From Zero to main(): Bootstrapping libc with Newlib

12 Nov 2019 by **François Baldassari**

This is the fourth post in our Zero to main() series, where we worked methodically to demystify what happens to firmware before the main() function is called. So far, we bootstrapped a C environment, wrote a linker script from scratch, and implemented our own bootloader.

And yet, we cannot even write a hello world program! Consider the following `main.c` file:

```c
#include <stdio.h>

int main() {
  printf("Hello, World\n");
  while (1) {}
}
```

Compiling this using our Makefile and linker script from previous posts, we hit the following error:

```
$ make
...
Linking build/minimal.elf
arm-none-eabi/bin/ld: build/objs/a/b/c/minimal.o: in function `main':
/minimal/minimal.c:4: undefined reference to `printf'
collect2: error: ld returned 1 exit status
make: *** [build/minimal.elf] Error 1
```

Undefined reference to `printf`! How could this be? Our firmware's C environment is still missing a key component: a working C standard library. This means that commonly used functions such as `printf`, `memcpy`, or `strncpy` are all out of reach of our program so far.

In firmware-land, nothing comes free with the system: just like we had to explicitly zero out the `bss` region to initialize some of our static variables, we'll have to port a `printf` implementation alongside a C standard library if we want to use it.

In this post, we will add RedHat's Newlib to our firmware and highlight some of its features. We will implement syscalls, learn about constructors, and finally print out "Hello, World"! We will also learn how to replace parts or all of the standard C library.

Table of Contents

Setup

As we did in our previous posts, we are using Adafruit's Metro M0 development board to run our examples. We use a cheap CMSIS-DAP adapter and openOCD to program it.

You can find a step by step guide in our previous post.

As with previous examples, we start with our "minimal" example which you can find on GitHub. I've reproduced the source code for `main.c` below:

```
#include <samd21g18a.h>
#include <port.h>
```

```c
#include <string.h>

#define LED_0_PIN PIN_PA17

static void set_output(const uint8_t pin) {
  struct port_config config_port_pin;
  port_get_config_defaults(&config_port_pin);
  config_port_pin.direction = PORT_PIN_DIR_OUTPUT;
  port_pin_set_config(pin, &config_port_pin);
  port_pin_set_output_level(pin, false);
}

int main() {
  memcpy(NULL, NULL, 0);
  set_output(LED_0_PIN);
  while (true) {
    port_pin_toggle_output_level(LED_0_PIN);
    for (int i = 0; i < 100000; ++i) {}
  }
}
```


Implementing Newlib

Why Newlib?

There are several implementations of the C Standard Library, starting with the venerable `glibc` found on most GNU/Linux systems. Alternative implementations include Musl libc[1], Bionic libc[2], ucLibc[3], and dietlibc[4].

Newlib is an implementation of the C Standard Library targeted at bare-metal embedded systems that is maintained by RedHat. It has become the de-facto standard in embedded software because it is complete, has optimizations for a wide range of architectures, and produces relatively small code.

Today Newlib is bundled alongside toolchains and SDK provided by vendors such as ARM (`arm-none-eabi-gcc`) and Espressif (ESP-IDF for ESP32).

> *Note: when code-size constrained, you may choose to use a variant of newlib, called newlib-nano, which does away with some C99 features, and some `printf` bells and whistles to deliver a more compact standard library. Newlib-nano is enabled*

with the `--specs=nano.specs` CFLAG. You can read more about it in our code size blog post

Enabling Newlib

14:47

Newlib is enabled **by default** when you build a project with `arm-none-eabi-gcc`. Indeed must explicitly opt-out with `-nostdlib` if you prefer to build your firmware without it.

