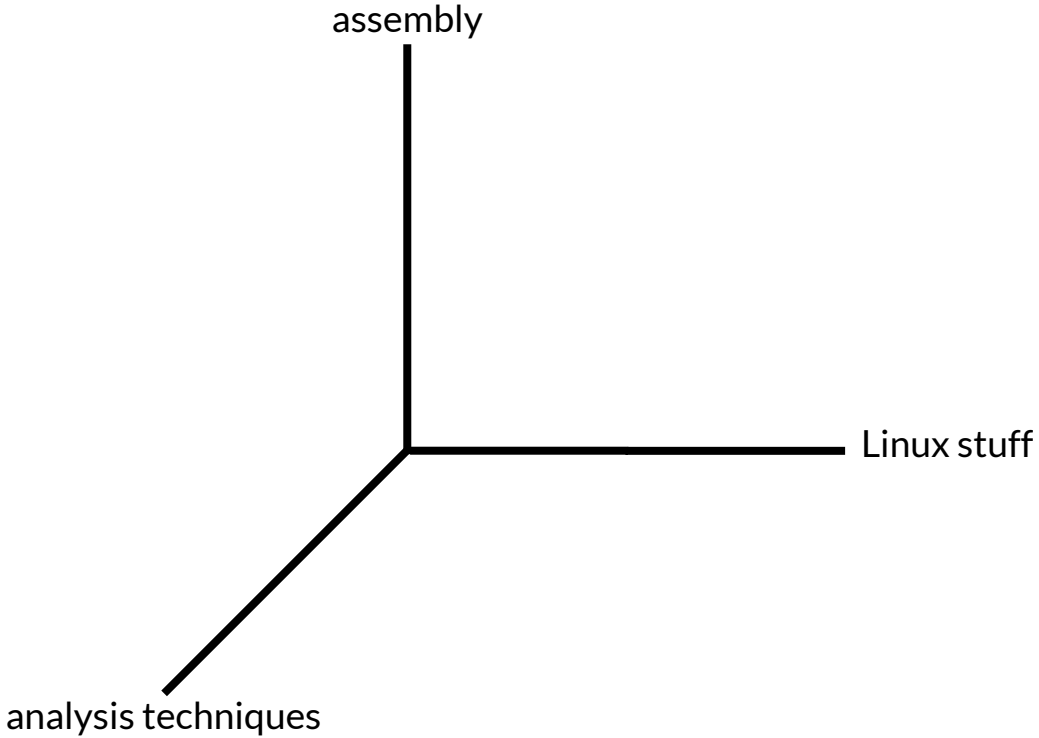
CSC 6580
Spring 2020

Instructor: Stacy Prowell

The Class in 3D



Assembly: Jump Tables

Case Statements

```
switch(x) {
  case 0: f();
  case 1: g(); break;
  case 2: h(); break;
  default: j();
}
```

What will the compiler do here?

It will *probably* just create a sequence of branches as this is very simple code, but it might also create a *jump table*.

Case Statements and Jump Tables

```
switch(x) {
  case 0: f();
  case 1: g(); break;
  case 2: h(); break;
  default: j();
}
```

We can implement this as a **jump table**. This is a common structure.

```
section .data
jumptab: dq targetf, targetg, targeth

section .text
     ; ...
     cmp rax, 3
     ja .def
     jmp [jumptab + rax*8]
.def:
     jmp targetj
```

ELF, the entry point, and main

Very Simple File

Let's examine a very simple program: hello world.

