嵌入式宝典

嵌入式宝典带领你经历从零创造一个 `#![no_std]` 应用的过程，经历为Coterx-M微控制器搭建架构特定的功能的迭代过程。

目的

通过阅读这本书你将会学到

- 搭建一个 `#![no_std]` 应用。这比搭建一个 `#![no_std]` 库更复杂，因为目标系统可能没有运行一个OS(或者你的目标就是搭建一个OS!)，而且你的程序可能是目标中运行的唯一进程(或者第一个进程)。在这种情况下，程序可能需要为目标系统进行定制。
- 精细控制一个Rust程序的存储布局的技巧。你将学到链接器(linkers)，链接器脚本和Rust中那些能让你控制Rust程序某些ABI的功能。
- 实现可以被静态重载(没有运行时消耗)的默认功能的一个技巧。

目标读者

这本书主要面向两个读者:

- 希望为一个生态系统还没有支持的架构提供裸板支持(比如，自Rust 1.28以来的Cortex-R)，或者为一个刚获得Rust支持的架构提供支持(比如 未来可能有Xtensa)
- 对像是 `cortex-m-rt`， `msp430-rt` 和 `riscv-rt` 这样的 *runtime* 库的不寻常的实现感到好奇的人。

翻译

这本书已经被慷慨的志愿者翻译了。如果你想要你的翻译被列在这里，请打开一个PR添加它。

- 日文 (repository)
- 中文 (repository)

要求

这本书是自洽的。读者不需要熟悉Cortex-M架构，也不需要一个Cortex-M微控制器 -- 这本书里包含 的所有例子能在QEMU中测试。然而，你需要安装下面的工具来运行和检查这本书中的示例:

- 这本书中所有代码使用的是2018版的Rust。如果你不熟悉2018的特性和术语，阅读 `edition guide` 。
- Rust 1.31 或者更新的具有ARM Cortex-M编译支持的工具链。
- `cargo-binutils`. v0.1.4 或者更新的版本。
- `cargo-edit`.
- 有ARM仿真支持的QEMU。 `qemu-system-arm` 程序必须被安装在你的电脑上。
- 有ARM支持的GDB 。

安装示例

所有操作系统通用的指令

```
$ # Rust 工具链
$ # 如果你是从零开始，从 https://rustup.rs/ 获取rustup
$ rustup default stable

$ # 工具链应该比这个更新
$ rustc -V
rustc 1.31.0 (abe02cefd 2018-12-04)

$ rustup target add thumbv7m-none-eabi

$ # cargo-binutils
$ cargo install cargo-binutils

$ rustup component add llvm-tools-preview
```

macOS

```
$ # arm-none-eabi-gdb
$ # 你可能需要先运行 `brew tap Caskroom/tap`
$ brew install --cask gcc-arm-embedded

$ # QEMU
$ brew install qemu
```

Ubuntu 16.04

```
$ # arm-none-eabi-gdb
$ sudo apt install gdb-arm-none-eabi

$ # QEMU
$ sudo apt install qemu-system-arm
```

Ubuntu 18.04 或者 Debian

```
$ # gdb-multiarch -- 当你希望启动gdb时，使用 `gdb-multiarch`
$ sudo apt install gdb-multiarch

$ # QEMU
$ sudo apt install qemu-system-arm
```

Windows

- arm-none-eabi-gdb. The GNU Arm Embedded Toolchain 包括 GDB 。
- QEMU

从ARM安装一个工具链(可选的步骤) (在Ubuntu 18.04上测试过)

- 2018年之后，对于Cortex-M微控制器GCC的链接器换成了LLD，gcc-arm-none-eabi 不再需要了。但是对于那些想使用这个工具链的人，可以从这里安装并按照下面的步骤设置:

```
$ tar xvjf gcc-arm-none-eabi-8-2018-q4-major-linux.tar.bz2
$ mv gcc-arm-none-eabi-<version_downloaded> <your_desired_path> # 可选
$ export PATH=${PATH}:<path_to_arm_none_eabi_folder>/bin # 把这行添加到 .bashrc
使它永久有效
```

最小的 `#![no_std]` 程序

在这部分，我们将写一个可以*编译*的最小的 `#![no_std]` 程序。

`#![no_std]` 是什么意思?

`#![no_std]` 是一个crate层的属性，其指出crate将链接到 `core` 而不是 `std` crate，但是这对应用来说意味着什么呢?

`std` crate 是Rust的标准库。它包含的功能需要假设程序将运行在一个操作系统上而不是[*直接运行在裸机*]上。 `std` 也假设操作系统是一个通用目的操作系统，像是会在服务器和桌面看到的那些系统。因此， `std` 在通常会在操作系统中的找到的那些功能:线程，文件，套接字，一个文件系统，进程，等等之上，提供一个标准的API 。

换句话说， `core` crate是 `std` crate的一个子集，其不对将运行程序的系统做任何假设。它提供与语言的基本类型，像是浮点数，字符串和切片有关的APIs，也提供像是原子操作和SIMD指令这样暴露处理器特性的APIs 。然而它缺少涉及到堆内存分配和I/O有关的APIs。

对于一个应用来说， `std` 不仅仅只是提供一种方法访问OS抽象。 `std` 在某些情况下，也提供栈溢出保护，处理命令行参数，在一个程序的 `main` 函数被启动前打开主线程。一个 `#![no_std]` 应用缺少上述的所有运行时，因此它必须初始化它自己的运行时，如果有需要的话。

由于这些特点，一个 `#![no_std]` 应用可以成为第一个或者是唯一一个运行在一个系统上的代码。它可以成为许多一个标准Rust应用无法成为的东西，比如:

- 一个操作系统的内核。
- 固件。
- 一个启动引导。

代码

讲完了，我们可以转向最小的 `#![no_std]` 程序了:

```
$ cargo new --edition 2018 --bin app

$ cd app
```

```
$ # 把 main.rs 改成这些内容
$ cat src/main.rs
```

```
#![no_main]
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
    loop {}
}
```

这个程序包含在标准的Rust程序中你将不会看到的一些东西:

`#![no_std]` 属性，我们已经讲过了。

`#![no_main]` 属性，它意味这程序将不会使用标准的 `main` 函数作为入口。在写这本书的时候，Rust的 `main` 接口对程序执行的环境做了一些假设: 比如，它假设存在命令行参数，因此，通常它不适合 `#![no_std]` 程序。

`#[panic_handler]` 属性。用这个属性标记的函数定义了恐慌的行为，包括库层级的恐慌(`core::painc!`)和语言层级的恐慌(越界索引)。

这个程序不产生任何有用的东西。事实上，它将产生一个空的二进制项。

```
$ # 等于 `size target/thumbv7m-none-eabi/debug/app`
$ cargo size --target thumbv7m-none-eabi --bin app
```

```
   text    data     bss     dec     hex filename
      0       0       0       0       0 app
```

在链接之前，crate 包含恐慌函数的符号。

```
$ cargo rustc --target thumbv7m-none-eabi -- --emit=obj

$ cargo nm -- target/thumbv7m-none-eabi/debug/deps/app-*.o | grep '[0-9]* [^N]
'
```

```
00000000 T rust_begin_unwind
```

然而，它是我们的起始点。在下一个部分，我们将搭建一些有用的东西。但是继续之前，让我们设置一个默认的编译目标避免每次调用Cargo不得不传递 `--target` 标志。

```
$ mkdir .cargo

$ # 把 .cargo/config 改成这些内容
$ cat .cargo/config
```

```
[build]
target = "thumbv7m-none-eabi"
```

eh_personality

如果你的配置在恐慌时不会无条件终止，大多数与完整的操作系统有关的目标都不会(或者如果你的 自制目标 不包含 `"panic-strategy": "abort"`)，那么你必须告诉Cargo这么做或者添加一个 `eh_personality` 函数，其需要一个nightly版的编译器。这里是关于它的Rust文档， 这里是一些关于它的讨论.

在你的 Cargo.toml 中， 添加:

```
[profile.dev]
panic = "abort"

[profile.release]
panic = "abort"
```

另外，声明 `eh_personality` 函数。一个简单的不包含任何特别的东西的实现，展开时像下面一样:

```
#![feature(lang_items)]

#[lang = "eh_personality"]
extern "C" fn eh_personality() {}
```

如果没有，你将会收到这个错误 `language item required, but not found: 'eh_personality'` 。

存储布局

下一步是确保程序有正确的存储布局，因此目标系统才可以执行它。在我们的例子里，我将使用一个虚拟的Cortex-M3微控制器: LM3S6965 。我们的程序是运行在设备上的唯一一个进程，因此它也必须要负责初始化设备。

背景信息

Cortex-M 设备需要向量表放在它们代码区的开始处。 向量表是一组指针；启动设备需要前两个指针，剩下来的指针都与异常有关。我们现在忽略它们。

链接器决定程序的最终存储布局，但是我们可以用链接器脚本对它进行一些控制。链接器提供的控制布局的精细度在*sections*层级。一个section是分布在连续存储中的*符号*的集合。符号，依次，可以是数据(一个静态变量)，或者指令(一个Rust函数)。

每一个符号有一个由编译器分配的名字。自Rust 1.28以来，Rust编译器分配给符号的名字都是这样的格式: `_ZN5krate6module8function17he1dfc17c86fe16daE`，其展开是 `krate::module::function::he1dfc17c86fe16da` 其中 `krate::module::function` 是函数或者变量的路径， `he1dfc17c86fe16da` 某个哈希值。Rust编译器将把每个符号放进它自己的独有的section中；比如之前提到的符号将被放进一个名为 `.text._ZN5krate6module8function17he1dfc17c86fe16daE` 的section中。

这些编译器产生的符号和section名在不同的Rust编译器发布版中不保证是不变的。然而，语言让我们可以使用这些属性控制符号名和放置的section:

- `#[export_name = "foo"]` 把符号名设置成 `foo` 。
- `#[no_mangle]` 意思是: 使用函数或者变量名(不是它的全路径)作为它的符号名。 `#[no_mangle] fn bar()` 将产生一个名为 `bar` 的符号。
- `#[link_section = ".bar"]` 把符号放进一个名为 `.bar` 的section中。

有了这些属性，我们可以暴露一个程序的稳定的ABI，并在链接器脚本中使用它。

Rust 部分

像上面提到的，对于Cortex-M设备，我们需要修改向量表的前两项。第一个，栈指针的初始值，只能使用链接器脚本修改。第二个，重置向量，需要在Rust代码中生成并使用链接器脚本放置到正确的地方。

重置向量是一个指向重置处理函数的指针。重置处理函数是在一个系统重启后设备将会执行的函数，或者第一次上电后。重置处理函数总是硬件调用栈中的第一个栈帧；从它返回是未定义的行为，因为没有其它栈帧可以给它返回。通过让它变成一个divergent function，我们可以强调这个重置处理函数从来不会返回，divergent function的签名是 `fn(/* .. */) -> !` 。

```
#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    let _x = 42;

    // can't return so we go into an infinite loop here
    loop {}
}

// The reset vector, a pointer into the reset handler
#[link_section = ".vector_table.reset_vector"]
#[no_mangle]
pub static RESET_VECTOR: unsafe extern "C" fn() -> ! = Reset;
```

我们通过使用 `extern "C"` 告诉编译器下面的函数使用C的ABI，而不是Rust的ABI，后者不稳定，将函数变成这里硬件期望的格式。

为了从链接器脚本中指出重置处理函数和重置向量，我们需要它们有一个稳定的符号名因此我们使用 `#[no_mangle]` 。我们需要稍微控制下 `RESET_VECTOR` 的位置，因此我们将它放进一个已知的section中， `.vector_table.reset_vector` 。重置处理函数， `Reset` ，的确切位置不重要。我们只使用编译器默认生成的section 。

这个链接器将忽略有内部链接的符号(也叫内部符号)，遍历输入目标文件的列表，因此我们需要我们的符号具有外部链接。在Rust中让一个符号编程外部的方法只有使它的相关项变成公共的(`pub`)和*可到达的*(在项和crate的根路径间没有私有模块)。

链接器脚本部分

一个最小的，将向量表放进正确的位置的链接器脚本如下所示。让我们仔细看看。

```
$ cat link.x
```

```
/* Memory layout of the LM3S6965 microcontroller */
/* 1K = 1 KiBi = 1024 bytes */
MEMORY
{
  FLASH : ORIGIN = 0x00000000, LENGTH = 256K
  RAM : ORIGIN = 0x20000000, LENGTH = 64K
}

/* The entry point is the reset handler */
ENTRY(Reset);

EXTERN(RESET_VECTOR);

SECTIONS
{
  .vector_table ORIGIN(FLASH) :
  {
    /* First entry: initial Stack Pointer value */
    LONG(ORIGIN(RAM) + LENGTH(RAM));

    /* Second entry: reset vector */
    KEEP(*(.vector_table.reset_vector));
  } > FLASH

  .text :
  {
    *(.text .text.*);
  } > FLASH

  /DISCARD/ :
  {
    *(.ARM.exidx .ARM.exidx.*);
  }
}
```