This is what we do for our "minimal" example, to guarantee we do not include any libc functionality by mistake.

```
PROJECT := minimal
BUILD_DIR ?= build

CFLAGS += -nostdlib

SRCS = \
        startup_samd21.c \
        $(PROJECT).c

include ../common-standalone.mk
```

It is very easy to add a dependency on the C standard library without meaning to, as GCC will sometimes use standard C functions implicitly. For example, consider this code used to zero-initialize a struct:

```
int main() {
  int b[50] = {0}; // zero initialize a struct
  /* ... */
}
```

We added no new `#include`, nor any call to C library functions. Yet if we compile this code with `-nostdlib`, we'll get the following error:

```
...
Linking build/minimal.elf
/usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/a
/minimal/minimal.c:16: undefined reference to `memset'
collect2: error: ld returned 1 exit status
make: *** [build/minimal.elf] Error 1
```

If we remove `-nostdlib`, the program compiles and link without problems.

```
Linking build/minimal.elf
arm-none-eabi-objdump -D build/minimal.elf > build/minimal.lst
arm-none-eabi-objcopy build/minimal.elf build/minimal.bin -O binary
arm-none-eabi-size build/minimal.elf
   text    data     bss     dec     hex filename
   1292       0    8192    9484    250c build/minimal.elf
```


So here we are, using Newlib, and we did not have to do anything. Could it really be this simple?

> *Note: the variant of Newlib bundled with `arm-none-eabi-gcc` is not compiled with `-g`, which can make debugging difficult. For that reason, you may chose to replace it with your own build of Newlib. You can read more about that process in the Implementing our own C standard library section of this article.*

System Calls

Not so fast! Let's go back to our "Hello World" example:

```
int main() {
  printf("Hello World!\n");
  while(1) {}
}
```

Removing `-nostdlib` is not quite enough. Instead of `printf` being undefined, we now see a whole mess of undefined symbols:

```
Linking build/minimal.elf
/usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/a
a(lib_a-sbrkr.o): in function `_sbrk_r':
sbrkr.c:(.text._sbrk_r+0xc): undefined reference to `_sbrk'
/usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/a
a(lib_a-writer.o): in function `_write_r':
writer.c:(.text._write_r+0x10): undefined reference to `_write'
/usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/a
a(lib_a-closer.o): in function `_close_r':
closer.c:(.text._close_r+0xc): undefined reference to `_close'
```

```
/usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/a
a(lib_a-lseekr.o): in function `_lseek_r':
lseekr.c:(.text._lseek_r+0x10): undefined reference to `_lseek'
/usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/gcc/a
a(lib_a-readr.o): in function `_read_r':
readr.c:(.text._read_r+0x10): undefined reference to `_read'
/usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/
a(lib_a-fstatr.o): in function `_fstat_r':
fstatr.c:(.text._fstat_r+0xe): undefined reference to `_fstat'
/usr/local/Cellar/arm-none-eabi-gcc/8-2018-q4-major/gcc/bin/../lib/
a(lib_a-isattyr.o): in function `_isatty_r':
isattyr.c:(.text._isatty_r+0xc): undefined reference to `_isatty'
collect2: error: ld returned 1 exit status
```


Specifically, the compiler is asking for `_fstat`, `_read`, `_lseek`, `_close`, `_write`, and `_sbrk`.

The newlib documentation[5] calls these functions "system calls". In short, they are the handful of things newlib expects the underlying "operating system". The complete list of them is provided below:

```
_exit, close, environ, execve, fork, fstat, getpid, isatty, kill,
link, lseek, open, read, sbrk, stat, times, unlink, wait, write
```

You'll notice that several of the syscalls relate to filesystem operation or process control. These do not make much sense in a firmware context, so we'll often simply provide a stub that returns an error code.

Let's look at the ones our "Hello, World" example requires.

fstat

`fstat` returns the status of an open file. The minimal version of this should identify all files as character special devices. This forces one-byte-read at a time.

```
#include <sys/stat.h>
int fstat(int file, struct stat *st) {
  st->st_mode = S_IFCHR;
  return 0;
}
```

lseek

`lseek` repositions the file offset of the open file associated with the file descriptor `fd` to the argument `offset` according to the directive `whence`.

Here we can simply return 0, which implies the file is empty.


```
int lseek(int file, int offset, int whence) {
  return 0;
}
```

close

`close` closes a file descriptor `fd`.