We can compile this with:

```
ac () { eval $( head -1 $1 | cut -c3- ) }
```

This produces a file about 9,000 bytes in length.

```
; nasm -f elf64 hello.asm && ld -o hello hello.o

        section .text
        global _start

_start:
        mov rdi, 1
        mov rsi, msg1
        mov rdx, msg1.len
        mov rax, 1
        syscall

        mov rdi, 1
        mov rsi, msg2
        mov rdx, msg2.len
        mov rax, 1
        syscall

        mov rdi, 0
        mov rax, 60
        syscall
        hlt

        section .data nowrite align=16
msg1:   db 'Hello world!',10
.len:   equ $-msg1
msg2:   db 'Goodbye world!',10
.len:   equ $-msg2
```

Very Simple File

Define a value. The dollar sign (**$**) refers to the *current address*.

```
; nasm -f elf64 hello.asm && ld -o hello hello.o

        section .text
        global _start

_start:
        mov rdi, 1
        mov rsi, msg1
        mov rdx, msg1.len
        mov rax, 1
        syscall

        mov rdi, 1
        mov rsi, msg2
        mov rdx, msg2.len
        mov rax, 1
        syscall

        mov rdi, 0
        mov rax, 60
        syscall
        hlt

        section .data nowrite align=16
msg1:   db 'Hello world!',10
.len:   equ $-msg1
msg2:   db 'Goodbye world!',10
.len:   equ $-msg2
```

Very Simple File

You can specify additional properties for your sections. Here we make `.data` read-only (it is normally writeable) and force it to be aligned on a 16-byte boundary.

```
; nasm -f elf64 hello.asm && ld -o hello hello.o

        section .text
        global _start

_start:
        mov rdi, 1
        mov rsi, msg1
        mov rdx, msg1.len
        mov rax, 1
        syscall

        mov rdi, 1
        mov rsi, msg2
        mov rdx, msg2.len
        mov rax, 1
        syscall

        mov rdi, 0
        mov rax, 60
        syscall
        hlt

        section .data nowrite align=16
msg1:   db 'Hello world!',10
.len:   equ $-msg1
msg2:   db 'Goodbye world!',10
.len:   equ $-msg2
```

Very Simple File

We can get an idea of what's in it with `objdump`.

```
$ objdump -s hello

hello:     file format elf64-x86-64

Contents of section .text:
 401000 bf010000 0048be00 20400000 000000ba  .....H.. @......
 401010 0d000000 b8010000 000f05bf 01000000  ................
 401020 48be0d20 40000000 0000ba0f 000000b8  H.. @...........
 401030 01000000 0f05bf00 000000b8 3c000000  ............<...
 401040 0f05f4                               ...
Contents of section .data:
 402000 48656c6c 6f20776f 726c6421 0a476f6f  Hello world!.Goo
 402010 64627965 20776f72 6c64210a           dbye world!.
```

Very Simple File

We can get an idea of what's in it with `objdump`.

It's *really simple*. There are just two sections: `.text` and `.data`.

Sections eventually get mapped to *segments*. You can have any sections you want, and can specify permissions on them.

```
section .special write align=4
```

```
$ objdump -s hello

hello:     file format elf64-x86-64

Contents of section .text:
 401000 bf010000 0048be00 20400000 000000ba  .....H.. @......
 401010 0d000000 b8010000 000f05bf 01000000  ................
 401020 48be0d20 40000000 0000ba0f 000000b8  H.. @...........
 401030 01000000 0f05bf00 000000b8 3c000000  ............<...
 401040 0f05f4                               ...
Contents of section .data:
 402000 48656c6c 6f20776f 726c6421 0a476f6f  Hello world!.Goo
 402010 64627965 20776f72 6c64210a           dbye world!.
```

Very Simple File

We can get an idea of what's in it with `objdump`.

It's *really simple*. There are just two sections: `.text` and `.data`.

The numbers at the start of each line are *virtual addresses* and *not* offsets into the file.

```
$ objdump -s hello

hello:     file format elf64-x86-64

Contents of section .text:
 401000 bf010000 0048be00 20400000 000000ba  .....H.. @......
 401010 0d000000 b8010000 000f05bf 01000000  ................
 401020 48be0d20 40000000 0000ba0f 000000b8  H.. @...........
 401030 01000000 0f05bf00 000000b8 3c000000  ............<...
 401040 0f05f4                               ...
Contents of section .data:
 402000 48656c6c 6f20776f 726c6421 0a476f6f  Hello world!.Goo
 402010 64627965 20776f72 6c64210a           dbye world!.
```

Very Simple File

We can get an idea of what's in it with `objdump`.

