APPENDIX E

Linking Assembly and C++

Objective

Appendix E describes how to create a stub and driver for mixing Assembly and C++ code. Examples are provided for Windows using Visual Studio and MASM, for macOS using Xcode and Clang/LLVM (GAS compatible), and for Linux using NASM.

Windows—Visual Studio—MASM

- Open Visual Studio and create a new empty C++ project.
- Add a new file named something such as *main.cpp*.
- Type the following code in *main.cpp*.

```
// main.cpp
extern "C" void asmMain();

int main(){
      asmMain();
      return 0;
}
```

- Add another new file named something like *code.asm*.
- Type the following code in *code.asm*.

MASM (32-bit)	MASM (64-bit)

`; code.asm`
`.386`
`.MODEL FLAT, stdcall`
`.STACK 4096`
`.CODE`
`asmMain PROC C`
`    ; your code will go here`
`    ret`
`asmMain ENDP`
`END`
	`; code.asm`
	`.CODE`
	`asmMain PROC`
	`      ; your code will go here`
	`      ret`
	`asmMain ENDP`
	`END`

- The 32-bit code defines the `asmMain` procedure to use the *cdecl* calling convention, which could be set for all procedures by changing the `.MODEL` directive to be `.MODEL FLAT, C`. The 64-bit code needs no explicit call type because only one exists, the x64 calling convention.
- Be sure to set the Solution Platform to the appropriate setting. The default is x86, so if you are writing x64 code, switch to x64.

Automated Approach (preferred)

- Once the code files have been created, right-click on the project in the Solution Explorer and select "Build Dependencies" → "Build Customizations." Check the "masm" box and click "OK."
- Right-click on the Assembly file (e.g., *code.asm*), and in the "Configuration Properties" → "General" click on the "Excluded From Build" drop-down and select "No." Then click on "Item Type" and select "Microsoft Macro Assembler." The option will only show up if you followed the previous step! Apply the changes and click "OK."
- Now, you can test by setting debugging breakpoints on both the C++ side and the Assembly side, run the code, and everything should work fine.
- If you create your own functions, do not forget to add the prototype in the .asm file *before* the `.DATA` segment.

  32-bit Example: `inputInteger PROTO C`
  64-bit Example: `inputInteger PROTO`

Manual Approach (good to know how to manually assemble a file)

- Open the **Developers Command Prompt**, which can be found by searching for "developer" or "command prompt" in the Windows Start Menu or by navigating to the appropriate location. The path may vary depending on your Visual Studio version and installation location. We have provided an example path and a couple of MSDN links for more information.

  Path: `C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\Common 7\Tools`
  Links: : https://docs.microsoft.com/en-us/dotnet/framework/tools/developer-command-prompt-for-vs
  https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line
- In the Developers Command Prompt, change directory (`cd`) into the folder that contains *code.asm*, and then run the appropriate command to assemble the file and produce an object file.

32-bit	64-bit
`ml /c /Cx /coff code.asm`	`ml64 /c /Cx code.asm`

`ml` = MASM 32-bit Assembler and linker
Example Path: `C:\Program Files(x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.26.28801\bin\Hostx64\x86\ml.exe`

`ml64` = MASM 64-bit Assembler and linker
Example Path: `C:\Program Files(x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.26.28801\bin\Hostx64\x64\ml64.exe`

`/c` = assemble only, do not link
`/Cx` = preserves case in public and external symbols
`/coff` = object file format (required in 32-bit, not for 64-bit)

`ml` and `ml64` Command-Line Reference: https://docs.microsoft.com/en-us/cpp/assembler/masm/ml-and-ml64-command-line-reference

Depending on the version of Visual Studio, Windows, and installation settings, you may need to add the paths for `ml` or `ml64` to the system PATH Environment Variable. We provide some helpful links.