MEMORY

链接器脚本的这个部分描述了目标中存储区块的大小和位置。有两个存储区块被定义了: `FLASH` 和 `RAM` ；它们都与目标中可用的物理存储关联。这里用的值与LM3S6965微控制器有关。

ENTRY

这里我们指给链接器，符号名为 `Reset` 的重置处理函数是程序的 *entry point* 。链接器会主动丢弃未使用的部分。链接器认为entry point和从它处被调用的函数是*被使用了的*，所以链接器将不会丢弃它们。没有这一行，链接器将会丢弃 `Reset` 函数和所有接下来所有从它处调用的函数。

EXTERN

链接器是懒惰的；一旦它们找到了从entry point处递归引用的所有的符号，它们将停止查看输入目标文件。`EXTERN` 强迫链接器去寻找 `EXTERN` 的参数即使其它所有的被引用的符号都被找到后。因为thumb的一个规则，如果你需要一个符号，其没有被entry point调用，总是出现在输出的二进制项中，你应该使用结合了 `KEEP` 的 `EXTERN` 。

SECTIONS

这部分描述了输入目标文件中的sections(也被称为*input sections*)是如何被安排进输出目标文件的sections(也被称为output sections)中的或者它们是否应该被丢弃。这里，我们定义两个输出sections:

```
  .vector_table ORIGIN(FLASH) : { /* .. */ } > FLASH
```

`.vector_table` 包含向量表且坐落于 `FLASH` 存储的开始处。

```
  .text : { /* .. */ } > FLASH
```

`.text` 包含程序的子例程且坐落于 `FLASH` 的某些位置。它的开始地址没有指定，但是链接器将把它放在先前的输出section， `.vector_table` 之后。

输出 `.vector_table` section包含:

```
    /* First entry: initial Stack Pointer value */
    LONG(ORIGIN(RAM) + LENGTH(RAM));
```

我们将把(调用)栈放在RAM的末尾(栈是*完全递减的*；它向着更小的地址增长)因此RAM的末尾地址将被用作栈指针(SP)值。使用我们输入的`RAM`存储区块的信息，在链接器中那个地址可以被链接器算出来。

```
    /* Second entry: reset vector */
    KEEP(*(.vector_table.reset_vector));
```

接下来，我们使用 `KEEP` 去强迫链接器在初始的SP值之后插入所有的名为 `.vector_table.reset_vector` 的input sections 。位于那个section中的唯一一个符号是 `RESET_VECTOR`，所以这将可以把 `RESEET_VECTOR` 放在向量表的第二个位置。

输出的 `.text` section 包含:

```
    *(.text .text.*);
```

这包括所有的名为 `.text` 和 `.text.*` 的input sections 。注意我们这里没有使用 `KEEP` 去让链接器丢弃不使用的部分。

最后，我们使用特殊的 `/DISCARD/` section 去丢弃

```
    *(.ARM.exidx .ARM.exidx.*);
```

名为 `.ARM.exidx.*` 的input sections 。这些sections与异常处理有关，但是我们在恐慌时不进行栈展开，它们占用Flash的存储空间，因此我们就丢弃它们。

全放到一起去

现在我们能链接应用了。作为参考，这里是完整的Rust程序:

```
#![no_main]
#![no_std]

use core::panic::PanicInfo;

// The reset handler
#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    let _x = 42;

    // can't return so we go into an infinite loop here
    loop {}
}

// The reset vector, a pointer into the reset handler
#[link_section = ".vector_table.reset_vector"]
#[no_mangle]
pub static RESET_VECTOR: unsafe extern "C" fn() -> ! = Reset;

#[panic_handler]
fn panic(_panic: &PanicInfo<'_>) -> ! {
    loop {}
}
```

我们不得不修改链接的过程让它使用我们的链接器脚本。通过传递 `-C link-arg` 标志给 `rustc` 来完成它。使用 `cargo-rustc` 或者 `cargo-build` 就可以完成了。

重要: 在运行这个命令之前确保你有 `.cargo/config` 文件，其在上个章节末尾处被添加。

使用 `cargo-rustc` 子命令:

```
$ cargo rustc -- -C link-arg=-Tlink.x
```

或者你可以在 `.cargo/config` 中设置rustflags，其继续使用 `cargo-build` 子命令。我们将会使用后者因为它与 `cargo-binutils` 集成的更好。

```
# 将 .cargo/config 修改成这些内容
$ cat .cargo/config
```

```
[target.thumbv7m-none-eabi]
rustflags = ["-C", "link-arg=-Tlink.x"]

[build]
target = "thumbv7m-none-eabi"
```

`[target.thumbv7m-none-eabi]` 部分告知这些标志只会被用于当交叉编译的目标是 `thumbv7m-none-eabi` 时。

检查它

现在让我们检查下输出的二进制项，确保存储布局跟我们想要的一样 (这需要 `cargo-binutils`):

```
$ cargo objdump --bin app -- -d --no-show-raw-insn
```

```
app:    file format elf32-littlearm

Disassembly of section .text:

<Reset>:
                sub     sp, #4
                movs    r0, #42
                str     r0, [sp]
                b       0x10 <Reset+0x8>        @ imm = #-2
                b       0x10 <Reset+0x8>        @ imm = #-4
```

这是 `.text` section的反汇编。我们看到重置处理函数，名为 `Reset`，位于 `0x8` 地址。

```
$ cargo objdump --bin app -- -s --section .vector_table
```

```
app:    file format elf32-littlearm
Contents of section .vector_table:
 0000 00000120 09000000                    ... ....
```

这是 `.vector_table` section 的内容。我们可以看到section开始于地址 `0x0` 且 section 的第一个字是 `0x2001_0000` (`objdump` 的输出是小端模式)。这是初始的SP值，它与RAM的末尾地址匹配。第二个字是 `0x9`；这是重置处理函数的 *thumb mode* 地址。当一个函数运行在thumb mode下，它的地址的第一位被设置成1 。

测试它

这个程序是一个有效的LM3S6965程序；我们可以在一个虚拟微控制器(QEMU)中执行它去测试。

```
$ # 这个程序将会阻塞住
$ qemu-system-arm \
      -cpu cortex-m3 \
      -machine lm3s6965evb \
      -gdb tcp::3333 \
      -S \
      -nographic \
      -kernel target/thumbv7m-none-eabi/debug/app
```

```
$ # 在一个不同的终端上
$ arm-none-eabi-gdb -q target/thumbv7m-none-eabi/debug/app
Reading symbols from target/thumbv7m-none-eabi/debug/app...done.

(gdb) target remote :3333
Remote debugging using :3333
Reset () at src/main.rs:8
8       pub unsafe extern "C" fn Reset() -> ! {

(gdb) # the SP has the initial value we programmed in the vector table
(gdb) print/x $sp
$1 = 0x20010000

(gdb) step
9           let _x = 42;

(gdb) step
12          loop {}

(gdb) # next we inspect the stack variable `_x`
(gdb) print _x
$2 = 42

(gdb) print &_x
$3 = (i32 *) 0x2000fffc

(gdb) quit
```

一个 `main` 接口

我们现在有了一个可以工作的最小的程序，但是我们需要用一个方法将它打包起来，这个方法可以让终端用户在其上搭建安全的程序。这部分，我们将实现一个 `main` 接口，就像一个标准的Rust程序所用的那样。

首先，我们将把我们的binary crate转换成一个library crate:

```
$ mv src/main.rs src/lib.rs
```

然后把它重命名为 `rt`，其表示"runtime" 。

```
$ sed -i s/app/rt/ Cargo.toml

$ head -n4 Cargo.toml
```

```
[package]
edition = "2018"
name = "rt" # <-
version = "0.1.0"
```

第一个改变是让重置处理函数调用一个外部的 `main` 函数:

```
$ head -n13 src/lib.rs
```

```
#![no_std]

use core::panic::PanicInfo;

// CHANGED!
#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    extern "Rust" {
        fn main() -> !;
    }

    main()
}
```

我们也去掉了 `#![no_main]` 属性，因为它对library crates没有影响。

> 这个阶段存在一个正交问题: `rt` 库应该提供一个标准的恐慌时行为吗， 或者它不应该提供一个 `#[panic_handler]` 函数而让终端用户去选 择恐慌时行为吗？这个文档将不会深入这个问题，且为了简洁性，在 `rt` crate 中留了一个空的 `#[panic_handler]` 函数。然而，我们想告诉读者存在其它选择。

第二个改变涉及到给应用crate提供我们之前写的链接器脚本。链接器将会在库搜索路径(`-L`)和它被调用的文件夹中寻找链接器脚本。应用crate不应该需要将`link.x`的副本挪来挪去所以我们将使用一个build script让`rt` crate 将链接器脚本放到库搜索路径中。

```
$ # 在`rt`的根目录中生成一个带有这些内容的 build.rs 文件
$ cat build.rs
```

```
use std::{env, error::Error, fs::File, io::Write, path::PathBuf};

fn main() -> Result<(), Box<dyn Error>> {
    // build directory for this crate
    let out_dir = PathBuf::from(env::var_os("OUT_DIR").unwrap());

    // extend the library search path
    println!("cargo:rustc-link-search={}", out_dir.display());

    // put `link.x` in the build directory
    File::create(out_dir.join("link.x"))?.write_all(include_bytes!("link.x"))?;

    Ok(())
}
```

现在用户可以写一个暴露了`main`符号的应用了，且将它链接到`rt` crate上。`rt` 将负责给予程序正确的存储布局。

```
$ cd ..

$ cargo new --edition 2018 --bin app

$ cd app

$ # 修改Cargo.toml将`rt` crate包含进来作为一个依赖
$ tail -n2 Cargo.toml
```

```
[dependencies]
rt = { path = "../rt" }
```

```
$ # 拷贝整个设置了一个默认目标的config文件并修改链接器命令
$ cp -r ../rt/.cargo .

$ # 把 `main.rs` 的内容改成
$ cat src/main.rs
```

```
#![no_std]
#![no_main]

extern crate rt;

#[no_mangle]
pub fn main() -> ! {
    let _x = 42;

    loop {}
}
```

反汇编将是相似的，除了现在包含了用户的`main`函数。

```
$ cargo objdump --bin app -- -d --no-show-raw-insn
```

```

app:    file format elf32-littlearm

Disassembly of section .text:

<main>:
                sub     sp, #4
                movs    r0, #42
                str     r0, [sp]
                b       0x10 <main+0x8>         @ imm = #-2
                b       0x10 <main+0x8>         @ imm = #-4

<Reset>:
                push    {r7, lr}
                mov     r7, sp
                bl      0x8 <main>              @ imm = #-18
                trap
```

把它变成类型安全的

`main`接口工作了，但是它容易出错。比如，用户可以把`main`写成一个non-divergent function，它们将不会出现编译时错误并带来未定义的行为(编译器将会错误优化这个程序)。

我们通过暴露一个宏给用户而不是符号接口可以添加类型安全性。在`rt` crate 中，我们可以写这个宏:

```
$ tail -n12 ../rt/src/lib.rs
```

```
#[macro_export]
macro_rules! entry {
    ($path:path) => {
        #[export_name = "main"]
        pub unsafe fn __main() -> ! {
            // type check the given path
            let f: fn() -> ! = $path;

            f()
        }
    }
}
```

然后应用的作者可以像这样调用它:

```
$ cat src/main.rs
```

```
#![no_std]
#![no_main]

use rt::entry;

entry!(main);

fn main() -> ! {
    let _x = 42;

    loop {}
}
```

如果作者把`main`的签名改成non divergent function，比如`fn`，将会出现一个错误。

main之前的生活

`rt`看起来不错了，但是它的功能不够完整！用它编写的应用不能使用`static`变量或者字符串字面值，因为`rt`的链接器脚本没有定义标准的`.bss`，`.data`和`.rodata` sections。让我们修复它!