It's *really simple*. There are just two sections: `.text` and `.data`.

The numbers at the start of each line are *virtual addresses* and *not* offsets into the file.

You can use `-F` to see the file offsets.

```
$ objdump -s hello -F

hello:     file format elf64-x86-64

Contents of section .text:  (Starting at file offset: 0x1000)
 401000 bf010000 0048be00 20400000 000000ba  .....H.. @......
 401010 0d000000 b8010000 000f05bf 01000000  ................
 401020 48be0d20 40000000 0000ba0f 000000b8  H.. @...........
 401030 01000000 0f05bf00 000000b8 3c000000  ............<...
 401040 0f05f4                               ...
Contents of section .data:  (Starting at file offset: 0x2000)
 402000 48656c6c 6f20776f 726c6421 0a476f6f  Hello world!.Goo
 402010 64627965 20776f72 6c64210a           dbye world!.
```

Very Simple File

We can get an idea of what's in it with `objdump`.

It's *really simple*. There are just two sections: `.text` and `.data`.

The numbers at the start of each line are *virtual addresses* and *not* offsets into the file.

The `-s` flag displays all *non-empty* sections. If you just want a few, use `-j` to specify those sections.

```
$ objdump -s hello -j .text

hello:     file format elf64-x86-64

Contents of section .text:
 401000 bf010000 0048be00 20400000 000000ba  .....H.. @......
 401010 0d000000 b8010000 000f05bf 01000000  ................
 401020 48be0d20 40000000 0000ba0f 000000b8  H.. @...........
 401030 01000000 0f05bf00 000000b8 3c000000  ............<...
 401040 0f05f4                               ...
```

Very Simple File

We can get the file headers with `objdump -f`, or (better yet) with `readelf -h`.

There are actually 6 sections in this file!

```
$ readelf -h hello

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little
endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices
X86-64
  Version:                           0x1
  Entry point address:               0x401000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8616 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         6
  Section header string table index: 5
```

Very Simple File

Here are the six sections, as per `readelf`.

```
$ readelf -S hello
There are 6 section headers, starting at offset 0x21a8:

Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        0000000000401000  00001000
       0000000000000043  0000000000000000  AX       0     0     16
  [ 2] .data             PROGBITS        0000000000402000  00002000
       000000000000001c  0000000000000000   A       0     0     16
  [ 3] .symtab           SYMTAB          0000000000000000  00002020
       0000000000000120  0000000000000018           4     8     8
  [ 4] .strtab           STRTAB          0000000000000000  00002140
       000000000000003f  0000000000000000           0     0     1
  [ 5] .shstrtab         STRTAB          0000000000000000  0000217f
       0000000000000027  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T
(TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

Very Simple File

The entry point is the (virtual) address where the program begins executing. In our case it is `0x401000`.

```
$ readelf -h hello

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little
endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices
X86-64
  Version:                           0x1
  Entry point address:               0x401000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8616 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         6
  Section header string table index: 5
```

Very Simple File

The entry point is the (virtual) address where the program begins executing. In our case it is `0x401000`.

Dumping the symbol table, we see that this is the address of `_start`.

```
$ readelf -s hello

Symbol table '.symtab' contains 12 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000401000     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000402000     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.asm
     4: 0000000000402000     0 NOTYPE  LOCAL  DEFAULT    2 msg1
     5: 000000000000000d     0 NOTYPE  LOCAL  DEFAULT  ABS msg1.len
     6: 000000000040200d     0 NOTYPE  LOCAL  DEFAULT    2 msg2
     7: 000000000000000f     0 NOTYPE  LOCAL  DEFAULT  ABS msg2.len
     8: 0000000000401000     0 NOTYPE  GLOBAL DEFAULT    1 _start
     9: 000000000040201c     0 NOTYPE  GLOBAL DEFAULT    2 __bss_start
    10: 000000000040201c     0 NOTYPE  GLOBAL DEFAULT    2 _edata
    11: 0000000000402020     0 NOTYPE  GLOBAL DEFAULT    2 _end
```

Find the Entry Point

Let's try this for a real file: `ls`.

First… there is no `_start` symbol!
(You can do this, too, with a linker script.)

There must still be an entry point. Let's find it.