Setting the Path and Environment Variables for Command-Line Builds:
https://docs.microsoft.com/en-us/cpp/build/setting-the-path-and-environment-variables-for-command-line-builds

General instructions for adding a PATH environment variable:
https://docs.microsoft.com/en-us/previous-versions/office/developer/sharepoint-2010/ee537574(v=office.14)

- Back in Visual Studio, add an "Existing item…" to the project and select the *code.obj* file created in the previous step.
- Build and run the program to verify that it works.
- You can now add code in *code.asm* that calls C/C++ functions defined or included in *main.cpp* and vice versa.
- Reminder: You *must* re-assemble the *.asm* file after any Assembly code changes for the changes to take effect when rebuilding and running the program. You can either go back to the Developers Command Prompt, press the up arrow, and re-run the command or automate the process by right-clicking on the project in the Solution Explorer, go to "Properties" → "Build Events" → "Pre-Link Event," click on the "Command Line" drop-down, select "<Edit…>" and enter the appropriate command given the platform.
- Debugging breakpoints will work on the C++ side, but may not work on the Assembly side. One way to set a debugging breakpoint on the Assembly side, is to use the `int 3` instruction where breaks are desired.

macOS – Xcode – Clang/LLVM (GAS compatible)

- Open Xcode and create a new empty C++ project (macOS → Command Line Tool).
- A default *main.cpp* is provided, so just replace the code with the following starter code.

```
// main.cpp
extern "C" void asmMain();

int main(){
      asmMain();
      return 0;
}
```

- Add a new file named something such as *code.s*. The easiest way to add a file is to right-click on the project folder where *main.cpp* is contained and select "New File…" and then scroll down to the "Other" category and select "Assembly File."
- Type the following code in *code.s*.

```
# code.s
.data

.bss

.text
.globl _asmMain
_asmMain:
      # your code will go here
ret
.end
```

- Optional: Click on the project in the Project Navigator and go to "Build Settings." Scroll down to "Apple Clang—Custom Compiler Flags" and in "Other C Flags" add "`-mstackrealign`." The flag realigns the stack for function calls, which allows for mixing legacy (4-byte aligned) and modern (16-byte aligned) code. Both 32-bit and 64-bit code on macOS must be 16-byte aligned. If you are only using 64-bit code, then the flag is not needed, as 16-byte alignment happens by default.
- Build and run the program to verify that it works (the program works for both 32-bit and 64-bit).
- You can now add code in *code.s* that calls C/C++ functions defined or included in *main.cpp* and vice versa.
- Prefix function names in *code.s* with an *underscore*, even if a function does not begin with an underscore in the *.cpp* file.

Linux—NASM

Using your preferred development environment or editor,

- Create a new file named something such as *main.cpp*.
- Type the following code in *main.cpp*.

```
// main.cpp
extern "C" void asmMain();

int main(){
      asmMain();
      return 0;
}
```

- Create another new file named something such as *code.asm*.
- Type the following code in *code.asm*.

```
; code.asm
SECTION .data

SECTION .bss

SECTION .text
global asmMain
asmMain:
      ; your code will go here
ret
```

- Build and run the program to verify that it works. The following assumes *gcc* and *g++ multilib* are installed on a 64-bit Linux system.
  - 32-bit
    - Assemble: `nasm -f elf32 code.asm` (just using `elf` also implies 32-bit)
    - Compile and link `g++ -m32 main.cpp code.o -o test`
    - Run: `./test`
  - 64-bit
    - Assemble: `nasm -f elf64 code.asm`
    - Compile and link: `g++ main.cpp code.o -o test`
    - Alternative: `g++ -m64 main.cpp code.o -o test`
    - Run: `./test`
- You can now add code in *code.asm* that calls C/C++ functions defined or included in *main.cpp* and vice versa.

What is `extern "C"`?

Prefixing a C++ function with `extern "C"` turns off **name mangling**. Unlike the C language, C++ allows for *function overloading*: naming multiple functions the same name, assuming the parameter lists are different. Overloading works in C++ because the compiler will "mangle" function names so the system can distinguish between function calls. Mangling involves pre-fixing and post-fixing a function name with symbols that indicate information about the function. Almost every compiler mangles names differently.

Looking at the Disassembly in **Example 6.4 Stack Alignment Using NOP** in **Chapter 6** on line 24, notice that the name of the `sum` function has been modified to be `__Z3sumii`. For the Clang/LLVM compiler, `__Z` indicates a mangled symbol, `3` indicates the number of characters in the function name, `sum` is the original function name, and `ii` signifies that the function receives two integer parameters. Again, the mangling scheme varies from compiler to compiler. You can find mangling schemes online, and a suitable starting point is https://en.wikipedia.org/wiki/Name_mangling.

In order to call a C++ function from Assembly, you need to know the name of the function post-compilation. So unless you want to memorize mangling conventions, turn off name mangling so you can use the name of the function as declared by the programmer (you). Any C++ functions you create for calling from within Assembly and vice versa should be marked as `extern "C"`.