第一步是在链接器脚本中定义这些sections:

```
$ # 只展示文件的一小块
$ sed -n 25,46p ../rt/link.x
```

```
  .text :
  {
    *(.text .text.*);
  } > FLASH

  /* NEW! */
  .rodata :
  {
    *(.rodata .rodata.*);
  } > FLASH

  .bss :
  {
    *(.bss .bss.*);
  } > RAM

  .data :
  {
    *(.data .data.*);
  } > RAM

  /DISCARD/ :
```

它们只是重新导出input sections并指定每个output section将会进入哪个内存区域。

有了这些改变，下面的程序将会编译:

```
#![no_std]
#![no_main]

use rt::entry;

entry!(main);

static RODATA: &[u8] = b"Hello, world!";
static mut BSS: u8 = 0;
static mut DATA: u16 = 1;

fn main() -> ! {
    let _x = RODATA;
    let _y = unsafe { &BSS };
    let _z = unsafe { &DATA };

    loop {}
}
```

然而如果你在正真的硬件上运行这个程序并调试它，你将发现到达 `main` 时， `static` 变量 `BSS` 和 `DATA` 没有 `0` 和 `1` 值。反而，这些变量将是垃圾值。问题是在设备上电之后，RAM的内容是随机的。如果你在QEMU中运行这个程序，你将看不到这个影响。

在目前的情况下，如果你的程序在对 `static` 变量执行一个写入之前，读取任何 `static` 变量，那么你的程序会出现未定义的行为。让我们通过在调用 `main` 之前初始化所有的 `static` 变量来修复它。

我们需要修改下链接器脚本去进行RAM初始化:

```
$ # 只展示文件的一块
$ sed -n 25,52p ../rt/link.x
```

```
  .text :
  {
    *(.text .text.*);
  } > FLASH

  /* CHANGED! */
  .rodata :
  {
    *(.rodata .rodata.*);
  } > FLASH

  .bss :
  {
    _sbss = .;
    *(.bss .bss.*);
    _ebss = .;
  } > RAM

  .data : AT(ADDR(.rodata) + SIZEOF(.rodata))
  {
    _sdata = .;
    *(.data .data.*);
    _edata = .;
  } > RAM

  _sidata = LOADADDR(.data);

  /DISCARD/ :
```

让我们深入下细节:

```
    _sbss = .;
```

```
    _ebss = .;
```

```
    _sdata = .;
```

```
    _edata = .;
```

我们将符号关联到`.bss`和`.data` sections的开始和末尾地址，我之后将会从Rust代码中使用。

```
  .data : AT(ADDR(.rodata) + SIZEOF(.rodata))
```

我们将`.data` section的加载内存地址(LMA)设置成`.rodata` section的末尾处。`.data`包含具有一个非零的初始值的`static`变量；`.data` section的虚拟内存地址(VMA)在RAM中的某处 -- 这是`static`变量所在的地方。这些`static`变量的初始值，然而，必须被分配在非易失存储中(Flash)；LMA是Flash中存放这些初始值的地方。

```
  _sidata = LOADADDR(.data);
```

最后，我们将一个符号和`.data`的LMA关联起来。我们可以从Rust代码引用我们在链接器脚本中生成的符号。这些符号的*地址*[1]是`.bss`和`.data` sections的边界处。

下面展示的是更新了的重置处理函数:

```
$ head -n32 ../rt/src/lib.rs
```

```
#![no_std]

use core::panic::PanicInfo;
use core::ptr;

#[no_mangle]
pub unsafe extern "C" fn Reset() -> ! {
    // NEW!
    // Initialize RAM
    extern "C" {
        static mut _sbss: u8;
        static mut _ebss: u8;

        static mut _sdata: u8;
        static mut _edata: u8;
        static _sidata: u8;
    }

    let count = &_ebss as *const u8 as usize - &_sbss as *const u8 as usize;
    ptr::write_bytes(&mut _sbss as *mut u8, 0, count);

    let count = &_edata as *const u8 as usize - &_sdata as *const u8 as usize;
    ptr::copy_nonoverlapping(&_sidata as *const u8, &mut _sdata as *mut u8,
count);

    // Call user entry point
    extern "Rust" {
        fn main() -> !;
    }

    main()
}
```

现在终端用户可以直接地和间接地使用`static`变量而不会导致未定义的行为!

我们上面展示的代码中，内存的初始化按照一种逐字节的方式进行。可以强迫 `.bss` 和 `.data` section对齐，比如，四个字节。然后Rust代码可以利用这个事实去执行逐字的初始化而不需要对齐检查。如果你想知道这是如何做到，看下 `cortex-m-rt` crate 。

---

[1] 必须在这使用链接器脚本符号这件事会让人疑惑且反直觉。可以在这里找到与这个怪现象有关的详尽解释。

异常处理

在"存储布局"部分，我们决定从简单处开始并忽略异常的处理。在这部分，我们将添加与处理它们有关的支持；这作为如何在稳定版的Rust中(比如 不依赖不稳定的 `#[linkage = "weak"]` 属性，它让一个符号变成弱链接)实现编译时可重载的行为的示例。

背景信息

简而言之，*异常*是Cortex-M和其它架构提供的一个机制，为了让应用可以响应异步，通常是外部的，事件。异常最常见的类型，大多数人将知道的，是经典的(硬件)中断。

Cortex-M异常机制工作起来像这样: 当处理器接收到与某个异常的类型相关的信号或者事件，它挂起现在的子例程的执行(通过将状态存进调用栈中)然后在一个新的栈帧中继续执行相关的异常处理函数，另一个子例程。在异常处理函数执行完成之后(比如 从它返回)，处理器恢复被挂起的子例程的执行。

处理器使用向量表去确定执行哪个处理函数。表中的每一项包含一个指向一个处理函数的指针，每一项与一个不同的异常类型关联起来。比如，第二项是重置处理函数，第三项是NMI(不可屏蔽中断)处理函数，等等。

像之前提到的，处理器希望向量表在存储中某个特定的位置，在运行时处理器可能用到表中的每一项。因此，表中的各项必须包含有效的值。此外，我们希望 `rt` crate是灵活的，因此终端用户可以定制每个异常处理函数的行为。最后，向量表要坐落在只读存储中，或者有些在不那么容易被修改的存储中，因此用户必须静态地注册处理函数，而不是在运行时。

为了满足所有的这些需求，我们将给 `rt` crate中的向量表所有的项分配一个*默认值*，但是让这些值变*弱*以让终端用户可以在编译时重载它们。

Rust部分

让我们看下所有这些如何被实现。为了间接性，我们将只使用向量表的前16个项；这些项不是特定于设备的，所以它们在所有类型的Cotex-M微控制器上都有相同的功能。

我们做的第一件事是在 `rt` create的代码中创造一个向量(指向异常处理函数的指针)数组:

```
$ sed -n 56,91p ../rt/src/lib.rs
```

```
pub union Vector {
    reserved: u32,
    handler: unsafe extern "C" fn(),
}

extern "C" {
    fn NMI();
    fn HardFault();
    fn MemManage();
    fn BusFault();
    fn UsageFault();
    fn SVCall();
    fn PendSV();
    fn SysTick();
}

#[link_section = ".vector_table.exceptions"]
#[no_mangle]
pub static EXCEPTIONS: [Vector; 14] = [
    Vector { handler: NMI },
    Vector { handler: HardFault },
    Vector { handler: MemManage },
    Vector { handler: BusFault },
    Vector {
        handler: UsageFault,
    },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: SVCall },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: PendSV },
    Vector { handler: SysTick },
];
```

向量表中的一些项是*保留的*；ARM文档说它们应该被分配 `0` 值，所以我们使用一个联合体来完成它。必须指向一个处理函数的项使用的是*external*函数；这很重要，因为它让终端用户来*提供*实际的函数定义。

接下来，我们在Rust代码中定义一个默认的异常处理函数。没有被终端用户分配的异常将使用这个默认处理函数。

```
$ tail -n4 ../rt/src/lib.rs
```

```
#[no_mangle]
pub extern "C" fn DefaultExceptionHandler() {
    loop {}
}
```

链接器脚本部分

在链接器脚本那部分，我们将这些新的异常向量放在重置向量之后。

```
$ sed -n 12,25p ../rt/link.x
```

```
EXTERN(RESET_VECTOR);
EXTERN(EXCEPTIONS); /* <- NEW */

SECTIONS
{
  .vector_table ORIGIN(FLASH) :
  {
    /* First entry: initial Stack Pointer value */
    LONG(ORIGIN(RAM) + LENGTH(RAM));

    /* Second entry: reset vector */
    KEEP(*(.vector_table.reset_vector));

    /* The next 14 entries are exception vectors */
    KEEP(*(.vector_table.exceptions)); /* <- NEW */
  } > FLASH
```

并且我们使用 `PROVIDE` 给我们在 `rt` 中未定义的处理函数赋予一个默认值 (`NMI` 和上面的其它处理函数):

```
$ tail -n8 ../rt/link.x
```

```
PROVIDE(NMI = DefaultExceptionHandler);
PROVIDE(HardFault = DefaultExceptionHandler);
PROVIDE(MemManage = DefaultExceptionHandler);
PROVIDE(BusFault = DefaultExceptionHandler);
PROVIDE(UsageFault = DefaultExceptionHandler);
PROVIDE(SVCall = DefaultExceptionHandler);
PROVIDE(PendSV = DefaultExceptionHandler);
PROVIDE(SysTick = DefaultExceptionHandler);
```

当检测完所有的输入目标文件而等号左侧的符号仍然没有定义的时候， `PROVIDE` 才会发挥作用。也就是用户没有为相关的异常实现处理函数。

测试它

这就完了！ `rt` crate现在支持异常处理函数了。我们可以用下列的应用测试它:

> **注意**: 在QEMU中生成一个异常很难。在实际的硬件上对一个无效的存储地址进行 一个读取是足够产生一个中断的，但是QEMU却能接受这个操作并且返回零。一个陷入指 令在QEMU

和硬件上都可以发挥作用，但是不幸的是在稳定版上不可以用它，所以你需要 暂时切换到nightly版本中去运行这个和下个示例。

```
#![feature(core_intrinsics)]
#![no_main]
#![no_std]

use core::intrinsics;

use rt::entry;

entry!(main);

fn main() -> ! {
    // this executes the undefined instruction (UDF) and causes a HardFault
exception
    intrinsics::abort()
}
```

```
(gdb) target remote :3333
Remote debugging using :3333
Reset () at ../rt/src/lib.rs:7
7       pub unsafe extern "C" fn Reset() -> ! {

(gdb) b DefaultExceptionHandler
Breakpoint 1 at 0xec: file ../rt/src/lib.rs, line 95.

(gdb) continue
Continuing.

Breakpoint 1, DefaultExceptionHandler ()
    at ../rt/src/lib.rs:95
95          loop {}

(gdb) list
90          Vector { handler: SysTick },
91      ];
92
93      #[no_mangle]
94      pub extern "C" fn DefaultExceptionHandler() {
95          loop {}
96      }
```

为了完整性，这里列出程序被优化过的版本的反汇编:

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
```

```
app:    file format elf32-littlearm

Disassembly of section .text:

<main>:
                trap
                trap

<Reset>:
                push    {r7, lr}
                mov     r7, sp
                movw    r1, #0x0
                movw    r0, #0x0
                movt    r1, #0x2000
                movt    r0, #0x2000
                subs    r1, r1, r0
                bl      0x9c <__aeabi_memclr>   @ imm = #0x3e
                movw    r1, #0x0
                movw    r0, #0x0
                movt    r1, #0x2000
                movt    r0, #0x2000
                subs    r2, r1, r0
                movw    r1, #0x282
                movt    r1, #0x0
                bl      0x84 <__aeabi_memcpy>   @ imm = #0x8
                bl      0x40 <main>             @ imm = #-0x40
                trap

<UsageFault>:
```

```
$ cargo objdump --bin app --release -- -s -j .vector_table
```

```
app:    file format elf32-littlearm
Contents of section .vector_table:
 0000 00000120 45000000 83000000 83000000  ... E...........
 0010 83000000 83000000 83000000 00000000  ................
 0020 00000000 00000000 00000000 83000000  ................
 0030 00000000 00000000 83000000 83000000  ................
```

向量表现在像是集合了这本书中迄今为止所有的代码片段。总结下:

- 在早期的存储章节的*检查它*部分，我们知道了:
  - 向量表中第一项包含了栈指针的初始值。
  - Objdump使用 小端 格式打印，所以栈开始于 `0x2001_0000` 。
  - 第二项指向地址 `0x0000_0045`，重置处理函数。
    - 在上面的反汇编中可以看到重置处理函数的地址，是 `0x44` 。
    - 由于对齐的要求被设置成1的第一位不会改变地址。而是，它让函数在 *thumb mode* 下执行。
- 之后，可以看到在 `0x83` 和 `0x00` 之间交替的地址模式。

- 看下上面的反汇编，很明显 `0x83` 指的是 `DefaultExceptionHandler`（`0x84`用thumb模式执行)。
- 来回查看在这个章节早期所设置的向量表和Cortex-M的向量表布局的模式，很明显每次有个处理函数项出现在表中， `DefaultExceptionHandler` 的地址就会出现。
- 可以看到Rust代码中的向量表的数据结构的布局与Cortex-M向量表中的保留项依次对齐了。因此，所有的保留项被正确的设置成了零值。

重载一个处理函数

为了重载一个异常处理函数，用户必须提供一个函数，其符号名完全匹配我们在 `EXCEPTIONS` 中使用的名字。

```
#![feature(core_intrinsics)]
#![no_main]
#![no_std]

use core::intrinsics;

use rt::entry;

entry!(main);

fn main() -> ! {
    intrinsics::abort()
}

#[no_mangle]
pub extern "C" fn HardFault() -> ! {
    // do something interesting here
    loop {}
}
```

你可以在QEMU中测试它

```
(gdb) target remote :3333
Remote debugging using :3333
Reset () at /home/japaric/rust/embedonomicon/ci/exceptions/rt/src/lib.rs:7
7       pub unsafe extern "C" fn Reset() -> ! {

(gdb) b HardFault
Breakpoint 1 at 0x44: file src/main.rs, line 18.