```
$ readelf -s `which ls` | grep _start
    53: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (3)
    65: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
```

Find the Entry Point

The entry point is `0x67d0`.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x67d0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          140224 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 29
```

Disassembling at the entry point

Let's try to disassemble `` `which ls` ``.

The code at the entry point ends with a `call` to `[rip + 0x1be1e]`

This is an `RIP`-indexed instruction.

A few were possible in the 32-bit world, but this is *very common* in the 64-bit world.

Where does it go???

```
67d0: f3 0f 1e fa             endbr64
67d4: 31 ed                   xor    ebp,ebp
67d6: 49 89 d1                mov    r9,rdx
67d9: 5e                      pop    rsi
67da: 48 89 e2                mov    rdx,rsp
67dd: 48 83 e4 f0             and    rsp,0xfffffffffffffff0
67e1: 50                      push   rax
67e2: 54                      push   rsp
67e3: 4c 8d 05 66 0d 01 00    lea    r8,[rip+0x10d66]
67ea: 48 8d 0d ef 0c 01 00    lea    rcx,[rip+0x10cef]
67f1: 48 8d 3d f8 e5 ff ff    lea    rdi,[rip+0xfffffffffffe5f8]
67f8: ff 15 d2 c7 01 00       call   QWORD PTR [rip+0x1c7d2]
67fe: f4                      hlt
```

Disassembling at the entry point

Take the start address of the next instruction… `0x67fe`

```
67d0: f3 0f 1e fa             endbr64
67d4: 31 ed                   xor    ebp,ebp
67d6: 49 89 d1                mov    r9,rdx
67d9: 5e                      pop    rsi
67da: 48 89 e2                mov    rdx,rsp
67dd: 48 83 e4 f0             and    rsp,0xfffffffffffffff0
67e1: 50                      push   rax
67e2: 54                      push   rsp
67e3: 4c 8d 05 66 0d 01 00    lea    r8,[rip+0x10d66]
67ea: 48 8d 0d ef 0c 01 00    lea    rcx,[rip+0x10cef]
67f1: 48 8d 3d f8 e5 ff ff    lea    rdi,[rip+0xfffffffffffe5f8]
67f8: ff 15 d2 c7 01 00       call   QWORD PTR [rip+0x1c7d2]
67fe: f4                      hlt
```

Disassembling at the entry point

Take the start address of the next instruction… `0x67fe`

…and add the displacement.

```
   0x1c7d2
+  0x 67fe
   0x22fd0
```

This is the target of the call… but it is outside the program!

```
67d0: f3 0f 1e fa             endbr64
67d4: 31 ed                   xor    ebp,ebp
67d6: 49 89 d1                mov    r9,rdx
67d9: 5e                      pop    rsi
67da: 48 89 e2                mov    rdx,rsp
67dd: 48 83 e4 f0             and    rsp,0xfffffffffffffff0
67e1: 50                      push   rax
67e2: 54                      push   rsp
67e3: 4c 8d 05 66 0d 01 00    lea    r8,[rip+0x10d66]
67ea: 48 8d 0d ef 0c 01 00    lea    rcx,[rip+0x10cef]
67f1: 48 8d 3d f8 e5 ff ff    lea    rdi,[rip+0xfffffffffffe5f8]
67f8: ff 15 d2 c7 01 00       call   QWORD PTR [rip+0x1c7d2]
67fe: f4                      hlt
```

Starting the C runtime

This call is to `__libc_start_main`.

```
int __libc_start_main(
    int *(main) (int, char **, char **),  // This is the main function.
    int argc,                             // Number of command line arguments.
    char ** ubp_av,                       // The command line arguments (unbounded).
    void (*init) (void),                  // The initialization function.
    void (*fini) (void),                  // The finalization function.
    void (*rtld_fini) (void),             // Finalize dynamic shared objects.
    void (* stack_end));                  // The end of the stack.
```

Let's apply the calling convention.

