A General Overview of What Happens Before main()

8 APRIL 2019 BY PHILLIP JOHNSTON • LAST UPDATED 22 AUGUST 2022

For most programmers, a C or C++ program's life begins at the `main` function. They are blissfully unaware of the hidden steps that happen between invoking a program and executing `main`. Depending on the program and the compiler, there are all kinds of interesting functions that get run before `main`, automatically inserted by the compiler and linker and invisible to casual observers.

Unfortunately for programmers who are curious about the program startup process, the literature on what happens before `main` is quite sparse.

Embedded Artistry has been hard at working creating a C++ embedded framework. The final piece of the puzzle was implementing program startup code. To aid in the design of our framework's boot process, I performed an exploratory survey of existing program startup implementations. My goal was to identify a general program startup model. I also want to provide a more comprehensive look into how our programs get to `main`.

In this six-part series, we will be investigating what it takes to get to `main`:

1. What Happens Before `main()?`
2. Exploring Startup Implementations: Newlib (ARM)
3. Exploring Startup Implementations: OS X
4. Exploring Startup Implementations: Custom Embedded System with ThreadX
5. Abstracting a Generic Flow for Getting to `main`
6. Implementing our Generic Startup Flow

generalized so that they apply to *many* systems. We will supplement the general theory in the following articles with an analysis of real-world implementations.

Table of Contents:

Getting to Main: A General Overview

Before we dive into our exploration of how existing systems get to `main`, we should develop a hypothesis about what generally happens. Since others have already explored program startup, we can start with a clear idea of what happens before `main`.

The `_start` Function

For most C and C++ programs, the true entry point is not `main`, it's the `_start` function. This function initializes the program runtime and invokes the program's `main` function.

The use of `_start` is merely a general convention. The entry function can vary depending on the system, compiler, and standard libraries. For example, OS X only has dynamically linked applications; the loader takes care of setup, and the entry point to the program is actually `main`.

The linker controls the program's entry point. The default entry point can be overridden by clang and GCC linkers using the `-e` flag, although this is rarely done for most programs.

The implementation of the `_start` function is usually supplied by `libc`. The `_start` function is often written in assembly. Many implementations store the `_start` function in a file called `crt0.s`. Compilers typically ship with pre-compiled `crt0.o` object files for each supported architecture.

Although much of this code is usually implemented by the C runtime, program startup code behavior is not specified by the C and C++ standards. Instead, the standards describe the conditions that must be true when the `main` function is called. However, there are many steps that are commonly performed across the majority of `_start` implementations.

At a high level, the `_start` function handles:

1. Early low-level initialization, such as:
    1. Configuring processor registers
    2. Initializing external memory
    3. Enabling caches
    4. Configuring the MMU
2. Stack initialization, making sure that the stack is properly aligned per the ABI requirements
3. Frame pointer initialization
4. Initialization of the C/C++ runtime
5. Initialization of other scaffolding required by the system
6. Jumping to `main`
7. Exiting the program with the return code from `main`

While the `_start` routine typically encompasses these activities, the specific order and implementation varies from system to system. For example, *early low-level initialization* code is commonly found with bare-metal embedded systems, but rarely on host machines with an OS. Your Linux or OS X program startup code will have multiple scaffolding functions which you will not find in embedded startup code.

The startup code below assumes that the program loader put:

- `*argv` and `*envp` variables on the stack
- `argc` in register `%rdi`
- `argv` in register `%rsi`
- `envc` in register `%rdx`
- `envp` in register `%rcx`

Here's the implementation of `_start`:

```
.section .text

.global _start
_start:
    # Set up end of the stack frame linked list
    movq $0, %rbp
    pushq %rbp # rip=0
    pushq %rbp # rbp=0
    movq %rsp, %rbp

    # Save argc and argv on the stack
    # We need those in a moment when we call main
    pushq %rsi
    pushq %rdi

    # Prepare signals, memory allocation, stdio, etc.
    call initialize_standard_library

    # Run the global constructors.
    call _init

    # Restore argc and argv before calling main
    popq %rdi
    popq %rsi

    # Run main
    call main

    # Terminate the process with the exit code
    # that was returned from main
```

Let's dive in and see what happens during the runtime setup process (`initialize_standard_library` above).

Runtime Setup

C/C++ runtime setup is a universal requirement for program startup. At a high level, our runtime setup must accomplish the following:

1. Relocate any relocatable sections (if not handled by the loader or linker)
2. Initializing global and static memory
3. Prepare the `argc` and `argv` variables for invoking `main` (even if it's just setting these to `0`/`NULL`)
4. Perform any additional setup steps required by the C/C++ standard library implementation.

Initializing global and static memory is broken down into two distinct steps that deserve additional details.