(gdb) continue
Continuing.

Breakpoint 1, HardFault () at src/main.rs:18
18          loop {}

(gdb) list
13      }
14
15      #[no_mangle]
16      pub extern "C" fn HardFault() -> ! {
17          // do something interesting here
18          loop {}
19      }
```

程序现在执行了用户定义的`HardFault`函数而不是`rt` crate中的`DefaultExceptionHandler`。

与我们在一个`main`接口中进行的第一次尝试一样，这个实现的问题是没有类型安全性。它也容易混淆异常的名字，但是不会生成一个错误或者警告。而是仅仅忽略用户定义的处理函数。这些问题可以使用一个像是在`cortex-m-rt` v0.5.x 中定义的`exception!`宏和`cortex-m-rt` v0.6.x中的`exception`属性来解决。

稳定版中的汇编

> 注意: 自Rust 1.59以来，*inline* 汇编(`asm!`)和 *free form* 汇编(`global_asm!`) 变得稳定了。但是因为现存的crates需要花点时间来消化这种变化，并且了解我们在历史中曾用过的处 理汇编的方法对我来说也有好处，所以我们保留了这一章。

到目前位置我们已经成功地引导了设备和处理中断而没使用一行汇编。这真是一个壮举!但是取决于你的目标架构你可能需要一些汇编才能达成目的。也有一些操作像是上下文切换需要汇编，等等。

问题是 *inline* 汇编(`asm!`) 和 *free form* 汇编(`global_asm!`)是不稳定的，没法估计它们什么时候将变得稳定，所以你不能在稳定版中使用它们。这不是一个演示，因为这里记录的是一些变通的方法。

为了激发对这章的兴趣，我们将修改下`HardFault`处理函数以提供关于产生了异常的栈帧的信息。

这是我们想要做的:

我们将让`rt` crate在向量表中放置一个跳向用户定义的`HardFault`的跳板，而不是让用户直接将它们的`HardFault`处理函数放在向量表中。

```
$ tail -n36 ../rt/src/lib.rs
```

```
extern "C" {
    fn NMI();
    fn HardFaultTrampoline(); // <- CHANGED!
    fn MemManage();
    fn BusFault();
    fn UsageFault();
    fn SVCall();
    fn PendSV();
    fn SysTick();
}

#[link_section = ".vector_table.exceptions"]
#[no_mangle]
pub static EXCEPTIONS: [Vector; 14] = [
    Vector { handler: NMI },
    Vector { handler: HardFaultTrampoline }, // <- CHANGED!
    Vector { handler: MemManage },
    Vector { handler: BusFault },
    Vector {
        handler: UsageFault,
    },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: SVCall },
    Vector { reserved: 0 },
    Vector { reserved: 0 },
    Vector { handler: PendSV },
    Vector { handler: SysTick },
];

#[no_mangle]
pub extern "C" fn DefaultExceptionHandler() {
    loop {}
}
```

这个跳板将读取栈指针并调用用户的`HardFault`处理函数。这个跳板必须要用汇编来写:

```
  mrs r0, MSP
  b HardFault
```

由于ARM ABI的工作原理，它将主堆栈指针(MSP)设置为`HardFault`函数/例程的第一个参数。这个MSP值碰巧也是一个指针，其指向被异常推进栈中的寄存器。有了这些改变，用户`HardFault`处理函数现在必须拥有签名`fn(&StackedRegisters) -> !`。

.s 文件

在一个外部文件中写汇编是一个通向稳定版的汇编的方法:

```
$ cat ../rt/asm.s
```

```
  .section .text.HardFaultTrampoline
  .global HardFaultTrampoline
  .thumb_func
HardFaultTrampoline:
  mrs r0, MSP
  b HardFault
```

并且使用`rt` crate的build script中的`cc` crate去把那个文件汇编成一个目标文件(`.o`)，然后变成一个归档文件(`.a`)。

```
$ cat ../rt/build.rs
```

```
use std::{env, error::Error, fs::File, io::Write, path::PathBuf};

use cc::Build;

fn main() -> Result<(), Box<dyn Error>> {
    // build directory for this crate
    let out_dir = PathBuf::from(env::var_os("OUT_DIR").unwrap());

    // extend the library search path
    println!("cargo:rustc-link-search={}", out_dir.display());

    // put `link.x` in the build directory
    File::create(out_dir.join("link.x"))?.write_all(include_bytes!("link.x"))?;

    // assemble the `asm.s` file
    Build::new().file("asm.s").compile("asm"); // <- NEW!

    // rebuild if `asm.s` changed
    println!("cargo:rerun-if-changed=asm.s"); // <- NEW!

    Ok(())
}
```

```
$ tail -n2 ../rt/Cargo.toml
```

```
[build-dependencies]
cc = "1.0.25"
```

完成了!

通过编写一个非常简单的程序我们可以确认向量表包含一个指向`HardFaultTrampoline`的指针。

```rust
#![no_main]
#![no_std]

use rt::entry;

entry!(main);

fn main() -> ! {
    loop {}
}

#[allow(non_snake_case)]
#[no_mangle]
pub fn HardFault(_ef: *const u32) -> ! {
    loop {}
}
```

这是反汇编。我们看下 `HardFaultTrampoline` 的地址。

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
```

```
app:    file format elf32-littlearm

Disassembly of section .text:

<HardFault>:
                b       0x40 <HardFault>        @ imm = #-0x4

<main>:
                b       0x42 <main>             @ imm = #-0x4

<Reset>:
                push    {r7, lr}
                mov     r7, sp
                bl      0x42 <main>             @ imm = #-0xa
                trap

<UsageFault>:
                b       0x4e <UsageFault>       @ imm = #-0x4

<HardFaultTrampoline>:
                mrs     r0, msp
                b       0x40 <HardFault>        @ imm = #-0x18
```

注意: 为了让这个反汇编更小我注释掉了RAM的初始化

现在看下向量表。第四项应该是`HardFaultTrampoline`的地址加一。

```
$ cargo objdump --bin app --release -- -s -j .vector_table
```

```
app:     file format elf32-littlearm
Contents of section .vector_table:
 0000 00000120 45000000 4f000000 51000000  ... E...O...Q...
 0010 4f000000 4f000000 4f000000 00000000  O...O...O.......
 0020 00000000 00000000 00000000 4f000000  ............O...
 0030 00000000 00000000 4f000000 4f000000  ........O...O...
```

`.o` / `.a` 文件

使用`cc` crate的缺点是它要求一些编译机器上的汇编器程序。比如当目标是ARM Cortex-M时，`cc` crate使用`arm-none-eabi-gcc`作为汇编器。

我们可以用`rt` crate来搬运一个预先汇编好的文件而不用在编译机器上汇编文件。这方法不需要在编译机器上拥有汇编器程序。然而，打包和发布crate的机器上仍然需要一个汇编器。

一个汇编(`.s`)文件和它的*编译*版:目标(`.o`)文件之间没有太大区别。汇编器不会做任何优化；它仅仅为目标架构选择正确的目标文件格式。

Cargo提供将归档文件(`.a`)和crates绑在一起的支持。使用`ar`命令我们可将目标文件打包进一个归档文件中，然后将归档文件和crate绑一起。事实上，这就是`cc` crate做的事；通过搜索一个在`target`文件夹中名为`output`的文件你可以看到`cc` crate调用的命令。

```
$ grep running $(find target -name output)
```

```
running: "arm-none-eabi-gcc" "-O0" "-ffunction-sections" "-fdata-sections" "-
fPIC" "-g" "-fno-omit-frame-pointer" "-mthumb" "-march=armv7-m" "-Wall" "-
Wextra" "-o" "/tmp/app/target/thumbv7m-none-eabi/debug/build/rt-
6ee84e54724f2044/out/asm.o" "-c" "asm.s"
running: "ar" "crs" "/tmp/app/target/thumbv7m-none-eabi/debug/build/rt-
6ee84e54724f2044/out/libasm.a" "/home/japaric/rust-
embedded/embedonomicon/ci/asm/app/target/thumbv7m-none-eabi/debug/build/rt-
6ee84e54724f2044/out/asm.o"
```

```
$ grep cargo $(find target -name output)
```

```
cargo:rustc-link-search=/tmp/app/target/thumbv7m-none-eabi/debug/build/rt-
6ee84e54724f2044/out
cargo:rustc-link-lib=static=asm
cargo:rustc-link-search=native=/tmp/app/target/thumbv7m-none-
eabi/debug/build/rt-6ee84e54724f2044/out
```

我们将做一些类似的事来生成一个归档文件。

```
$ # `cc` 使用的大多数标志在汇编时没有影响因此我们丢弃它们
$ arm-none-eabi-as -march=armv7-m asm.s -o asm.o

$ ar crs librt.a asm.o

$ arm-none-eabi-objdump -Cd librt.a
```

```
In archive librt.a:

asm.o:     file format elf32-littlearm


Disassembly of section .text.HardFaultTrampoline:

00000000 <HardFaultTrampoline>:
   0:   f3ef 8008       mrs     r0, MSP
   4:   e7fe            b.n     0 <HardFault>
```

接下来我们修改build script以把这个归档文件和`rt` rlib绑一起。

```
$ cat ../rt/build.rs
```

```
use std::{
    env,
    error::Error,
    fs::{self, File},
    io::Write,
    path::PathBuf,
};

fn main() -> Result<(), Box<dyn Error>> {
    // build directory for this crate
    let out_dir = PathBuf::from(env::var_os("OUT_DIR").unwrap());

    // extend the library search path
    println!("cargo:rustc-link-search={}", out_dir.display());

    // put `link.x` in the build directory
    File::create(out_dir.join("link.x"))?.write_all(include_bytes!("link.x"))?;

    // link to `librt.a`
    fs::copy("librt.a", out_dir.join("librt.a"))?; // <- NEW!
    println!("cargo:rustc-link-lib=static=rt"); // <- NEW!

    // rebuild if `librt.a` changed
    println!("cargo:rerun-if-changed=librt.a"); // <- NEW!

    Ok(())
}
```

现在我们可以用以前的简单程序测试这个新版本，我们将得到相同的输出

```
$ cargo objdump --bin app --release -- -d --no-show-raw-insn --print-imm-hex
```

```
app:    file format elf32-littlearm

Disassembly of section .text:

<HardFault>:
                b       0x40 <HardFault>        @ imm = #-0x4

<main>:
                b       0x42 <main>             @ imm = #-0x4

<Reset>:
                push    {r7, lr}
                mov     r7, sp
                bl      0x42 <main>             @ imm = #-0xa
                trap

<UsageFault>:
                b       0x4e <UsageFault>       @ imm = #-0x4

<HardFaultTrampoline>:
                mrs     r0, msp
                b       0x40 <HardFault>        @ imm = #-0x18
```

注意: 像之前一样我已经注释掉了RAM的初始化以让反汇编变得更小。

```
$ cargo objdump --bin app --release -- -s -j .vector_table
```

```
app:    file format elf32-littlearm
Contents of section .vector_table:
 0000 00000120 45000000 4f000000 51000000  ... E...O...Q...
 0010 4f000000 4f000000 4f000000 00000000  O...O...O.......
 0020 00000000 00000000 00000000 4f000000  ............O...
 0030 00000000 00000000 4f000000 4f000000  ........O...O...
```

搬运预先汇编好的归档文件的缺点是，在最糟糕的场景中，你将需要为每个你的库支持的编译目标搬运一个build artifact 。

Logging with symbols

这部分将向你展示如何使用符号 This section will show you how to utilize symbols and the ELF format to achieve super cheap logging.

Arbitrary symbols

Whenever we needed a stable symbol interface between crates we have mainly used the `no_mangle` attribute and sometimes the `export_name` attribute. The `export_name` attribute takes a string which becomes the name of the symbol whereas `#[no_mangle]` is basically sugar for `#[export_name = <item-name>]`.

Turns out we are not limited to single word names; we can use arbitrary strings, e.g. sentences, as the argument of the `export_name` attribute. As least when the output format is ELF anything that doesn't contain a null byte is fine.

Let's check that out:

```
$ cargo new --lib foo

$ cat foo/src/lib.rs
```

```
#[export_name = "Hello, world!"]
#[used]
static A: u8 = 0;

#[export_name = "こんにちは"]
#[used]
static B: u8 = 0;
```

```
$ ( cd foo && cargo nm --lib )
foo-d26a39c34b4e80ce.3lnzqy0jbpxj4pld.rcgu.o:
0000000000000000 r Hello, world!
0000000000000000 V __rustc_debug_gdb_scripts_section__
0000000000000000 r こんにちは
```

Can you see where this is going?