Starting the C runtime

This call is to `__libc_start_main`.

```
int __libc_start_main(
    int *(main) (int, char **, char **),  // rdi
    int argc,                             // rsi
    char ** ubp_av,                       // rdx
    void (*init) (void),                  // rcx
    void (*fini) (void),                  // r8
    void (*rtld_fini) (void),             // r9
    void (* stack_end));                  // on stack
```

Starting the C runtime

At entry (from the loader):

- `RDX` contains the address of the destructor function call handler for the dynamic linker, `_dl_fini`.
- The stack contains `argc`, `argv`, and `envp`, with `argc` on top.

When we call `main`, we at least want:

- `EDI` to contain `argc`
- `ESI` to contain the pointer to `argv`
- `EDX` to contain the pointer to `envp`

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

Let's map assembly instructions to arguments.

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

This is a branch protection instruction. Think of it as a no-op. Technically it is!

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

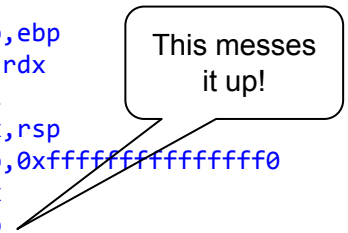
```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

16-bit stack alignment

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

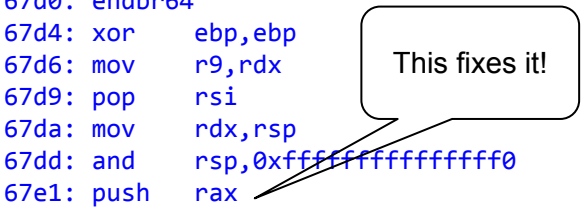


Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

Now the stack end (RSP) is at the top of an aligned stack.

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```



Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
2's comp.     0x10000
            - 0x e5f8
              0x 1a08 (disp)
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
main:        0x67f8 (rip)
           - 0x1a08 (disp)
             0x4df0
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
main:        0x67f8 (rip)
           - 0x1a08 (disp)
             0x4df0
```

```
>>> hex(0x67f8 - (0x10000-0xe5f8))
'0x4df0'
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

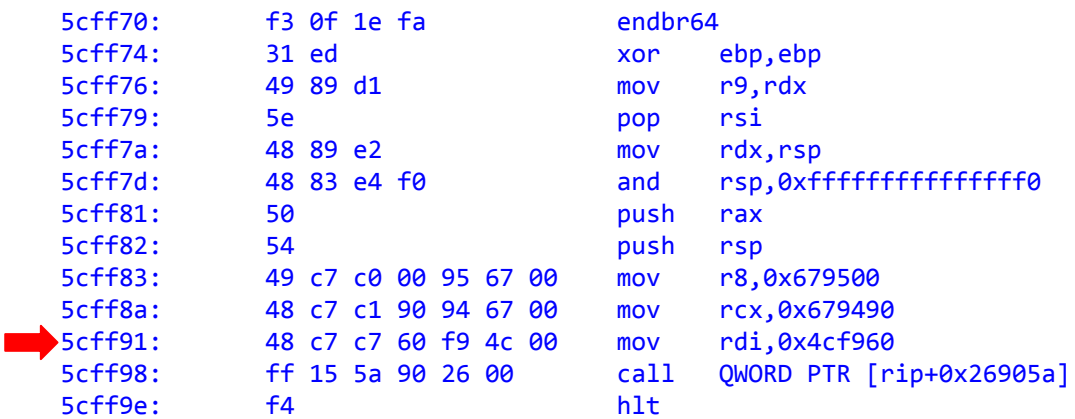
```
int __libc_start_main(
    int *(main) (int, char **, char **),      // rdi
    int argc,                                 // rsi
    char ** ubp_av,                           // rdx
    void (*init) (void),                      // rcx
    void (*fini) (void),                      // r8
    void (*rtld_fini) (void),                 // r9
    void (* stack_end));                      // on stack
```

This does not return! Remember that the top-level process (`_start`) needs to call `sys_exit`.