Since no file should have gotten `open`-ed, we can just return an error on close:

```
int close(int fd) {
  return -1;
}
```

write

This is where things get interesting! `write` writes up to `count` bytes from the buffer starting at `buf` to the file referred to by the file descriptor `fd`.

Functions like `printf` rely on `write` to write bytes to `STDOUT`. In our case, we will want those bytes to be written to serial instead.

On the SAMD21 chip we are using, writing bytes to serial is done using the `usart_serial_putchar` function. We can use it to implement `write`:

```
static struct usart_module stdio_uart_module;

int _write (int fd, char *buf, int count) {
  int written = 0;

  for (; count != 0; --count) {
    if (usart_serial_putchar(&stdio_uart_module, (uint8_t)*buf++)) {
      return -1;
    }
    ++written;
  }
```

```
  return written;
}
```

We'll also need to initialize the USART peripheral prior to calling printf:


```
static void serial_init(void) {
  struct usart_config usart_conf;

  usart_get_config_defaults(&usart_conf);
  usart_conf.mux_setting = USART_RX_3_TX_2_XCK_3;
  usart_conf.pinmux_pad0 = PINMUX_UNUSED;
  usart_conf.pinmux_pad1 = PINMUX_UNUSED;
  usart_conf.pinmux_pad2 = PINMUX_PB22D_SERCOM5_PAD2;
  usart_conf.pinmux_pad3 = PINMUX_PB23D_SERCOM5_PAD3;

  usart_serial_init(&stdio_uart_module, SERCOM5, &usart_conf);
  usart_enable(&stdio_uart_module);
}

int main() {
  serial_init();

  printf("Hello, World!\n");
  while (1) {}
}
```

read

`read` attempts to read up to `count` bytes from file descriptor `fd` into the buffer at `buf`.

Similarly to `write`, we want `read` to read bytes from serial:

```
int _read (int fd, char *buf, int count) {
  int read = 0;

  for (; count > 0; --count) {
    usart_serial_getchar(&stdio_uart_module, (uint8_t *)buf++);
    read++;
  }

  return read;
}
```

sbrk

`sbrk` increases the program's data space by `increment` bytes. In other words, it increases the size of the heap.

What does `printf` have to do with the heap, you will justly ask? It turns out that newlib's `printf` implementations allocates data on the heap and depends on a working `malloc` implementation.

The source for `printf` is hard to follow, but you will find that indeed it calls malloc!

Here's a simple implementation of `sbrk`:

```
void *_sbrk(int incr) {
  static unsigned char *heap = HEAP_START;
  unsigned char *prev_heap = heap;
  heap += incr;
  return prev_heap;
}
```

More often than not, we want the heap to use all the RAM not used by anything else. We therefore set `HEAP_START` to the first address not spoken for in our linker script. In our previous post, we had added the `_end` variable in our linker script to that end.

We replace `HEAP_START` with `_end` and get:

```
void *_sbrk(int incr) {
  static unsigned char *heap = NULL;
  unsigned char *prev_heap;

  if (heap == NULL) {
    heap = (unsigned char *)&_end;
  }
  prev_heap = heap;

  heap += incr;

  return prev_heap;
}
```

Putting it all together, we get the following `main.c` file:

```
static struct usart_module stdio_uart_module;
```

```
// LIBC SYSCALLS
////////////////////

extern int _end;

void *_sbrk(int incr) {
  static unsigned char *heap = NULL;
  unsigned char *prev_heap;

  if (heap == NULL) {
    heap = (unsigned char *)&_end;
  }
  prev_heap = heap;

  heap += incr;

  return prev_heap;
}

int _close(int file) {
  return -1;
}

int _fstat(int file, struct stat *st) {
  st->st_mode = S_IFCHR;

  return 0;
}

int _isatty(int file) {
  return 1;
}

int _lseek(int file, int ptr, int dir) {
  return 0;
}

void _exit(int status) {
  __asm("BKPT #0");
}

void _kill(int pid, int sig) {
  return;
}

int _getpid(void) {
  return -1;
}

int _write (int file, char * ptr, int len) {
  int written = 0;

  if ((file != 1) && (file != 2) && (file != 3)) {
    return -1;
  }
```