Encoding

Here's what we'll do: we'll create one `static` variable per log message but instead of storing the messages *in* the variables we'll store the messages in the variables' *symbol*

names. What we'll log then will not be the contents of the `static` variables but their addresses.

As long as the `static` variables are not zero sized each one will have a different address. What we're doing here is effectively encoding each message into a unique identifier, which happens to be the variable address. Some part of the log system will have to decode this id back into the message.

Let's write some code to illustrate the idea.

In this example we'll need some way to do I/O so we'll use the `cortex-m-semihosting` crate for that. Semihosting is a technique for having a target device borrow the host I/O capabilities; the host here usually refers to the machine that's debugging the target device. In our case, QEMU supports semihosting out of the box so there's no need for a debugger. On a real device you'll have other ways to do I/O like a serial port; we use semihosting in this case because it's the easiest way to do I/O on QEMU.

Here's the code

```
#![no_main]
#![no_std]

use core::fmt::Write;
use cortex_m_semihosting::{debug, hio};

use rt::entry;

entry!(main);

fn main() -> ! {
    let mut hstdout = hio::hstdout().unwrap();

    #[export_name = "Hello, world!"]
    static A: u8 = 0;

    let _ = writeln!(hstdout, "{:#x}", &A as *const u8 as usize);

    #[export_name = "Goodbye"]
    static B: u8 = 0;

    let _ = writeln!(hstdout, "{:#x}", &B as *const u8 as usize);

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

We also make use of the `debug::exit` API to have the program terminate the QEMU process. This is a convenience so we don't have to manually terminate the QEMU process.

And here's the `dependencies` section of the Cargo.toml:

```
[dependencies]
cortex-m-semihosting = "0.3.1"
rt = { path = "../rt" }
```

Now we can build the program

```
$ cargo build
```

To run it we'll have to add the `--semihosting-config` flag to our QEMU invocation:

```
$ qemu-system-arm \
      -cpu cortex-m3 \
      -machine lm3s6965evb \
      -nographic \
      -semihosting-config enable=on,target=native \
      -kernel target/thumbv7m-none-eabi/debug/app
```

```
0x1fe0
0x1fe1
```

> **NOTE**: These addresses may not be the ones you get locally because addresses of `static` variable are not guaranteed to remain the same when the toolchain is changed (e.g. optimizations may have improved).

Now we have two addresses printed to the console.

Decoding

How do we convert these addresses into strings? The answer is in the symbol table of the ELF file.

```
$ cargo objdump --bin app -- -t | grep '\.rodata\s*0*1\b'
```

```
00001fe1 g       .rodata                 00000001 Goodbye
00001fe0 g       .rodata                 00000001 Hello, world!
$ # first column is the symbol address; last column is the symbol name
```

`objdump -t` prints the symbol table. This table contains *all* the symbols but we are only looking for the ones in the `.rodata` section and whose size is one byte (our variables have type `u8`).

It's important to note that the address of the symbols will likely change when optimizing the program. Let's check that.

> **PROTIP** You can set `target.thumbv7m-none-eabi.runner` to the long QEMU command from before (`qemu-system-arm -cpu (..) -kernel`) in the Cargo configuration file (`.cargo/conifg`) to have `cargo run` use that *runner* to execute the output binary.

```
$ head -n2 .cargo/config
```

```
[target.thumbv7m-none-eabi]
runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel"
```

```
$ cargo run --release
     Running `qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel target/thumbv7m-none-
eabi/release/app`
```

```
0xb9c
0xb9d
```

```
$ cargo objdump --bin app --release -- -t | grep '\.rodata\s*0*1\b'
```

```
00000b9d g     O .rodata        00000001 Goodbye
00000b9c g     O .rodata        00000001 Hello, world!
```

So make sure to always look for the strings in the ELF file you executed.

Of course, the process of looking up the strings in the ELF file can be automated using a tool that parses the symbol table (`.symtab` section) contained in the ELF file. Implementing such tool is out of scope for this book and it's left as an exercise for the reader.

Making it zero cost

Can we do better? Yes, we can!

The current implementation places the `static` variables in `.rodata`, which means they occupy size in Flash even though we never use their contents. Using a little bit of linker script magic we can make them occupy *zero* space in Flash.

```
$ cat log.x
```

```
SECTIONS
{
  .log 0 (INFO) : {
    *(.log);
  }
}
```

We'll place the `static` variables in this new output `.log` section. This linker script will collect all the symbols in the `.log` sections of input object files and put them in an output `.log` section. We have seen this pattern in the Memory layout chapter.

The new bit here is the `(INFO)` part; this tells the linker that this section is a non-allocatable section. Non-allocatable sections are kept in the ELF binary as metadata but they are not loaded onto the target device.

We also specified the start address of this output section: the `0` in `.log 0 (INFO)`.

The other improvement we can do is switch from formatted I/O (`fmt::Write`) to binary I/O, that is send the addresses to the host as bytes rather than as strings.

Binary serialization can be hard but we'll keep things super simple by serializing each address as a single byte. With this approach we don't have to worry about endianness or framing. The downside of this format is that a single byte can only represent up to 256 different addresses.

Let's make those changes:

```
#![no_main]
#![no_std]

use cortex_m_semihosting::{debug, hio};

use rt::entry;

entry!(main);

fn main() -> ! {
    let mut hstdout = hio::hstdout().unwrap();

    #[export_name = "Hello, world!"]
    #[link_section = ".log"] // <- NEW!
    static A: u8 = 0;

    let address = &A as *const u8 as usize as u8;
    hstdout.write_all(&[address]).unwrap(); // <- CHANGED!

    #[export_name = "Goodbye"]
    #[link_section = ".log"] // <- NEW!
    static B: u8 = 0;

    let address = &B as *const u8 as usize as u8;
    hstdout.write_all(&[address]).unwrap(); // <- CHANGED!

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

Before you run this you'll have to append `-Tlog.x` to the arguments passed to the linker. That can be done in the Cargo configuration file.

```
$ cat .cargo/config
```

```
[target.thumbv7m-none-eabi]
runner = "qemu-system-arm -cpu cortex-m3 -machine lm3s6965evb -nographic -
semihosting-config enable=on,target=native -kernel"
rustflags = [
  "-C", "link-arg=-Tlink.x",
  "-C", "link-arg=-Tlog.x", # <- NEW!
]

[build]
target = "thumbv7m-none-eabi"
```

Now you can run it! Since the output now has a binary format we'll pipe it through the `xxd` command to reformat it as a hexadecimal string.

```
$ cargo run | xxd -p
```

```
0001
```

The addresses are `0x00` and `0x01`. Let's now look at the symbol table.

```
$ cargo objdump --bin app -- -t | grep '\.log'
```

```
00000001 g     O .log   00000001 Goodbye
00000000 g     O .log   00000001 Hello, world!
```

There are our strings. You'll notice that their addresses now start at zero; this is because we set a start address for the output `.log` section.

Each variable is 1 byte in size because we are using `u8` as their type. If we used something like `u16` then all address would be even and we would not be able to efficiently use all the address space (`0...255`).

Packaging it up

You've noticed that the steps to log a string are always the same so we can refactor them into a macro that lives in its own crate. Also, we can make the logging library more reusable by abstracting the I/O part behind a trait.

```
$ cargo new --lib log

$ cat log/src/lib.rs
```

```
#![no_std]

pub trait Log {
    type Error;

    fn log(&mut self, address: u8) -> Result<(), Self::Error>;
}

#[macro_export]
macro_rules! log {
    ($logger:expr, $string:expr) => {{
        #[export_name = $string]
        #[link_section = ".log"]
        static SYMBOL: u8 = 0;

        $crate::Log::log(&mut $logger, &SYMBOL as *const u8 as usize as u8)
    }};
}
```

Given that this library depends on the `.log` section it should be its responsibility to provide the `log.x` linker script so let's make that happen.

```
$ mv log.x ../log/
```

```
$ cat ../log/build.rs
```

```
use std::{env, error::Error, fs::File, io::Write, path::PathBuf};

fn main() -> Result<(), Box<dyn Error>> {
    // Put the linker script somewhere the linker can find it
    let out = PathBuf::from(env::var("OUT_DIR")?);

    File::create(out.join("log.x"))?.write_all(include_bytes!("log.x"))?;

    println!("cargo:rustc-link-search={}", out.display());

    Ok(())
}
```

Now we can refactor our application to use the `log!` macro:

```
$ cat src/main.rs
```

```
#![no_main]
#![no_std]

use cortex_m_semihosting::{
    debug,
    hio::{self, HStdout},
};

use log::{log, Log};
use rt::entry;

struct Logger {
    hstdout: HStdout,
}

impl Log for Logger {
    type Error = ();

    fn log(&mut self, address: u8) -> Result<(), ()> {
        self.hstdout.write_all(&[address])
    }
}

entry!(main);

fn main() -> ! {
    let hstdout = hio::hstdout().unwrap();
    let mut logger = Logger { hstdout };

    let _ = log!(logger, "Hello, world!");

    let _ = log!(logger, "Goodbye");

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}
```

Don't forget to update the `Cargo.toml` file to depend on the new `log` crate.

```
$ tail -n4 Cargo.toml
```

```
[dependencies]
cortex-m-semihosting = "0.3.1"
log = { path = "../log" }
rt = { path = "../rt" }
```

```
$ cargo run | xxd -p
```

```
0001
```

```
$ cargo objdump --bin app -- -t | grep '\.log'
```

```
00000001 g     O .log   00000001 Goodbye
00000000 g     O .log   00000001 Hello, world!
```

与之前一样的输出!

Bonus: Multiple log levels

Many logging frameworks provide ways to log messages at different *log levels*. These log levels convey the severity of the message: "this is an error", "this is just a warning", etc. These log levels can be used to filter out unimportant messages when searching for e.g. error messages.

We can extend our logging library to support log levels without increasing its footprint. Here's how we'll do that:

We have a flat address space for the messages: from `0` to `255` (inclusive). To keep things simple let's say we only want to differentiate between error messages and warning messages. We can place all the error messages at the beginning of the address space, and all the warning messages *after* the error messages. If the decoder knows the address of the first warning message then it can classify the messages. This idea can be extended to support more than two log levels.

Let's test the idea by replacing the `log` macro with two new macros: `error!` and `warn!`.

```
$ cat ../log/src/lib.rs
```

```
#![no_std]

pub trait Log {
    type Error;

    fn log(&mut self, address: u8) -> Result<(), Self::Error>;
}

/// Logs messages at the ERROR log level
#[macro_export]
macro_rules! error {
    ($logger:expr, $string:expr) => {{
        #[export_name = $string]
        #[link_section = ".log.error"] // <- CHANGED!
        static SYMBOL: u8 = 0;

        $crate::Log::log(&mut $logger, &SYMBOL as *const u8 as usize as u8)
    }};
}

/// Logs messages at the WARNING log level
#[macro_export]
macro_rules! warn {
    ($logger:expr, $string:expr) => {{
        #[export_name = $string]
        #[link_section = ".log.warning"] // <- CHANGED!
        static SYMBOL: u8 = 0;

        $crate::Log::log(&mut $logger, &SYMBOL as *const u8 as usize as u8)
    }};
}
```

We distinguish errors from warnings by placing the messages in different link sections.

The next thing we have to do is update the linker script to place error messages before the warning messages.

```
$ cat ../log/log.x
```

```
SECTIONS
{
  .log 0 (INFO) : {
    *(.log.error);
    __log_warning_start__ = .;
    *(.log.warning);
  }
}
```

We also give a name, `__log_warning_start__`, to the boundary between the errors and the warnings. The address of this symbol will be the address of the first warning message.

We can now update the application to make use of these new macros.

```
$ cat src/main.rs
```

```
#![no_main]
#![no_std]

use cortex_m_semihosting::{
    debug,
    hio::{self, HStdout},
};

use log::{error, warn, Log};
use rt::entry;

entry!(main);

fn main() -> ! {
    let hstdout = hio::hstdout().unwrap();
    let mut logger = Logger { hstdout };

    let _ = warn!(logger, "Hello, world!"); // <- CHANGED!

    let _ = error!(logger, "Goodbye"); // <- CHANGED!

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}

struct Logger {
    hstdout: HStdout,
}

impl Log for Logger {
    type Error = ();

    fn log(&mut self, address: u8) -> Result<(), ()> {
        self.hstdout.write_all(&[address])
    }
}
```

The output won't change much:

```
$ cargo run | xxd -p
```

```
0100
```

We still get two bytes in the output but the error is given the address 0 and the warning is given the address 1 even though the warning was logged first.