First, the runtime initializes a subset of *uninitialized* memory (no = in the declaration) to `0`. This includes global and static variables, but not stack variables. All uninitialized data that needs to be set to `0` is placed into the `.bss` section of the compiled program image by the linker. The location of the `.bss` section is identified during initialization, and the memory is typically set to `0` with `memset`.

Second, C++ global objects must be constructed before calling `main`. The linker places these constructors into the `.init`, `.init_array`, or `.ctors` section of the image. Some compilers also allow C and C++ functions to be marked as a constructor using a compiler attribute (e.g., `__attribute__((constuctor))`). The constructors are stored in a list by the linker. The runtime initialization process iterates through the list and calls each constructor.

These additional runtime initialization steps are run for many programs (but not all):

1. Heap initialization
2. Initialize `stdio` (i.e., `stdin`, `stdout`, `stderr`)
3. Initialize exception support (if using C++)

5. Prepare environment variables

In practice, the line between the responsibilities of `_start` and the C runtime initialization can be fuzzy. Some implementations of `_start` handle pieces of the runtime setup directly, such as setting the `.bss` section contents to `0` and calling global constructors. Other implementations handle these tasks as part of the C runtime setup routines.

Assembly files commonly found during this portion of the startup process are `crtbegin.s`, `crtend.s`, `crti.s`, and `crtn.s`. Compilers often ship pre-compiled object files for supported architectures. These files are related to calling global constructors and destructors. When the files are not used, equivalent functionality is often implemented in C and invoked during runtime initialization.

Other Scaffolding

The setup process may invoke other functions to set up program scaffolding that the system requires. Program scaffolding setup before `main` might include:

1. Threading support and thread local storage
2. Buffer overrun detection
3. Stack logging
4. Run-time error checks
5. Locale settings
6. Math error handling
7. Default math library precision

The specific scaffolding functions invoked vary across standard library implementations and operating systems.

Jumping to `main`

Once we have a fully initialized system, we can safely jump to `main` and execute the programmer's portion of the application.

The most important aspect: once the program reaches `main`, it must be in a standards-

Returning From `main`

While we were primarily interested in how we get to `main`, we should finish our explanation of the `_start` function's responsibilities.

Because `_start` invokes `main`, it also handles its return. When control returns from `main` to `_start`, the next function to run is `exit`. The `exit` function calls all functions registered with `atexit`and `__cxa_atexit` during the startup process. Then `exit` calls the global destructors (those placed in the `.fini`, `.fini_array`, or `.dtors` sections). Finally, `exit` terminates the program with the return value provided by `main`.

The `exit` function is primarily used for hosted programs. Bare metal programs rarely have use for the `exit` function or global destructors.

How Do We Get to `_start`?

Now that we know how our program gets to `main` by way of the `_start` function, you may wonder how the program gets to `_start`.

There are three common paths:

1. Baremetal: reset vector
2. Bootloader launches application
3. OS Calls an `exec` function

Baremetal: Reset Vector

A baremetal embedded application represents the simplest path to `_start`.

Consider a baremetal platform with a binary stored in flash memory. When power is applied to the processor, the processor will copy the program from flash and store it in RAM[1]. Once the program is loaded into memory, the processor jumps to the reset interrupt vector address.

The embedded program's reset interrupt handler initializes the system after power-on or reset.

Some systems do not utilize the C standard library, and in that case `_start` will not be called. Instead, the reset handler will invoke other setup functions or will directly execute necessary program setup steps.

[1]: If the chip supports execute-in-place (XIP), the processor will skip the copy step and run the program directly from flash memory.

Bootloader Launches Application

Many embedded applications are composed of multiple distinct images which run sequentially during the boot process.

Many systems use a bootloader or hypervisor, which runs before loading and executing the main application. Bootloaders perform a wide range of activities, including initializing system hardware, decryption, decompression, checking that a firmware image is valid before loading it, selecting a firmware image to boot, or determining whether to enter firmware update mode. Bootloader complexity depends on the system's requirements; not of the listed tasks will be performed.

Other systems require an incremental boot process, especially when the main application is larger than the processor's internal RAM capacity. The first boot stage is typically a small image which fits into the processor's internal memory. This image will initialize external RAM and load the main application from flash into the external RAM. The first stage boot may perform additional steps, such as processor vector remapping or MMU configuration. Once the main application is loaded, the first stage boot invokes the reset vector of the main application.

Multi-stage boot scenarios complicate the program startup model. Each boot stage is technically a standalone program. However, not every stage will run through the full program startup process. Simple boot stages may only need to clear the `.bss` section to perform their duties, while complex bootloaders need a fully initialized C/C++ runtime. Program startup activities may be distributed across the boot process, with each stage handling specific tasks.

