necessary build templates for WASM are located in the ..\MSFS SDK\WASM\vs\2019 folder. If we open up Microsoft.Cpp.MSFS.Common.props, we can examine the properties for the various build tools and quickly ascertain that WASM gets compiled with clang-cl.exe, linked with wasm-ld.exe, and static library files get generated with llvm-ar.exe. Not surprisingly, those tools can be found in ..MSFS SDK\WASM\llvm. If we open a command window and visit that folder, we can run each of those executables with the "help" switch in order to view the available command options. Additionally, we can find Clang documentation here: Clang C Language Family Frontend for LLVM, wasm-lld documentation here: WebAssembly lld port — lld 13 documentation (llvm.org).

Now we can build, and in doing so, we get a complete dump of the entire build process—both within the output window of Visual Studio and within the build log file in our build directory. Searching the output, we can locate the calls to clang-cl.exe and wasm-ld.exe.

Here is the clang-cl call:

C:\MSFS SDK\WASM\llvm\bin**clang-cl.exe** /c /I"C:\MSFS SDK\WASM\wasi-sysroot\include" /I"C:\MSFS SDK\WASM\wasi-sysroot\include\c++\v1" /I"C:\MSFS SDK\WASM\include" /I"C:\MSFS SDK\SimConnect SDK\include" /nologo /WX- /diagnostics:column /D __wasi__ /D _STRING_H_CPLUSPLUS_98_CONFORMANCE_ /D _WCHAR_H_CPLUSPLUS_98_CONFORMANCE_ /D _LIBCPP_HAS_NO_THREADS /D _WINDLL /D _MBCS /MDd /GS- /fp:precise /std:c++14 /Fo"MSFS\Debug\\" /Gd /TP /clang:-g --target=wasm32-unknown-wasi /showFilenames /clang:--sysroot="C:\MSFS SDK\WASM\wasi-sysroot" /clang:-fvisibility=hidden /clang:-ffunction-sections /clang:-fdata-sections /clang:-fno-stack-protector /clang:-fno-exceptions /clang:-fms-extensions -Werror=return-type -Wno-unused-command-line-argument -m32 WASM_Module1.cpp

Assuming some familiarity with compilers, there is nothing here that is too surprising. We do note that the "–target" switch specifies "wasm32-unknown-wasi" which seems reasonable.

The wasm-ld command line is probably more interesting as this is where the linking to WASM happens. Here is that call.

C:\MSFS SDK\WASM\llvm\bin**wasm-ld.exe** --no-entry -o "D:\MyFSProjects\Sources\WASM_Module1\MSFS\Debug\WASM_Module1.wasm" --allow-undefined --export malloc --export free --export __wasm_call_ctors --export-table --gc-sections -O3 --lto-O3 -L "C:\MSFS SDK\WASM\wasi-sysroot\lib\wasm32-wasi" -lc++ -lc++abi -lc "C:\MSFS SDK\WASM\wasi-sysroot\lib\wasm32-wasi\libclang_rt.builtins-wasm32.a" "MSFS\Debug\WASM_Module1.obj"

There are a couple of exports to note—the fact that malloc and free are exported could be interesting, but we still do not know how we get an executable from this, so we don't yet know how such exports might be used. What is probably more interesting are these switches: "-lc++ -lc++abi -lc". The -l switch is used for importing libraries, so it looks like libc++.a, libc++abi.a, and libc++abi.a are all being imported (see the man page for the GNU linker for a description of how the -l flag works). Clearly, this is where our C runtime library (CRT) is coming from. We will want to investigate that.

Object Files in WASM

Before we try to understand the CRT implementation, we will want to have a look at the object code that Clang generated for WASM. Unfortunately, the Microsoft Flight Simulator SDK does not provide any tools with which to do that. Visual Studio provides dumpbin.exe, but when we attempt to use that, the format of the obj file is not recognized.

We could start by loading WASM_Module1.obj (which is the compiler generated object file) into a hex editor just to see what we have. I prefer HxD, which is freeware and is very feature rich. It can be found here: HxD - Freeware Hex Editor and Disk Editor | mh-nexus. If we do that, we will see a bunch of recognizable text, but it does not appear to be COFF (Common Object File Format). From the COFF documentation (found here: PE Format - Win32 apps | Microsoft Docs) we would expect that the first two bytes of a COFF file would be the Machine field of the COFF File Header. The first two bytes are 0x00 0x61, which reconstituting in little-endian gives us 0x6100, which is not a valid machine type (we would expect 0x014c, which is IMAGE_FILE_MACHINE_I386). No wonder dumpbin could not read this.