Now look at the symbol table.

```
$ cargo objdump --bin app -- -t | grep '\.log'
```

```
00000000 g     O .log   00000001 Goodbye
00000001 g     O .log   00000001 Hello, world!
00000001 g       .log   00000000 __log_warning_start__
```

There's now an extra symbol, `__log_warning_start__`, in the `.log` section. The address of this symbol is the address of the first warning message. Symbols with addresses lower than this value are errors, and the rest of symbols are warnings.

With an appropriate decoder you could get the following human readable output from all this information:

```
WARNING Hello, world!
ERROR Goodbye
```

If you liked this section check out the `stlog` logging framework which is a complete implementation of this idea.

Global singletons

In this section we'll cover how to implement a global, shared singleton. The embedded Rust book covered local, owned singletons which are pretty much unique to Rust. Global singletons are essentially the singleton pattern you see in C and C++; they are not specific to embedded development but since they involve symbols they seemed a good fit for the embedonomicon.

> **TODO**(resources team) link "the embedded Rust book" to the singletons section when it's up

---

To illustrate this section we'll extend the logger we developed in the last section to support global logging. The result will be very similar to the `#[global_allocator]` feature covered in the embedded Rust book.

> **TODO**(resources team) link `#[global_allocator]` to the collections chapter of the book when it's in a more stable location.

Here's the summary of what we want to:

In the last section we created a `log!` macro to log messages through a specific logger, a value that implements the `Log` trait. The syntax of the `log!` macro is `log!(logger, "String")`. We want to extend the macro such that `log!("String")` also works. Using the `logger`-less version should log the message through a global logger; this is how `std::println!` works. We'll also need a mechanism to declare what the global logger is; this is the part that's similar to `#[global_allocator]`.

It could be that the global logger is declared in the top crate and it could also be that the type of the global logger is defined in the top crate. In this scenario the dependencies can *not* know the exact type of the global logger. To support this scenario we'll need some indirection.

Instead of hardcoding the type of the global logger in the `log` crate we'll declare only the *interface* of the global logger in that crate. That is we'll add a new trait, `GlobalLog`, to the `log` crate. The `log!` macro will also have to make use of that trait.

```
$ cat ../log/src/lib.rs
```

```rust
#![no_std]

// NEW!
pub trait GlobalLog: Sync {
    fn log(&self, address: u8);
}

pub trait Log {
    type Error;

    fn log(&mut self, address: u8) -> Result<(), Self::Error>;
}

#[macro_export]
macro_rules! log {
    // NEW!
    ($string:expr) => {
        unsafe {
            extern "Rust" {
                static LOGGER: &'static dyn $crate::GlobalLog;
            }

            #[export_name = $string]
            #[link_section = ".log"]
            static SYMBOL: u8 = 0;

            $crate::GlobalLog::log(LOGGER, &SYMBOL as *const u8 as usize as u8)
        }
    };

    ($logger:expr, $string:expr) => {{
        #[export_name = $string]
        #[link_section = ".log"]
        static SYMBOL: u8 = 0;

        $crate::Log::log(&mut $logger, &SYMBOL as *const u8 as usize as u8)
    }};
}

// NEW!
#[macro_export]
macro_rules! global_logger {
    ($logger:expr) => {
        #[no_mangle]
        pub static LOGGER: &dyn $crate::GlobalLog = &$logger;
    };
}
```

There's quite a bit to unpack here.

Let's start with the trait.

```rust
pub trait GlobalLog: Sync {
    fn log(&self, address: u8);
}
```

Both `GlobalLog` and `Log` have a `log` method. The difference is that `GlobalLog.log` takes a shared reference to the receiver (`&self`). This is necessary because the global logger will be a `static` variable. More on that later.

The other difference is that `GlobalLog.log` doesn't return a `Result`. This means that it can *not* report errors to the caller. This is not a strict requirement for traits used to implement global singletons. Error handling in global singletons is fine but then all users of the global version of the `log!` macro have to agree on the error type. Here we are simplifying the interface a bit by having the `GlobalLog` implementer deal with the errors.

Yet another difference is that `GlobalLog` requires that the implementer is `Sync`, that is that it can be shared between threads. This is a requirement for values placed in `static` variables; their types must implement the `Sync` trait.

At this point it may not be entirely clear why the interface has to look this way. The other parts of the crate will make this clearer so keep reading.

Next up is the `log!` macro:

```
    ($string:expr) => {
        unsafe {
            extern "Rust" {
                static LOGGER: &'static dyn $crate::GlobalLog;
            }

            #[export_name = $string]
            #[link_section = ".log"]
            static SYMBOL: u8 = 0;

            $crate::GlobalLog::log(LOGGER, &SYMBOL as *const u8 as usize as u8)
        }
    };
```

When called without a specific `$logger` the macros uses an `extern` `static` variable called `LOGGER` to log the message. This variable *is* the global logger that's defined somewhere else; that's why we use the `extern` block. We saw this pattern in the main interface chapter.

We need to declare a type for `LOGGER` or the code won't type check. We don't know the concrete type of `LOGGER` at this point but we know, or rather require, that it implements the `GlobalLog` trait so we can use a trait object here.

The rest of the macro expansion looks very similar to the expansion of the local version of the `log!` macro so I won't explain it here as it's explained in the previous chapter.

Now that we know that `LOGGER` has to be a trait object it's clearer why we omitted the associated `Error` type in `GlobalLog`. If we had not omitted then we would have need to pick a type for `Error` in the type signature of `LOGGER`. This is what I earlier meant by "all users of `log!` would need to agree on the error type".

Now the final piece: the `global_logger!` macro. It could have been a proc macro attribute but it's easier to write a `macro_rules!` macro.

```
#[macro_export]
macro_rules! global_logger {
    ($logger:expr) => {
        #[no_mangle]
        pub static LOGGER: &dyn $crate::GlobalLog = &$logger;
    };
}
```

This macro creates the `LOGGER` variable that `log!` uses. Because we need a stable ABI interface we use the `no_mangle` attribute. This way the symbol name of `LOGGER` will be "LOGGER" which is what the `log!` macro expects.

The other important bit is that the type of this static variable must exactly match the type used in the expansion of the `log!` macro. If they don't match Bad Stuff will happen due to ABI mismatch.

Let's write an example that uses this new global logger functionality.

```
$ cat src/main.rs
```

```
#![no_main]
#![no_std]

use cortex_m::interrupt;
use cortex_m_semihosting::{
    debug,
    hio::{self, HStdout},
};

use log::{global_logger, log, GlobalLog};
use rt::entry;

struct Logger;

global_logger!(Logger);

entry!(main);

fn main() -> ! {
    log!("Hello, world!");

    log!("Goodbye");

    debug::exit(debug::EXIT_SUCCESS);

    loop {}
}

impl GlobalLog for Logger {
    fn log(&self, address: u8) {
        // we use a critical section (`interrupt::free`) to make the access to
the
        // `static mut` variable interrupt safe which is required for memory
safety
        interrupt::free(|_| unsafe {
            static mut HSTDOUT: Option<HStdout> = None;

            // lazy initialization
            if HSTDOUT.is_none() {
                HSTDOUT = Some(hio::hstdout()?);
            }

            let hstdout = HSTDOUT.as_mut().unwrap();

            hstdout.write_all(&[address])
        }).ok(); // `.ok()` = ignore errors
    }
}
```

TODO(resources team) use `cortex_m::Mutex` instead of a `static mut` variable when `const fn` is stabilized.

We had to add `cortex-m` to the dependencies.

```
$ tail -n5 Cargo.toml
```

```
[dependencies]
cortex-m = "0.5.7"
cortex-m-semihosting = "0.3.1"
log = { path = "../log" }
rt = { path = "../rt" }
```

This is a port of one of the examples written in the previous section. The output is the same as what we got back there.

```
$ cargo run | xxd -p
```

```
0001
```

```
$ cargo objdump --bin app -- -t | grep '\.log'
```

```
00000001 g     O .log   00000001 Goodbye
00000000 g     O .log   00000001 Hello, world!
```

Some readers may be concerned about this implementation of global singletons not being zero cost because it uses trait objects which involve dynamic dispatch, that is method calls are performed through a vtable lookup.

However, it appears that LLVM is smart enough to eliminate the dynamic dispatch when compiling with optimizations / LTO. This can be confirmed by searching for `LOGGER` in the symbol table.

```
$ cargo objdump --bin app --release -- -t | grep LOGGER
```

```

```

If the `static` is missing that means that there is no vtable and that LLVM was capable of transforming all the `LOGGER.log` calls into `Logger.log` calls.

Direct Memory Access (DMA)

This section covers the core requirements for building a memory safe API around DMA transfers.

The DMA peripheral is used to perform memory transfers in parallel to the work of the processor (the execution of the main program). A DMA transfer is more or less equivalent to spawning a thread (see `thread::spawn`) to do a `memcpy`. We'll use the fork-join model to illustrate the requirements of a memory safe API.

Consider the following DMA primitives:

```rust
/// A singleton that represents a single DMA channel (channel 1 in this case)
///
/// This singleton has exclusive access to the registers of the DMA channel 1
pub struct Dma1Channel1 {
    // ..
}

impl Dma1Channel1 {
    /// Data will be written to this `address`
    ///
    /// `inc` indicates whether the address will be incremented after every
byte transfer
    ///
    /// NOTE this performs a volatile write
    pub fn set_destination_address(&mut self, address: usize, inc: bool) {
        // ..
    }

    /// Data will be read from this `address`
    ///
    /// `inc` indicates whether the address will be incremented after every
byte transfer
    ///
    /// NOTE this performs a volatile write
    pub fn set_source_address(&mut self, address: usize, inc: bool) {
        // ..
    }

    /// Number of bytes to transfer
    ///
    /// NOTE this performs a volatile write
    pub fn set_transfer_length(&mut self, len: usize) {
        // ..
    }

    /// Starts the DMA transfer
    ///
    /// NOTE this performs a volatile write
    pub fn start(&mut self) {
        // ..
    }

    /// Stops the DMA transfer
    ///
    /// NOTE this performs a volatile write
    pub fn stop(&mut self) {
        // ..
    }

    /// Returns `true` if there's a transfer in progress
    ///
    /// NOTE this performs a volatile read
    pub fn in_progress() -> bool {
        // ..
    }
}
```

Assume that the `Dma1Channel1` is statically configured to work with serial port (AKA UART or USART) #1, `Serial1`, in one-shot mode (i.e. not circular mode). `Serial1` provides the following *blocking* API:

```
/// A singleton that represents serial port #1
pub struct Serial1 {
    // ..
}

impl Serial1 {
    /// Reads out a single byte
    ///
    /// NOTE: blocks if no byte is available to be read
    pub fn read(&mut self) -> Result<u8, Error> {
        // ..
    }

    /// Sends out a single byte
    ///
    /// NOTE: blocks if the output FIFO buffer is full
    pub fn write(&mut self, byte: u8) -> Result<(), Error> {
        // ..
    }
}
```

Let's say we want to extend `Serial1` API to (a) asynchronously send out a buffer and (b) asynchronously fill a buffer.

We'll start with a memory unsafe API and we'll iterate on it until it's completely memory safe. On each step we'll show you how the API can be broken to make you aware of the issues that need to be addressed when dealing with asynchronous memory operations.

A first stab

For starters, let's try to use the `Write::write_all` API as a reference. To keep things simple let's ignore all error handling.

```rust
/// A singleton that represents serial port #1
pub struct Serial1 {
    // NOTE: we extend this struct by adding the DMA channel singleton
    dma: Dma1Channel1,
    // ..
}

impl Serial1 {
    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all<'a>(mut self, buffer: &'a [u8]) -> Transfer<&'a [u8]> {
        self.dma.set_destination_address(USART1_TX, false);
        self.dma.set_source_address(buffer.as_ptr() as usize, true);
        self.dma.set_transfer_length(buffer.len());

        self.dma.start();

        Transfer { buffer }
    }
}

/// A DMA transfer
pub struct Transfer<B> {
    buffer: B,
}

impl<B> Transfer<B> {
    /// Returns `true` if the DMA transfer has finished
    pub fn is_done(&self) -> bool {
        !Dma1Channel1::in_progress()
    }

    /// Blocks until the transfer is done and returns the buffer
    pub fn wait(self) -> B {
        // Busy wait until the transfer is done
        while !self.is_done() {}

        self.buffer
    }
}
```

NOTE: `Transfer` could expose a futures or generator based API instead of the API shown above. That's an API design question that has little bearing on the memory safety of the overall API so we won't delve into it in this text.