OS Calls an `exec` function

When you launch a program, your shell or GUI invokes a program loader. The loader is responsible for copying the application image from the hard drive into memory and configuring the environment that the program will run in. On Linux or OS X, the loader is a function in the `exec()` family, typically `execve()` or `execvp()`. For Windows, the loader is the `LdrInitializeThunk`function in `ntdll.dll`.

Loaders will often perform the following actions:

- Check permissions
- Allocate space for the program's stack
- Allocate space for the program's heap
- Initialize registers (e.g., stack pointer)
- Push `argc`, `argv`, and `envp` onto the program stack
- Map virtual address spaces
- Dynamic linking
- Relocations
- Call pre-initialization functions

Once the loader has configured the program environment, it calls the program's `_start` function.

Exploring On Your Own

In the coming, we will review a selection of startup procedures which differ greatly in terms of process and style.

We won't be reviewing Linux program startup, because there are already high-quality articles on that topic. For detailed descriptions about how Linux programs start, we recommend these articles:

1. How Programs Get Run
2. How Programs Get Run: ELF Binaries
3. Linux x86 Program Start Up or – How the heck do we get to main()?

surprised if you find a different startup process than those described in this series and in other articles around the web.

You can explore your own program's startup behavior using `objdump` or a debugger (I.e. `gdb`, `lldb`). You can use debugging tools to tackle the problem from a variety of directions:

1. Set a breakpoint at `main()` and run a backtrace to see the function call stack
2. Set a breakpoint at `_start()` (or whatever entry point your backtrace shows) and step through the execution
3. Dump the assembly output for the program using `objdump`

As Daniel Näslund pointed out in the comments, your debugger may be configured to suppress backtraces that go past the `main` function. For `gdb`, you can run the following command:

```
(gdb) set backtrace past-main on
```

Further Reading

- Matt Godbolt – The Bits Between the Bits: How We Get to main()
- Embedded Artistry libc
- Real Time C++: Chapter 8 and Section 3.6.2
- How Programs Get Run
- How Programs Get Run: ELF Binaries
- Executing main in C/C++: Behind the Scenes
- Linux x86 Program Start Up or How the heck do we get to main()?
- The C Runtime Initialization, crt0.o
- What Happens Before Main
- OSDev Wiki: Creating a C Library
- OSDev Wiki: Calling Global Constructors
- OSDev Wiki: C++
- Linkers and Loaders
- Wikipedia: Loader (Computing)

- Wikipedia: `.bss`
- Memfault: Zero to Main() Series
    - Bare Metal C
    - How to Write a Bootloader from Scratch

UNCATEGORIZED

BOOT PROCESS, C, CPP, FEATURED, LESSONS FROM THE FIELD

6 Replies to “A General Overview of What Happens Before main()”

Bryce Schober

8 APRIL 2019 AT 22:16

Probably deserved a comment about static initialization, and where that fits in, and its implications…

Phillip Johnston

8 APRIL 2019 AT 22:49

Are you referring to initializing static memory in the .bss section to 0, or the invocation of global constructors?

Both are listed, but certainly deserve more commentary. They will also be expanded upon in the next 5 articles, since those are major parts of the boot flow.

Daniel Näslund

9 APRIL 2019 AT 05:58

Great article, as always!

With the GDB debugger, the default settings (atleast on my Ubuntu 18.10 box) is to not show backtraces past main. That can be changed with the set backtrace past-main command:

$ gdb a.out
(gdb) start
Temporary breakpoint 1, 0x0000555555554627 in main ()
(gdb) bt

0 0x0000555555554627 in main ()

(gdb) set backtrace past-main on
(gdb) bt

0 0x0000555555554627 in main ()

1 0x00007ffff7a05b97 in __libc_start_main (main=0x555555554623 <main>, argc=1, argv=0x7fffffffde98, init=<optimized out>, fini=<optimized out>, rtld_fini=<optimized out>, stack_end=0x7fffffffde88) at ../csu/libc-start.c:310

2 0x000055555555451a in _start ()

There are a ton of resources about program startup and you've done a good job summarizing the https://lwn.net/Articles/631631/. It’s quite likely that your planned article on OS X startup will overlap with these.

Phillip Johnston

9 APRIL 2019 AT 17:29

Thanks Daniel, I added the gdb command to the article.

I had not seen those two LWN articles (amazing what you can miss even with a targeted search). Thanks for sharing them, I have incorporated them into the article and further reading links.

Andrew K.

10 APRIL 2019 AT 21:50

Awesome article! I can’t wait for the next three.

Muldimalph

4 JUNE 2021 AT 01:10

Could you give some tips to a newbie who is beginner in RE and have some knowledge of c and want to understand this article properly