```
67d0: endbr64
67d4: xor    ebp,ebp
67d6: mov    r9,rdx
67d9: pop    rsi
67da: mov    rdx,rsp
67dd: and    rsp,0xfffffffffffffff0
67e1: push   rax
67e2: push   rsp
67e3: lea    r8,[rip+0x10d66]
67ea: lea    rcx,[rip+0x10cef]
67f1: lea    rdi,[rip+0xfffffffffffe5f8]
67f8: call   QWORD PTR [rip+0x1c7d2]
67fe: hlt
```

Starting the C runtime

Not always so indirect.



A Structuring Example

XOn / XOff Flow Control

RS232 (character oriented stream) flow control.

Essentially: Who gets to talk.

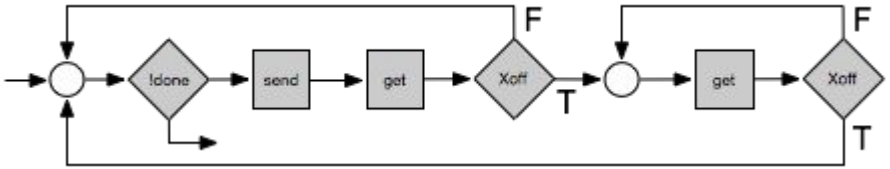
When the receiver buffer fills to the point it cannot accept any more data it sends an XOff (Transmit Off) to the transmitter. When the transmitter sees the XOff character, it stops transmitting. When the receiver can again accept data, it sends XOn. When the transmitter sees XOn, it resumes sending.

Normally we have:

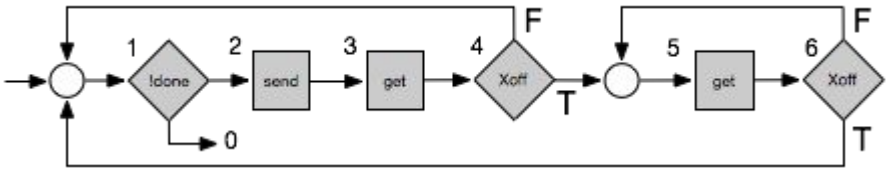
- XOn = CTRL+Q = 0x11
- XOff = CTRL+S = 0x13

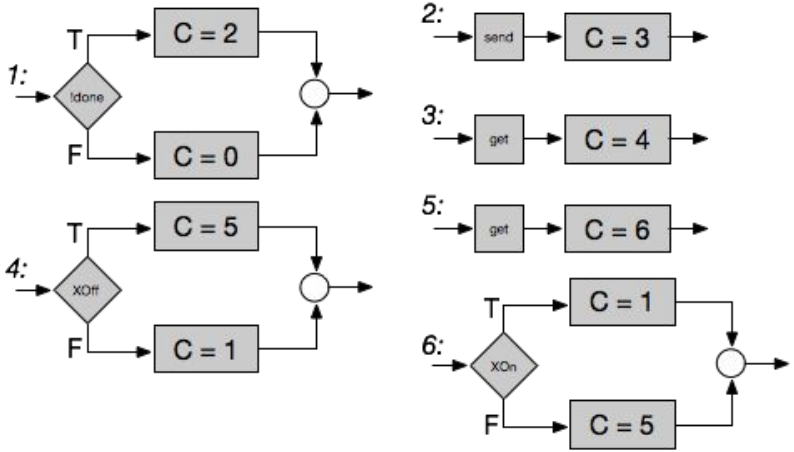
XOn / XOff Flow Control