We can also implement an asynchronous version of `Read::read_exact`.

```
impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<'a>(&mut self, buffer: &'a mut [u8]) -> Transfer<&'a mut
[u8]> {
        self.dma.set_source_address(USART1_RX, false);
        self.dma
            .set_destination_address(buffer.as_mut_ptr() as usize, true);
        self.dma.set_transfer_length(buffer.len());

        self.dma.start();

        Transfer { buffer }
    }
}
```

Here's how to use the `write_all` API:

```
fn write(serial: Serial1) {
    // fire and forget
    serial.write_all(b"Hello, world!\n");

    // do other stuff
}
```

And here's an example of using the `read_exact` API:

```
fn read(mut serial: Serial1) {
    let mut buf = [0; 16];
    let t = serial.read_exact(&mut buf);

    // do other stuff

    t.wait();

    match buf.split(|b| *b == b'\n').next() {
        Some(b"some-command") => { /* do something */ }
        _ => { /* do something else */ }
    }
}
```

`mem::forget`

`mem::forget` is a safe API. If our API is truly safe then we should be able to use both together without running into undefined behavior. However, that's not the case; consider the following example:

```
fn unsound(mut serial: Serial1) {
    start(&mut serial);
    bar();
}

#[inline(never)]
fn start(serial: &mut Serial1) {
    let mut buf = [0; 16];

    // start a DMA transfer and forget the returned `Transfer` value
    mem::forget(serial.read_exact(&mut buf));
}

#[inline(never)]
fn bar() {
    // stack variables
    let mut x = 0;
    let mut y = 0;

    // use `x` and `y`
}
```

Here we start a DMA transfer, in `start`, to fill an array allocated on the stack and then `mem::forget` the returned `Transfer` value. Then we proceed to return from `start` and execute the function `bar`.

This series of operations results in undefined behavior. The DMA transfer writes to stack memory but that memory is released when `start` returns and then reused by `bar` to allocate variables like `x` and `y`. At runtime this could result in variables `x` and `y` changing their value at random times. The DMA transfer could also overwrite the state (e.g. link register) pushed onto the stack by the prologue of function `bar`.

Note that if we had not use `mem::forget`, but `mem::drop`, it would have been possible to make `Transfer`'s destructor stop the DMA transfer and then the program would have been safe. But one can *not* rely on destructors running to enforce memory safety because `mem::forget` and memory leaks (see RC cycles) are safe in Rust.

We can fix this particular problem by changing the lifetime of the buffer from `'a` to `'static` in both APIs.

```
impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact(&mut self, buffer: &'static mut [u8]) ->
Transfer<&'static mut [u8]> {
        // .. same as before ..
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all(mut self, buffer: &'static [u8]) -> Transfer<&'static
[u8]> {
        // .. same as before ..
    }
}
```

If we try to replicate the previous problem we note that `mem::forget` no longer causes problems.

```
#[allow(dead_code)]
fn sound(mut serial: Serial1, buf: &'static mut [u8; 16]) {
    // NOTE `buf` is moved into `foo`
    foo(&mut serial, buf);
    bar();
}

#[inline(never)]
fn foo(serial: &mut Serial1, buf: &'static mut [u8]) {
    // start a DMA transfer and forget the returned `Transfer` value
    mem::forget(serial.read_exact(buf));
}

#[inline(never)]
fn bar() {
    // stack variables
    let mut x = 0;
    let mut y = 0;

    // use `x` and `y`
}
```

As before, the DMA transfer continues after `mem::forget`-ing the `Transfer` value. This time that's not an issue because `buf` is statically allocated (e.g. `static mut` variable) and not on the stack.

Overlapping use

Our API doesn't prevent the user from using the `Serial` interface while the DMA transfer is in progress. This could lead the transfer to fail or data to be lost.

There are several ways to prevent overlapping use. One way is to have `Transfer` take ownership of `Serial1` and return it back when `wait` is called.

```
/// A DMA transfer
pub struct Transfer<B> {
    buffer: B,
    // NOTE: added
    serial: Serial1,
}

impl<B> Transfer<B> {
    /// Blocks until the transfer is done and returns the buffer
    // NOTE: the return value has changed
    pub fn wait(self) -> (B, Serial1) {
        // Busy wait until the transfer is done
        while !self.is_done() {}

        (self.buffer, self.serial)
    }

    // ..
}

impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    // NOTE we now take `self` by value
    pub fn read_exact(mut self, buffer: &'static mut [u8]) -> Transfer<&'static
mut [u8]> {
        // .. same as before ..

        Transfer {
            buffer,
            // NOTE: added
            serial: self,
        }
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    // NOTE we now take `self` by value
    pub fn write_all(mut self, buffer: &'static [u8]) -> Transfer<&'static
[u8]> {
        // .. same as before ..

        Transfer {
            buffer,
            // NOTE: added
            serial: self,
        }
    }
}
```

The move semantics statically prevent access to `Serial1` while the transfer is in progress.

```
fn read(serial: Serial1, buf: &'static mut [u8; 16]) {
    let t = serial.read_exact(buf);

    // let byte = serial.read(); //~ ERROR: `serial` has been moved

    // .. do stuff ..

    let (serial, buf) = t.wait();

    // .. do more stuff ..
}
```

There are other ways to prevent overlapping use. For example, a (`Cell`) flag that indicates whether a DMA transfer is in progress could be added to `Serial1`. When the flag is set `read`, `write`, `read_exact` and `write_all` would all return an error (e.g. `Error::InUse`) at runtime. The flag would be set when `write_all` / `read_exact` is used and cleared in `Transfer.wait`.

Compiler (mis)optimizations

The compiler is free to re-order and merge non-volatile memory operations to better optimize a program. With our current API, this freedom can lead to undefined behavior. Consider the following example:

```
fn reorder(serial: Serial1, buf: &'static mut [u8]) {
    // zero the buffer (for no particular reason)
    buf.iter_mut().for_each(|byte| *byte = 0);

    let t = serial.read_exact(buf);

    // ... do other stuff ..

    let (buf, serial) = t.wait();

    buf.reverse();

    // .. do stuff with `buf` ..
}
```

Here the compiler is free to move `buf.reverse()` before `t.wait()`, which would result in a data race: both the processor and the DMA would end up modifying `buf` at the same time. Similarly the compiler can move the zeroing operation to after `read_exact`, which would also result in a data race.

To prevent these problematic reorderings we can use a `compiler_fence`

```
impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact(mut self, buffer: &'static mut [u8]) -> Transfer<&'static
mut [u8]> {
        self.dma.set_source_address(USART1_RX, false);
        self.dma
            .set_destination_address(buffer.as_mut_ptr() as usize, true);
        self.dma.set_transfer_length(buffer.len());

        // NOTE: added
        atomic::compiler_fence(Ordering::Release);

        // NOTE: this is a volatile *write*
        self.dma.start();

        Transfer {
            buffer,
            serial: self,
        }
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all(mut self, buffer: &'static [u8]) -> Transfer<&'static
[u8]> {
        self.dma.set_destination_address(USART1_TX, false);
        self.dma.set_source_address(buffer.as_ptr() as usize, true);
        self.dma.set_transfer_length(buffer.len());

        // NOTE: added
        atomic::compiler_fence(Ordering::Release);

        // NOTE: this is a volatile *write*
        self.dma.start();

        Transfer {
            buffer,
            serial: self,
        }
    }
}

impl<B> Transfer<B> {
    /// Blocks until the transfer is done and returns the buffer
    pub fn wait(self) -> (B, Serial1) {
        // NOTE: this is a volatile *read*
        while !self.is_done() {}

        // NOTE: added
        atomic::compiler_fence(Ordering::Acquire);

        (self.buffer, self.serial)
    }
```

```
    // ..
}
```

We use `Ordering::Release` in `read_exact` and `write_all` to prevent all preceding memory operations from being moved *after* `self.dma.start()`, which performs a volatile write.

Likewise, we use `Ordering::Acquire` in `Transfer.wait` to prevent all subsequent memory operations from being moved *before* `self.is_done()`, which performs a volatile read.

To better visualize the effect of the fences here's a slightly tweaked version of the example from the previous section. We have added the fences and their orderings in the comments.

```
fn reorder(serial: Serial1, buf: &'static mut [u8], x: &mut u32) {
    // zero the buffer (for no particular reason)
    buf.iter_mut().for_each(|byte| *byte = 0);

    *x += 1;

    let t = serial.read_exact(buf); // compiler_fence(Ordering::Release) ▲

    // NOTE: the processor can't access `buf` between the fences
    // ... do other stuff ..
    *x += 2;

    let (buf, serial) = t.wait(); // compiler_fence(Ordering::Acquire) ▼

    *x += 3;

    buf.reverse();

    // .. do stuff with `buf` ..
}
```

The zeroing operation can *not* be moved *after* `read_exact` due to the `Release` fence. Similarly, the `reverse` operation can *not* be moved *before* `wait` due to the `Acquire` fence. The memory operations *between* both fences *can* be freely reordered across the fences but none of those operations involves `buf` so such reorderings do *not* result in undefined behavior.

Note that `compiler_fence` is a bit stronger than what's required. For example, the fences will prevent the operations on `x` from being merged even though we know that `buf` doesn't overlap with `x` (due to Rust aliasing rules). However, there exist no intrinsic that's more fine grained than `compiler_fence`.

Don't we need a memory barrier?

That depends on the target architecture. In the case of Cortex M0 to M4F cores, AN321 says:

> 3.2 Typical usages
>
> (..)
>
> The use of DMB is rarely needed in Cortex-M processors because they do not reorder memory transactions. However, it is needed if the software is to be reused on other ARM processors, especially multi-master systems. For example:
>
> - DMA controller configuration. A barrier is required between a CPU memory access and a DMA operation.
>
> (..)
>
> 4.18 Multi-master systems
>
> (..)
>
> Omitting the DMB or DSB instruction in the examples in Figure 41 on page 47 and Figure 42 would not cause any error because the Cortex-M processors:
>
> - do not re-order memory transfers
> - do not permit two write transfers to be overlapped.

Where Figure 41 shows a DMB (memory barrier) instruction being used before starting a DMA transaction.

In the case of Cortex-M7 cores you'll need memory barriers (DMB/DSB) if you are using the data cache (DCache), unless you manually invalidate the buffer used by the DMA. Even with the data cache disabled, memory barriers might still be required to avoid reordering in the store buffer.

If your target is a multi-core system then it's very likely that you'll need memory barriers.

If you do need the memory barrier then you need to use `atomic::fence` instead of `compiler_fence`. That should generate a DMB instruction on Cortex-M devices.

Generic buffer

Our API is more restrictive that it needs to be. For example, the following program won't be accepted even though it's valid.

```
fn reuse(serial: Serial1, msg: &'static mut [u8]) {
    // send a message
    let t1 = serial.write_all(msg);

    // ..

    let (msg, serial) = t1.wait(); // `msg` is now `&'static [u8]`

    msg.reverse();

    // now send it in reverse
    let t2 = serial.write_all(msg);

    // ..

    let (buf, serial) = t2.wait();

    // ..
}
```

To accept such program we can make the buffer argument generic.

```
// as-slice = "0.1.0"
use as_slice::{AsMutSlice, AsSlice};

impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<B>(mut self, mut buffer: B) -> Transfer<B>
    where
        B: AsMutSlice<Element = u8>,
    {
        // NOTE: added
        let slice = buffer.as_mut_slice();
        let (ptr, len) = (slice.as_mut_ptr(), slice.len());

        self.dma.set_source_address(USART1_RX, false);

        // NOTE: tweaked
        self.dma.set_destination_address(ptr as usize, true);
        self.dma.set_transfer_length(len);

        atomic::compiler_fence(Ordering::Release);
        self.dma.start();

        Transfer {
            buffer,
            serial: self,
        }
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    fn write_all<B>(mut self, buffer: B) -> Transfer<B>
    where
        B: AsSlice<Element = u8>,
    {
        // NOTE: added
        let slice = buffer.as_slice();
        let (ptr, len) = (slice.as_ptr(), slice.len());

        self.dma.set_destination_address(USART1_TX, false);

        // NOTE: tweaked
        self.dma.set_source_address(ptr as usize, true);
        self.dma.set_transfer_length(len);

        atomic::compiler_fence(Ordering::Release);
        self.dma.start();

        Transfer {
            buffer,
            serial: self,
        }
    }
}
```

> **NOTE:** `AsRef<[u8]>` (`AsMut<[u8]>`) could have been used instead of `AsSlice<Element = u8>` (`AsMutSlice<Element = u8`).

Now the `reuse` program will be accepted.

Immovable buffers

With this modification the API will also accept arrays by value (e.g. `[u8; 16]`). However, using arrays can result in pointer invalidation. Consider the following program.

```
fn invalidate(serial: Serial1) {
    let t = start(serial);

    bar();

    let (buf, serial) = t.wait();
}

#[inline(never)]
fn start(serial: Serial1) -> Transfer<[u8; 16]> {
    // array allocated in this frame
    let buffer = [0; 16];

    serial.read_exact(buffer)
}

#[inline(never)]
fn bar() {
    // stack variables
    let mut x = 0;
    let mut y = 0;

    // use `x` and `y`
}
```

The `read_exact` operation will use the address of the `buffer` local to the `start` function. That local `buffer` will be freed when `start` returns and the pointer used in `read_exact` will become invalidated. You'll end up with a situation similar to the `unsound` example.