We could go looking for a specification for whatever format this file is in, (and in fact, it is documented here: tool-conventions/Linking.md at master · WebAssembly/tool-conventions (github.com)) so that we could grovel our way through the raw bits of the obj file, but fortunately, that

is not necessary. There are already tools for this. Namely, we can use WABT (the Web Assembly Binary Toolkit), which is available here: WebAssembly/wabt: The WebAssembly Binary Toolkit (github.com). You can either clone the repository or just down load and extract the tarball containing the latest release. Once you have done that, you will find a number of great tools in the ..\bin folder of the WABT root. Documentation links for all of the tools can be found on the project page in GitHub.

To dump the WASM obj file, we can use wasm-objdump.exe. With that tool, we can disassemble WASM byte-code to WAT and dump the headers.

Dumping the headers can be accomplished with the "-h", "-x", and/or "-s" with each switch providing a different level of information.

Dumping with "-h" gives us a list of the sections, including section names, starting and ending addresses, sizes, and an element count.

Dumping with "-x" (details), we can see some interesting things in the sections. Looking at the import section we see:

Import[4]:

- memory[0] pages: initial=1 <- env.__linear_memory

- table[0] type=funcref initial=0 <- env.__indirect_function_table

- global[0] i32 mutable=1 <- env.__stack_pointer

- func[0] sig=1 <env.printf> <- e

This is interesting because it suggests that there is some sort of an environment block which exists within the runtime through which access to system functionality is granted. Presumably, the environment defines the boundaries of the WASM sandbox.

There are also things like a symbol table, a data table, relocation tables, etc. All are standard things which we would expect would need to be provided to a linker in order to generate something executable.

Dumping the with "-s" gives us the raw bits for each section (along with ASCII).

Disassembling OBJ

OK, now that we have checked out the headers, we will examine the disassembly using the "-d" switch.:

```
WASM_Module1.obj:	file format wasm 0x1


Code Disassembly:


00009d func[1] <module_init>:
 00009e: 03 7f                      | local[0..2] type=i32
```

```
0000a0: 23 80 80 80 80 00       | global.get 0 <env.__stack_pointer>
0000a6: 21 00                   | local.set 0
0000a8: 41 80 80 80 80 00       | i32.const 0
0000ae: 21 01                   | local.set 1
0000b0: 41 00                   | i32.const 0
0000b2: 21 02                   | local.set 2
0000b4: 20 01                   | local.get 1
0000b6: 20 02                   | local.get 2
0000b8: 10 80 80 80 80 00       | call 0 <env.printf>
0000be: 1a                      | drop
0000bf: 20 00                   | local.get 0
0000c1: 24 80 80 80 80 00       | global.set 0 <env.__stack_pointer>
0000c7: 0f                      | return
0000c8: 0b                      | end
```

Looking at the output, it seems obvious what we are getting. Each line starts with the location of the instruction (given as a byte offset from the start of the file—which we can confirm by examining the raw bytes in the hex editor), followed by the byte-code for the instruction, followed by the assembly. Note however, that the text is assembly, not WAT. The assembly instructions are documented in the WASM byte code specification as one to one mappings of byte-code to text, but whether tools exist to enable assembling WASM assembly into executable code is unclear. I could imagine that such a thing might be useful for toolset developers, but I cannot think of a reason why anyone else would need (or want) to code in WASM assembly. Although WASMasm does have a nice ring to it, don't you think?

In any case, looking at the byte-code specification gives us enough information to decipher the assembly. Looking at our hex dump, offset **0x00009d** contains 0x01, which matches the index for module_init (which is 1) in the function table. For whatever reason, it appears that the function table indexing is base-one.

0x00009e : Sets up space for local variables. There are three of them and they are all integers. If it isn't evident from the assembly, local (and global) variables are stored in a vector and accessed by index.

0x0000a0: Sets up the frame pointer. The current stack pointer is read from the index zero of the global variable vector and pushed onto the stack. This marks the start of the base of the current stack frame.

0x0000a6: The stack pointer is popped off the stack and stored in element 0 of the local variables vector, preserving the frame base.