```
for (i = 0; i < bufflen; ++i) {
    transmit_byte();
    b = receive_byte();
    if (b != XOFF) continue;
    while (b != XON) {
        b = receive_byte();
    }
}
```

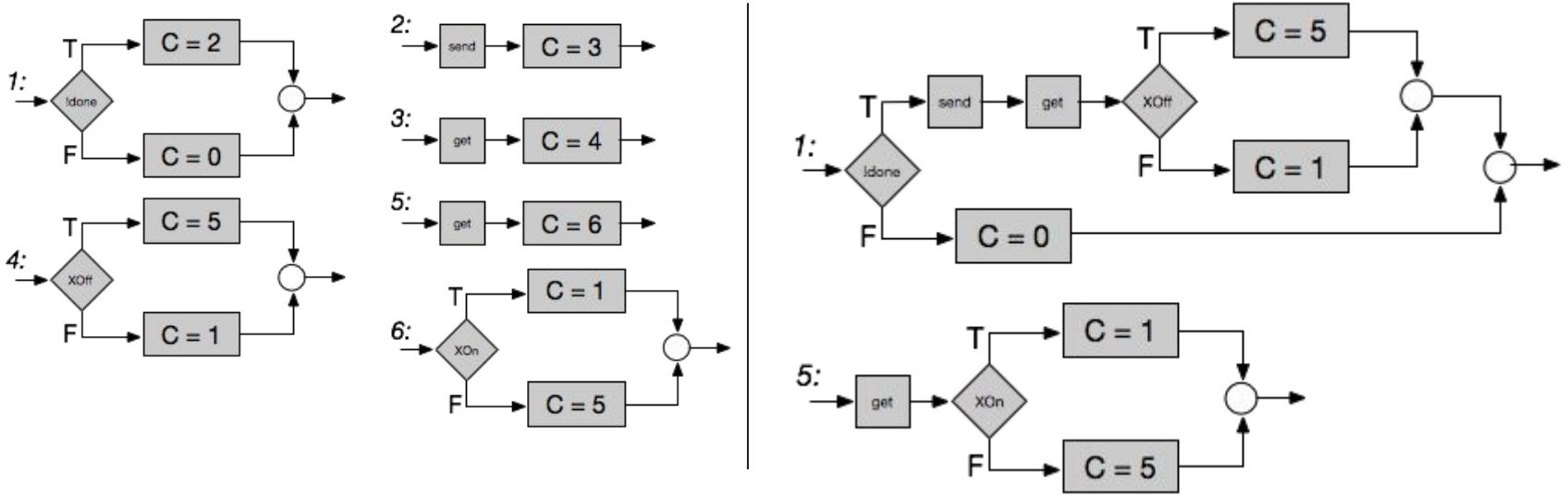


XOn / XOff Flow Control

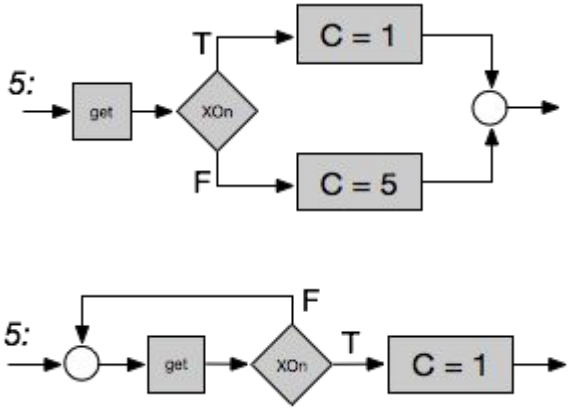




XOn / XOff Flow Control

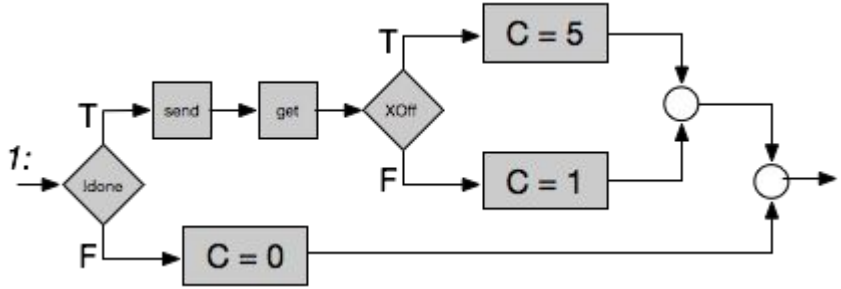


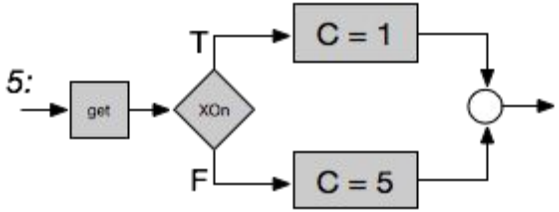
XOn / XOff Flow Control

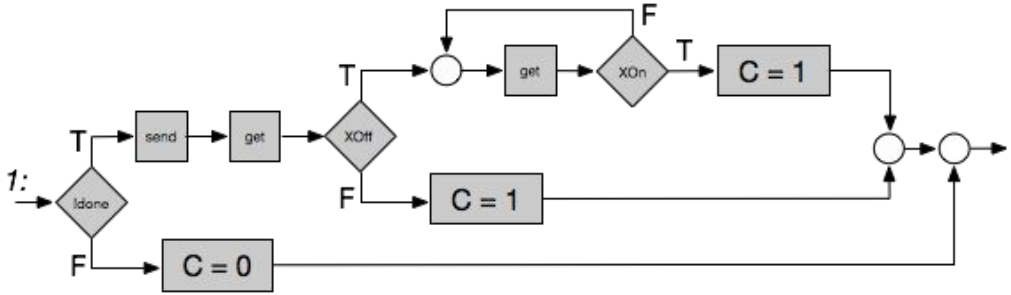