14:47

```
  for (; len != 0; --len) {
    if (usart_serial_putchar(&stdio_uart_module, (uint8_t)*ptr++)) {
      return -1;
    }
    ++written;
  }
  return written;
}

int _read (int file, char * ptr, int len) {
  int read = 0;

  if (file != 0) {
    return -1;
  }

  for (; len > 0; --len) {
    usart_serial_getchar(&stdio_uart_module, (uint8_t *)ptr++);
    read++;
  }
  return read;
}


// APP
//////////////////////

static void __attribute__((constructor)) serial_init(void) {
  struct usart_config usart_conf;

  usart_get_config_defaults(&usart_conf);
  usart_conf.mux_setting = USART_RX_3_TX_2_XCK_3;
  usart_conf.pinmux_pad0 = PINMUX_UNUSED;
  usart_conf.pinmux_pad1 = PINMUX_UNUSED;
  usart_conf.pinmux_pad2 = PINMUX_PB22D_SERCOM5_PAD2;
  usart_conf.pinmux_pad3 = PINMUX_PB23D_SERCOM5_PAD3;

  usart_serial_init(&stdio_uart_module, SERCOM5, &usart_conf);
  usart_enable(&stdio_uart_module);
}

int main() {
  serial_init();
  printf("Hello, World!\n");
}
```

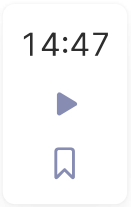

This compiles fine, and can be run on our MCU. Hello, World!

Initializing State with Constructors & Destructors

Although we could perfectly well stop here, we can improve a bit over the above.

In our example, `printf` depends implicitly on `serial_init` being called. This isn't the end of the world, but it goes against the spirit of a standard library function which should be usable anywhere in our program.

Instead, let's see what we can do so that this works:




```
int main() {
  printf("Hello, World\n");
}
```

Can you think of a solution?

If we want `printf` to work anywhere in our `main` function, then `serial_init` must be run **before** main. What runs before main? We know from our previous post that it is the `Reset_Handler`. A simple solution might therefore be:

```
void Reset_Handler(void)
{
  /* ... */
  /* Hardware Initialization */
  serial_init();

  /* Branch to main function */
  main();

  /* Infinite loop */
  while (1);
}
```

The GNU compiler collection and Newlib offer an alternative solution: constructors.

Constructors are functions which should be run before `main`. Conceptually, they are similar to the constructors of statically allocated C++ objects.

A function is marked as a constructor using the attribute syntax: `__attribute__((constructor))`. We can thus update `serial_init`:

```
static void __attribute__((constructor)) serial_init(void) {
  struct usart_config usart_conf;

  usart_get_config_defaults(&usart_conf);
  usart_conf.mux_setting = USART_RX_3_TX_2_XCK_3;
```

```
    usart_conf.pinmux_pad0 = PINMUX_UNUSED;
    usart_conf.pinmux_pad1 = PINMUX_UNUSED;
    usart_conf.pinmux_pad2 = PINMUX_PB22D_SERCOM5_PAD2;
    usart_conf.pinmux_pad3 = PINMUX_PB23D_SERCOM5_PAD3;

    usart_serial_init(&stdio_uart_module, SERCOM5, &usart_conf);
    usart_enable(&stdio_uart_module);
}
```


But how do these constructors get invoked? We know that in firmware, we do not get anything for free. This is where newlib comes in.

By default, GCC will put every constructor into an array in their own section of flash. Newlib then offers a function, `__libc_init_array` which iterates over the array and invokes every constructor. You can find out more about it by reading the source code.

All we need to do is call `__libc_init_array` prior to `main` in our `Reset_Handler`, and we are good to go.

```
void Reset_Handler(void)
{
  /* ... */
  /* Run constructors / initializers */
  __libc_init_array();

  /* Branch to main function */
  main();

  /* Infinite loop */
  while (1);
}
```

Newlib and Multi-threading

We have not yet talked much about multi-threading (e.g. with an RTOS) in this series, and going into details is outside of the scope of this article. However, there are a few things worth knowing when using Newlib in a multi-threaded environment.