0x0000a8: An integer constant, 0 is pushed onto the stack.

0x0000ae: The zero is popped from the stack and stored in element 1 of the local variables vector.

0x0000b0-0x0000b2: Another zero is subsequently pushed onto the stack and then popped and stored in element 2 of the local variables vector.

0x0000b4-0x0000b6: The contents of index 1 and index 2 of the local variables vector are pushed onto the stack.

0x0000b8: The printf function is called. Notice that only the parameters have been pushed onto the stack. From the disassembly, there is no hint of how execution would return from the call instruction. This is because the semantics of function invocation and return are defined by auxiliary rules which are embedded within the call and return instructions themselves. Remember that this is not executable code. It has to be converted to platform-specific native code in order to execute.

One thing that is not immediately clear to me is why there are two zero parameters being pushed onto the stack. Clearly one of them represents the constant string being passed; 0 is the segment index for the string in the data section. However, it is unclear what the second parameter is since our format string does not include any format specifiers.

0x0000be: Pop the return value of printf from the stack and discard it.

0x0000bf-0x0000c1: Clean up the stack by copying the frame pointer address stored in locals[0] back to the global stack pointer (globals[0]).

0x0000c7: Return control back to the caller.

Static Libraries

Recall that when we build our WASM, the command line included links to various static CRT libraries. Since we are importing the CRT function printf, it will be useful to take a moment to understand how that linkage is going to work. Reviewing the command line, we notice a library path (-L) of "C:\MSFS SDK\WASM\wasi-sysroot\lib\wasm32-wasi". This looks promising, and indeed, we find a file named "libcrt.a" there. Static libraries are a compressed directory containing a bunch of obj files. The linker will extract all files in the archive and insert any obj files that contain referenced symbols into the build output. The ".a" extension stands for archive. On the Windows platform, static libraries typically have the ".lib" extension (which is confusing since import library files also use that extension). In any case, ".lib" files are also archive files which can be extracted. So, we will start by manually unzipping libc.a. I prefer to do this with the freeware tool 7-Zip since it seamlessly handles a great number of archival formats. Having extracted libc.a to a folder called libc, we can see that among the contents is a file named "printf.o." Likely, that is the file containing the printf implementation, so let us have a look.

Since we know it is being linked into a WASM module, we can be pretty certain that "printf.o" itself contains WASM byte-code. Therefore we will attempt to use wasm-objdump.exe -d to disassemble the module, which works as expected. The output is as follows:

printf.o: file format wasm 0x1

Code Disassembly:

```
000092 func[1] <__small_printf>:
 000093: 01 7f                  | local[0] type=i32
 000095: 23 80 80 80 80 00      | global.get 0 <env.__stack_pointer>
 00009b: 41 10                  | i32.const 16
 00009d: 6b                     | i32.sub
 00009e: 22 02                  | local.tee 2
 0000a0: 24 80 80 80 80 00      | global.set 0 <env.__stack_pointer>
 0000a6: 20 02                  | local.get 2
 0000a8: 20 01                  | local.get 1
 0000aa: 36 02 0c               | i32.store 2 12
 0000ad: 41 80 80 80 80 00      | i32.const 0
 0000b3: 20 00                  | local.get 0
 0000b5: 20 01                  | local.get 1
 0000b7: 10 80 80 80 80 00      | call 0 <env.vfprintf>
 0000bd: 21 01                  | local.set 1
 0000bf: 20 02                  | local.get 2
 0000c1: 41 10                  | i32.const 16
 0000c3: 6a                     | i32.add
 0000c4: 24 80 80 80 80 00      | global.set 0 <env.__stack_pointer>
 0000ca: 20 01                  | local.get 1
 0000cc: 0b                     | end
```

We will not spend any time reading the disassembly, except to note that there is no function named “printf.” However, a quick look at the headers with “wasm-objdump -x” shows us this:

…

```
Function[1]:
 - func[1] sig=0 <__small_printf>
Code[1]:
 - func[1] size=59 <__small_printf>
```

Custom:

- name: "linking"

- symbol table [count=6]

- 0: F <printf> func=1 binding=global vis=hidden

…

From the custom “linking” section, we can see that there is a symbol for “printf” which is mapped to a function index of one. Looking at the Function section, we see that func[1] is indeed “__small_printf”. Therefore we know that “__small_printf” is indeed the function which external calls to “printf” are resolved to.