```
do {
    b = receive_byte();
} while (b != XON);
```

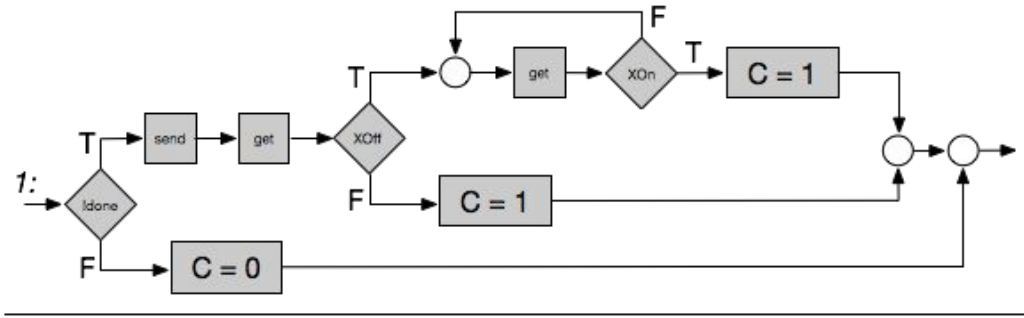
XOn / XOff Flow Control

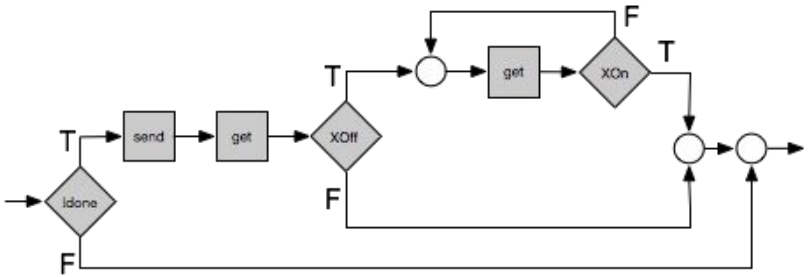






XOn / XOff Flow Control






```
for (i = 0; i < bufflen; ++i) {
    transmit_byte();
    b = receive_byte();
    if (b == XOFF) {
        while (b != XON) {
            b = receive_byte();
        }
    }
}
```

Next time:
Control Flow