`_impure_ptr` and the `_reent` struct

Most Newlib functions are *reentrant*. This means that they can be called by multiple processes safely.

For the functions that cannot be easily made re-entrant, newlib depends on the operating system correctly setting the `_impure_ptr` variable whenever a context switch occur. That variable is expected to hold a `struct _reent` for the current thread. That struct is used to store state for standard library functions being used by that thread.

Locking shared memory

Some standard library functions depend on global memory which would not make sense to hold in the `_reent` struct. This is especially important when using `malloc` to allocate memory out of the heap. If mutliple threads try modifying the heap at the same time, they risk corrupting it.

To allow multiple threads to call `malloc`, Newlib provides the `__malloc_lock` and `__malloc_unlock` APIs[6]. A good implementation of these APIs would lock and unlock a recursive mutex.

Implementing our own C standard library

In some cases, you may want to take different tradeoffs than the ones taken by the implementers of Newlib. Perhaps you are willing to sacrifice some functionality for code space, or are willing to trade performance for security. In most cases it is easier to replace a few functions, though you may end up with a fully custom C library.

Replacing a function

Because Newlib is a static library with a separate object file for every function, all you need to do to replace a function is define it in your program. The linker won't go looking for it in static libraries if it finds it in your code.

For example, we may want to replace Newlib's `printf` implementation, either because it is too large or because it depends on dynamic memory management. Using Marco Paland's excellent alternative[7] is as simple as a Makefile change.

We first clone it in our firmware's folder under `lib/printf`, and update our Makefile to reflect the change:

```
PROJECT := with-libc
BUILD_DIR ?= build
```

```
INCLUDES = \
  ... \
  lib/printf

SRCS = \
  ... \
  lib/printf/printf.c \
        startup_samd21.c \
        $(PROJECT).c

include ../common-standalone.mk
```


Full replacement

In some cases, you may want to do away with Newlib altogether. Perhaps you don't want any dynamic memory allocation, in which case you could use Embedded Artistry's solid alternative[8]. Another good reason to replace the version of Newlib provided by your toolchain is to use your own build of it because you would like to use different compile-time flags.

Once we have copied the static lib (`.a`) for our selected libc, we disable Newlib with `-nostdlib` and explicitly link in our substitute library. You can find the resulting Makefile below:

```
PROJECT := with-libc
BUILD_DIR ?= build

CFLAGS += -nostdlib
LDFLAGS += -L../lib/embeddedartistry_libc -lc

INCLUDES = \
        $(ASF_PATH)/sam0/drivers/sercom \
        $(ASF_PATH)/sam0/drivers/sercom/usart \
        $(ASF_PATH)/common/services/serial \
        $(ASF_PATH)/common/services/serial/sam0_usart

SRCS = \
        $(ASF_PATH)/sam0/drivers/sercom/usart/usart.c \
        $(ASF_PATH)/sam0/drivers/sercom/sercom.c \
        startup_samd21.c \
        $(PROJECT).c

include ../common-standalone.mk
```

Note that the `__libc_init_array` functionality is not found in every standard C library. You will either need to avoid using it, or bring in Newlib's implementation.

Closing

We hope reading this post has given you an understanding of how standard C functions come to be available in firmware code. As with previous posts, code examples are available in the Zero to Main Github repository.

Got a favorite libc implementation? Tell us all about it in the comments, or at interrupt@memfault.com.

Next time in the series, we'll talk about Rust!

EDIT: Post written! - From Zero to main(): Bare metal Rust

Reference Links

1. musl libc ↩
2. bionic libc ↩
3. ucLibc ↩
4. dietlibc ↩
5. newlib documentation ↩
6. __malloc_lock documentation ↩
7. Marco Paland's printf ↩
8. Embedded Artistry's libc ↩

François Baldassari *has worked on the embedded software teams at Sun, Pebble, and Oculus. He is currently the CEO of* ***Memfault****.*

Start Discussion 0 re

14:47

RSS Slack Subscribe Contribute Community Memfault.com