Other than that, we note that “__small_printf” is just a wrapper around “vfprintf”, so we there is not much that we can learn from this implementation. Rather than continuing to grovel through WASM disassembly to try to understand what is going on, we will instead direct our efforts towards examining the linked WASM module as well as understanding how Microsoft Flight Simulator generates native code from WASM.

Enter the WASM

Again recalling our wasm-ld.exe command line, we find the “-o” switch to get the name of our output file: “-o "D:\MyFSProjects\Sources\WASM_Module1\MSFS\Debug\WASM_Module1.wasm.” This is the complete WASM module. While we can (and will) use wasm-objdump.exe to view it, we will start by generating WAT by running wasm2wat against it. We will pipe the output to a text file for ease of use like so:

wasm2wat “D:\MyFSProjects\Sources\WASM_Module1\MSFS\Debug\WASM_Module1.wasm” > WASM_Module1_WAT.txt.

To start with, we will have a look at the linked version of our “module_init” function. In WAT, it looks like this:

```
(func $module_init (type 5)
  (local i32 i32 i32)
  global.get 0
  local.set 0
  i32.const 3365
  local.set 1
  i32.const 0
  local.set 2
  local.get 1
  local.get 2
```

```
  call $printf
  drop
  local.get 0
  global.set 0
  return)
```

Notice the bolded (5th) line. In our obj file, this line created an integer constant with a value of 0. Now it is creating an integer constant with a value of 3365. What is going on? To see, we will use "wasm-objdump -x" to view the headers.

What we notice is that the Data section is now giant. We recognize that 3365 = 0xD25 and we find that offset in the Data section:

- segment[0] memory=0 size=2364 - init i32=1024

…

```
 - 0000d10: 3078 0069 6e66 0049 4e46 006e 616e 004e    0x.inf.INF.nan.N
 - 0000d20: 414e 002e 0049 6e69 7469 616c 697a         6174  AN...Initializat
 - 0000d30: 696f 6e20 6361 6c6c 6564 2100              ion called!.
```

Sure enough, the linker has now fixed up the offset to the full data section so that the WASM code passes the correct string constant ("Initialization called!") to "printf."

Generating Names

One of feature of wasm2wat is the "—generate-names" switch. This switch gives auto-generated names to things like vectors which would otherwise remain anonymous. In some cases, it makes the code easier to read. We will make use of that switch as follows:

wasm2wat –generate-names "D:\MyFSProjects\Sources\WASM_Module1\MSFS\Debug\WASM_Module1.wasm" > WASM_Module1_WAT.txt.

Now looking at how our module_init function is generated, we see this:

```
 (func $module_init (type $t5)
  (local $l0 i32) (local $l1 i32) (local $l2 i32)
  global.get $g0
  local.set $l0
  i32.const 3365
  local.set $l1
  i32.const 0
```

```
local.set $l2
local.get $l1
local.get $l2
call $printf
drop
local.get $l0
global.set $g0
return)
```

Notice, for example, how instead of having a 3-vector of integers to represent locals, we now have three local integer variables named $l0, $l1 and $l2. For anyone struggling with the abstraction, generating WAT in this format may make things more comprehensible.

Imports

Now that we have the complete WASM module, complete with the static libraries linked in, we will want to see if it offers us a hint of how the runtime implementation is going to work.

Again, by running "wasm-objdump -x" we can have a look at the headers and we see the following in the imports section:

```
Import[4]:
 - func[0] sig=1 <__wasi_fd_close> <- wasi_snapshot_preview1.fd_close
 - func[1] sig=2 <__wasi_fd_write> <- wasi_snapshot_preview1.fd_write
 - func[2] sig=3 <__wasi_fd_fdstat_get> <- wasi_snapshot_preview1.fd_fdstat_get
 - func[3] sig=4 <__wasi_fd_seek> <- wasi_snapshot_preview1.fd_seek
```

We have four imported functions which are presumably provided by the runtime. This gives us something to look for once we move into Microsoft Flight Simulator. That is, we are going to want to figure out how these imports are getting hooked up. Doing so will likely give us an understanding of how system calls work from within the sandbox.

Microsoft Flight Simulator WASM Runtime

Ok, so at this point we have a low-level understanding of how WASM modules are compiled and linked. We know that there is a WASM implementation of the CRT which is statically linked as needed. Now it is time to discover the mechanism by which our WASM module gets converted into native code and executed in Microsoft Flight Simulator. In particular, we want to understand how system calls work.

Debugging Microsoft Flight Simulator

Microsoft Flight Simulator is extremely resistant to debugging. The executable (FlightSimulator.exe) for the Windows Store version is located in "..\Program

Files\WindowsApps\Microsoft.FlightSimulator_1.16.2.0_x64__8wekyb3d8bbwe." The first thing you will note is that the "WindowsApps" directory is ACLd to prevent users from exploring it. It is possible to change the ACLs to get access to the Flight Simulator directory to gain access, but doing so likely isn't useful. Ideally, it would be great to be able to CreateProcess with the debugger already attached, but I have been unable to get permissions to do that. Presumably, that is a solvable problem, but from looking at the executable, I can see that it is using various technologies to prevent reverse-engineering. All we want to know is the WASM details, so hopefully we develop an understanding through other means.

As a side note, it is possible to copy files out of the Flight Simulator directory from the file explorer dialogs accessed from the various tools in the developer menu while in developer mode. I cannot claim credit for that idea—some very clever person posted it in the Flight Simulator forum, but I can't remember who. In any case, it is a much better option than messing around with ACLs. That said, for anyone who does want to try changing ACLs, I would highly recommend creating a system restore point first. It is very easy to damage your system by doing this, so having a restore point is cheap insurance to be able to put the system back to rights.

Debugging WASM Modules

Ok, so we know we likely are not going to learn what we want to know by disassembling the executable, so we will just start with normal debugging. While Microsoft Flight Simulator is resistant to debugging, it does allow attaching a debugger after a certain point in the startup process. The timing of when it is allowable to attach a debugger is not exactly defined, but I have had no problems attaching as soon as the initial background screen appears. Attaching prior to that will yield repeating STATUS_SINGLE_STEP exceptions. I have attempted to ignore them in Windbg, but the product seemingly wouldn't load after I did that. Again, anti-debugger technologies at work (and no, setting the BeingDebugged flag in the PEB to FALSE does nothing at this point—which given the level of obfuscation in the executable isn't surprising).

In any case, to start debugging our WASM module, we just need to create a package for it and drop it in the Community folder. For stand alone modules, we need to have a "modules" folder under the package root which contains the WASM module. Since this is not a tutorial on how to create WASM modules, I will assume that anyone following along knows how to do that. Once the package has been copied to the Community folder, we launch Visual Studio (or whatever the preferred debugger is), set a breakpoint on our "printf" call in "module_init," launch Microsoft Flight Simulator, and attach the debugger once we see the background screen.

With Visual Studio, we can click the "Debug" menu and choose "Attach to Process" to attach the debugger. If FlightSimulator.exe is not yet running, we can just click the "Refresh" button to refresh the process list in the dialog until we see FlightSimulator.exe. Another nice feature is that there is a "Reattach to Process" option. That option can be used for subsequent attaches; it remembers the last process attached to and choses that (along with the corresponding debug environment).

Once the debugger has attached—assuming we were quick enough—Microsoft Flight Simulator will continue loading until we hit our break point. If we were too quick, the debugger will break when some exceptions are trapped (presumably part of the anti-debugging) logic. If that happens, we need to

restart and attach faster. Alternatively, if we reach the main menu, we were too slow and our WASM module had already loaded before we attached, so our breakpoint was never hit.

Going Native

We got our debugger attached and we hit our breakpoint, so the first question is where did the native code come from? The easiest thing to do is right click on the line of code where we are broken and choose “Go To Disassembly.” That will bring up the Disassembly tab and the first thing we want to do is look at the instruction address because we want to know where this code came from; i.e. was it loaded or was it JIT compiled. My Disassembly pane shows this:

```
extern "C" MSFS_CALLBACK void module_init(void)
00007FFD1EE7D740  sub         rsp,38h
00007FFD1EE7D744  mov         rax,0D2500000000h
00007FFD1EE7D74E  mov         qword ptr [rsp+24h],rax
00007FFD1EE7D753  mov         rax,qword ptr [m87304f354c82b615::linearmemory_0
(07FFD1EE92118h)]
00007FFD1EE7D75A  mov         qword ptr [rsp+30h],rax
{
00007FFD1EE7D75F  mov         eax,dword ptr [m87304f354c82b615::globalvariable_0
(07FFD1EE92000h)]
00007FFD1EE7D765  mov         dword ptr [rsp+24h],eax
       printf("Initialization called!");
00007FFD1EE7D769  mov         dword ptr [rsp+2Ch],0
00007FFD1EE7D771  mov         ecx,0D25h
00007FFD1EE7D776  xor         edx,edx
00007FFD1EE7D778  call        printf_34:m87304f354c82b615:external (07FFD1EE7D6B0h)
00007FFD1EE7D77D  mov         rax,qword ptr [m87304f354c82b615::linearmemory_0
(07FFD1EE92118h)]
00007FFD1EE7D784  mov         qword ptr [rsp+30h],rax
}
```

We can see that our instruction address is **00007FFD1EE7D765**. Clicking the “Debug” menu followed by “Windows”, “Modules” brings up the Modules pane. We can click on the “Address” field to sort by address. In doing that, we quickly find that this code resides in the address space of a dll with a funky (presumably) generated name. We can also see the path from where it was loaded. In my case, the dll lives in “C:\Users\geoff\AppData\Local\Packages\Microsoft.FlightSimulator_8wekyb3d8bbwe\LocalState\packages\wasm_sockets”

Dumping the WASM dll

Now that we know that Flight Simulator turned our WASM module into a dll, we have a good avenue with which to continue our investigation. We are dealing with a PE (Portable Executable) file, so we can examine it with “dumpbin.exe” For anyone unfamiliar with “dumpbin.exe”, it is a command line tool that ships with Visual Studio. Simply open a Developer Command Prompt (found in the Visual Studio 2019 folder under the start menu) and typing “dumpbin …” from there will execute the tool.

We will start by dumping the imports, which is as simple as calling “dumpbin <file name> /imports.” There are a few functions being imported, but nothing that looks like an obvious pipeline to the system. Interestingly, none of the “fd” functions that were being imported by our WASM module are being imported by the PE.

Next, we will use the "/disasm" switch to disassemble the dll. We will pipe the output to a file for easier navigation; i.e. "dumpbin <file name> /disasm > Wasm_disasm.txt" Looking at the resulting disassembly, we immediately see something interesting.

At the very top of the file, we see the following label: "<dll name>_WASM_innative_internal_init." Note that "<dll name>" will appear as the actual name of the generated dll (minus the file extension). Since I do not know how the dll name is generated, I'm not going to include it on the outside chance that it would provide information about my system that I care about remaining private. Back to the disassembly. What is interesting is the part of the "WASM_**innative**_internal_init" label. Innative suggests a proper name to me—like it might be the name of some specific technology. Indeed, searching the rest of disassembly yields a bunch of functions where "innative" is part of the name. A quick Bing search (we are using a Microsoft product after all) using the search term "innative wasm" gives us a promising link at the very top of the list. It turns out that inNative is an AOT (Ahead Of Time) compiler for WASM that generates (among other things) binaries that can be dynamically loaded. It is documented here: innative-sdk/innative: A native non-web embedding of WebAssembly for Windows/Linux (github.com). From looking at the documentation, it appears that inNative can be linked as a static library and that library exposes an API which can be used to generate a dll from a WASM file.

Calling Outside the Sandbox

We now know where the dll containing our WASM code is coming from, but we still do not yet understand how system calls work. Recall that when examining the WAT of the complete WASM module we noticed several import functions. Now that we have our disassembly, we will search it to see if we can find a symbol for one of the imports. The first import is:

- func[0] sig=1 <__wasi_fd_close> <- wasi_snapshot_preview1.fd_close

We will search for "fd_close." Since that is a substring of both strings, it gives us our best chance of getting a hit. In fact, we do get a hit:

```
0000000180001214: E8 B7 59 01 00     call       wasi_snapshot_preview1_WASM_fd_close
```

Clearly there is a function called "wasi_snapshot_preview1_WASM_fd_close," so we will try searching for that. Eventually we find the label and the code looks like this:

```
wasi_snapshot_preview1_WASM_fd_close:
  0000000180016BD0: 48 8B 05 99 B5 00  mov        rax,qword ptr [real_wasi_fd_close]
                    00
  0000000180016BD7: 48 8B 15 BA B6 00  mov        rdx,qword ptr
[?ModuleHdl@@3VWasmModuleHdl_Z@@A]
                    00
  0000000180016BDE: 48 FF E0           jmp        rax
```

Interesting. This code is clearly setting up an indirect call to a function that we have not seen yet: "real_wasi_fd_close." The prefix "real" makes me think we may finally be closing in on understanding

how system calls work. Searching specifically for the “real_wsi_fd_close” label, we find it in a function called InitWASI. The disassembly shows us this:

InitWASI:

```
 0000000180018090: 48 8B 01          mov        rax,qword ptr [rcx]

 0000000180018093: 48 89 05 FE A1 00  mov        qword ptr
[?ModuleHdl@@3VWasmModuleHdl_Z@@A],rax

                00

…

 00000001800180F2: 48 8B 41 48       mov        rax,qword ptr [rcx+48h]

 00000001800180F6: 48 89 05 73 A0 00  mov        qword ptr [real_wasi_fd_close],rax
```

Now it all makes sense. InitWASI is an instance of a WASM module. On entry, rcx will contain the instance pointer the module class or structure. This structure is used to fill out an import table. The addresses of “real_wasi_fd_close” along with all other external functions which may be called from within the sandbox are read and stored in the corresponding symbol locations.

The only thing left is to see this in action. The problem we are going to have is that ASLR (Address Space Layout Randomization) means that the next time we load Microsoft Flight Simulator, our WASM dll isn’t necessarily going to be loaded at the same base address. Therefore, we cannot just locate the current location of InitWASI (which is easy enough to do) and set a breakpoint at the raw address.

Instead, we launch Microsoft Flight Simulator and reattach the debugger. Once we break at “`module_init,`” we can manually add a breakpoint by clicking on the “Debug” menu and choosing “New Breakpoint” and “Function Breakpoint. In the “New Function Breakpoint” dialog box, we enter “{,,InitWASI}<dll name >}” and click OK. Be sure to replace “<dll name>” with the name of the dll (including extension) that was generated from the WASM module. Since this has already been called by the time we hit our breakpoint, restart Microsoft Flight Simulator and reattach the debugger. Be sure that the checkbox next to the breakpoint in the “Breakpoints” tab is checked (“Debug” “Windows” “Breakpoints”). In our case it wasn’t, so we quickly add the check and are rewarded when the debugger breaks at the “InitWASI” disassembly.

We can tell from the disassembly that there is a function table at [rcx+8] and looking at the memory, we see a bunch of address that all appear to be within the FlightSimulator.exe address space. So it is possible that none of the system calls are exposed directly; i.e. Microsoft Flight Simulator isn’t handing out pointers to external dlls, but instead is wrapping all of the functions that it exposes internally before calling them. This certainly makes sense for file system calls since the physical system is running on top of a virtual file system implementation. Looking at the GDI+ calls that are documented in the Low Level API, it would seem that these are also wrapped by FlightSimulator.exe since they all carry an “FS” prefix.

Anyway, looking at the disassembly for the function pointer to `real_wasi_fd_close`, we see:

```
00007FF7ED60F8B0  mov         qword ptr [rsp+10h],rdx
00007FF7ED60F8B5  push        rbx
00007FF7ED60F8B6  sub         rsp,40h
```

```
00007FF7ED60F8BA  mov         ebx,ecx
00007FF7ED60F8BC  lea         rdx,[rsp+58h]
00007FF7ED60F8C1  mov         rcx,qword ptr [7FF7F13AC868h]
00007FF7ED60F8C8  call        00007FF7ED2ED720
00007FF7ED60F8CD  test        rax,rax
00007FF7ED60F8D0  jne         00007FF7ED60F8DD
00007FF7ED60F8D2  mov         eax,15h
00007FF7ED60F8D7  add         rsp,40h
00007FF7ED60F8DB  pop         rbx
00007FF7ED60F8DC  ret
```

Basically, this just wraps another function inside of Microsoft Flight Simulator, which looks like it might do the real work.

At this point, I am satisfied that we understand how the WASM runtime works in Microsoft Flight Simulator as well as how system calls are implemented. Going forward, we now know enough to reverse engineer WASM code generation, the inNative process to generate the PE containing the WASM runtime, the PE itself, as well as any system calls. Armed with this knowledge we can now debug the full WASM stack to correctly isolate and identify any bug that we might encounter in any part of the compilation or execution process.