To avoid this problem we require that the buffer used with our API retains its memory location even when it's moved. The `Pin` newtype provides such guarantee. We can update our API to required that all buffers are "pinned" first.

> **NOTE:** To compile all the programs below this point you'll need Rust `>=1.33.0`. As of time of writing (2019-01-04) that means using the nightly channel.

```
/// A DMA transfer
pub struct Transfer<B> {
    // NOTE: changed
    buffer: Pin<B>,
    serial: Serial1,
}

impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<B>(mut self, mut buffer: Pin<B>) -> Transfer<B>
    where
        // NOTE: bounds changed
        B: DerefMut,
        B::Target: AsMutSlice<Element = u8> + Unpin,
    {
        // .. same as before ..
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all<B>(mut self, buffer: Pin<B>) -> Transfer<B>
    where
        // NOTE: bounds changed
        B: Deref,
        B::Target: AsSlice<Element = u8>,
    {
        // .. same as before ..
    }
}
```

NOTE: We could have used the `StableDeref` trait instead of the `Pin` newtype but opted for `Pin` since it's provided in the standard library.

With this new API we can use `&'static mut` references, `Box`-ed slices, `Rc`-ed slices, etc.

```rust
fn static_mut(serial: Serial1, buf: &'static mut [u8]) {
    let buf = Pin::new(buf);

    let t = serial.read_exact(buf);

    // ..

    let (buf, serial) = t.wait();

    // ..
}

fn boxed(serial: Serial1, buf: Box<[u8]>) {
    let buf = Pin::new(buf);

    let t = serial.read_exact(buf);

    // ..

    let (buf, serial) = t.wait();

    // ..
}
```

`'static` bound

Does pinning let us safely use stack allocated arrays? The answer is *no*. Consider the following example.

```
fn unsound(serial: Serial1) {
    start(serial);

    bar();
}

// pin-utils = "0.1.0-alpha.4"
use pin_utils::pin_mut;

#[inline(never)]
fn start(serial: Serial1) {
    let buffer = [0; 16];

    // pin the `buffer` to this stack frame
    // `buffer` now has type `Pin<&mut [u8; 16]>`
    pin_mut!(buffer);

    mem::forget(serial.read_exact(buffer));
}

#[inline(never)]
fn bar() {
    // stack variables
    let mut x = 0;
    let mut y = 0;

    // use `x` and `y`
}
```

As seen many times before, the above program runs into undefined behavior due to stack frame corruption.

The API is unsound for buffers of type `Pin<&'a mut [u8]>` where `'a` is *not* `'static`. To prevent the problem we have to add a `'static` bound in some places.

```
impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<B>(mut self, mut buffer: Pin<B>) -> Transfer<B>
    where
        // NOTE: added 'static bound
        B: DerefMut + 'static,
        B::Target: AsMutSlice<Element = u8> + Unpin,
    {
        // .. same as before ..
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all<B>(mut self, buffer: Pin<B>) -> Transfer<B>
    where
        // NOTE: added 'static bound
        B: Deref + 'static,
        B::Target: AsSlice<Element = u8>,
    {
        // .. same as before ..
    }
}
```

Now the problematic program will be rejected.

Destructors

Now that the API accepts `Box`-es and other types that have destructors we need to decide what to do when `Transfer` is early-dropped.

Normally, `Transfer` values are consumed using the `wait` method but it's also possible to, implicitly or explicitly, `drop` the value before the transfer is over. For example, dropping a `Transfer<Box<[u8]>>` value will cause the buffer to be deallocated. This can result in undefined behavior if the transfer is still in progress as the DMA would end up writing to deallocated memory.

In such scenario one option is to make `Transfer.drop` stop the DMA transfer. The other option is to make `Transfer.drop` wait for the transfer to finish. We'll pick the former option as it's cheaper.

```
/// A DMA transfer
pub struct Transfer<B> {
    // NOTE: always `Some` variant
    inner: Option<Inner<B>>,
}

// NOTE: previously named `Transfer<B>`
struct Inner<B> {
    buffer: Pin<B>,
    serial: Serial1,
}

impl<B> Transfer<B> {
    /// Blocks until the transfer is done and returns the buffer
    pub fn wait(mut self) -> (Pin<B>, Serial1) {
        while !self.is_done() {}

        atomic::compiler_fence(Ordering::Acquire);

        let inner = self
            .inner
            .take()
            .unwrap_or_else(|| unsafe { hint::unreachable_unchecked() });
        (inner.buffer, inner.serial)
    }
}

impl<B> Drop for Transfer<B> {
    fn drop(&mut self) {
        if let Some(inner) = self.inner.as_mut() {
            // NOTE: this is a volatile write
            inner.serial.dma.stop();

            // we need a read here to make the Acquire fence effective
            // we do *not* need this if `dma.stop` does a RMW operation
            unsafe {
                ptr::read_volatile(&0);
            }

            // we need a fence here for the same reason we need one in
`Transfer.wait`
            atomic::compiler_fence(Ordering::Acquire);
        }
    }
}

impl Serial1 {
    /// Receives data into the given `buffer` until it's filled
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn read_exact<B>(mut self, mut buffer: Pin<B>) -> Transfer<B>
    where
        B: DerefMut + 'static,
        B::Target: AsMutSlice<Element = u8> + Unpin,
    {
        // .. same as before ..
```

```
        Transfer {
            inner: Some(Inner {
                buffer,
                serial: self,
            }),
        }
    }

    /// Sends out the given `buffer`
    ///
    /// Returns a value that represents the in-progress DMA transfer
    pub fn write_all<B>(mut self, buffer: Pin<B>) -> Transfer<B>
    where
        B: Deref + 'static,
        B::Target: AsSlice<Element = u8>,
    {
        // .. same as before ..

        Transfer {
            inner: Some(Inner {
                buffer,
                serial: self,
            }),
        }
    }
}
```

Now the DMA transfer will be stopped before the buffer is deallocated.

```
fn reuse(serial: Serial1) {
    let buf = Pin::new(Box::new([0; 16]));

    let t = serial.read_exact(buf); // compiler_fence(Ordering::Release) ▲

    // ..

    // this stops the DMA transfer and frees memory
    mem::drop(t); // compiler_fence(Ordering::Acquire) ▼

    // this likely reuses the previous memory allocation
    let mut buf = Box::new([0; 16]);

    // .. do stuff with `buf` ..
}
```

Summary

To sum it up, we need to consider all the following points to achieve memory safe DMA transfers:

- Use immovable buffers plus indirection: `Pin<B>`. Alternatively, you can use the `StableDeref` trait.
- The ownership of the buffer must be passed to the DMA : `B: 'static`.
- Do *not* rely on destructors running for memory safety. Consider what happens if `mem::forget` is used with your API.
- *Do* add a custom destructor that stops the DMA transfer, or waits for it to finish. Consider what happens if `mem::drop` is used with your API.

This text leaves out up several details required to build a production grade DMA abstraction, like configuring the DMA channels (e.g. streams, circular vs one-shot mode, etc.), alignment of buffers, error handling, how to make the abstraction device-agnostic, etc. All those aspects are left as an exercise for the reader / community (`:P`).

与编译器支持有关的笔记

这本书使用了一个内置的*编译器*目标，`thumbv7m-none-eabi`，Rust团队为其分发了一个`rust-std`组件，`rust-std`是跟`core`和`std`一样的预先编译好的crates的集合。

如果你想尝试为一个不同的目标架构复制这本书的内容，你需要考虑下Rust为(编译)目标所提供的支持在什么级别。

LLVM 支持

自Rust 1.28以来，官方的Rust编译器，`rustc`，使用LLVM生成(机器)代码。Rust对某个架构提供的最小级别的支持是在`rustc`中让它的LLVM后端可用。通过运行下面的命令，你可以看到所有的`rustc`通过LLVM支持的架构:

```
$ # 运行这个命令，你需要安装`cargo-binutils`
$ cargo objdump -- -version
LLVM (http://llvm.org/):
  LLVM version 7.0.0svn
  Optimized build.
  Default target: x86_64-unknown-linux-gnu
  Host CPU: skylake

  Registered Targets:
    aarch64    - AArch64 (little endian)
    aarch64_be - AArch64 (big endian)
    arm        - ARM
    arm64      - ARM64 (little endian)
    armeb      - ARM (big endian)
    hexagon    - Hexagon
    mips       - Mips
    mips64     - Mips64 [experimental]
    mips64el   - Mips64el [experimental]
    mipsel     - Mipsel
    msp430     - MSP430 [experimental]
    nvptx      - NVIDIA PTX 32-bit
    nvptx64    - NVIDIA PTX 64-bit
    ppc32      - PowerPC 32
    ppc64      - PowerPC 64
    ppc64le    - PowerPC 64 LE
    sparc      - Sparc
    sparcel    - Sparc LE
    sparcv9    - Sparc V9
    systemz    - SystemZ
    thumb      - Thumb
    thumbeb    - Thumb (big endian)
    wasm32     - WebAssembly 32-bit
    wasm64     - WebAssembly 64-bit
    x86        - 32-bit X86: Pentium-Pro and above
    x86-64     - 64-bit X86: EM64T and AMD64
```

如果LLVM支持了你感兴趣的架构，但是构建的`rustc`没有使能其后端(自Rust 1.28以来AVR就是这样的情况)，那么你将需要修改Rust源码以启用它。PR rust-lang/rust#52787 的前两个commits能让你知道需要做的改变。

换句话说，如果LLVM不支持这个架构，但是LLVM的一个分支支持，在编译`rustc`之前你将需要使用这个分支替代原来的LLVM 。Rust编译系统允许这么做且理论上，它应该只需要修改`llvm`的子模组让它指向那个分支就可以了。

如果只有一些供应商提供的GCC支持你的目标架构，你可以选择使用`mrustc`，一个非官方的Rust编译器，去把你的Rust程序翻译成C代码然后使用GCC编译它。

内置的目标

一个编译目标不仅是它的架构。每个目标都有一个与它关联的规范，其中描述了它的架构，它的操作系统和默认的链接器。

Rust编译器知道几个目标。这些被*内置进*编译器里且可以通过下列的命令被列出来:

```
$ rustc --print target-list | column
aarch64-fuchsia                     mipsisa32r6el-unknown-linux-gnu
aarch64-linux-android               mipsisa64r6-unknown-linux-gnuabi64
aarch64-pc-windows-msvc             mipsisa64r6el-unknown-linux-gnuabi64
aarch64-unknown-cloudabi            msp430-none-elf
aarch64-unknown-freebsd             nvptx64-nvidia-cuda
aarch64-unknown-hermit              powerpc-unknown-linux-gnu
aarch64-unknown-linux-gnu           powerpc-unknown-linux-gnuspe
aarch64-unknown-linux-musl          powerpc-unknown-linux-musl
aarch64-unknown-netbsd              powerpc-unknown-netbsd
aarch64-unknown-none                powerpc-wrs-vxworks
aarch64-unknown-none-softfloat      powerpc-wrs-vxworks-spe
aarch64-unknown-openbsd             powerpc64-unknown-freebsd
aarch64-unknown-redox               powerpc64-unknown-linux-gnu
aarch64-uwp-windows-msvc            powerpc64-unknown-linux-musl
aarch64-wrs-vxworks                 powerpc64-wrs-vxworks
arm-linux-androideabi               powerpc64le-unknown-linux-gnu
arm-unknown-linux-gnueabi           powerpc64le-unknown-linux-musl
arm-unknown-linux-gnueabihf         riscv32i-unknown-none-elf
arm-unknown-linux-musleabi          riscv32imac-unknown-none-elf
arm-unknown-linux-musleabihf        riscv32imc-unknown-none-elf
armebv7r-none-eabi                  riscv64gc-unknown-linux-gnu
armebv7r-none-eabihf                riscv64gc-unknown-none-elf
armv4t-unknown-linux-gnueabi        riscv64imac-unknown-none-elf
armv5te-unknown-linux-gnueabi       s390x-unknown-linux-gnu
armv5te-unknown-linux-musleabi      sparc-unknown-linux-gnu
armv6-unknown-freebsd               sparc64-unknown-linux-gnu
armv6-unknown-netbsd-eabihf         sparc64-unknown-netbsd
armv7-linux-androideabi             sparc64-unknown-openbsd
armv7-unknown-cloudabi-eabihf       sparcv9-sun-solaris
armv7-unknown-freebsd               thumbv6m-none-eabi
armv7-unknown-linux-gnueabi         thumbv7a-pc-windows-msvc
armv7-unknown-linux-gnueabihf       thumbv7em-none-eabi
armv7-unknown-linux-musleabi        thumbv7em-none-eabihf
armv7-unknown-linux-musleabihf      thumbv7m-none-eabi
armv7-unknown-netbsd-eabihf         thumbv7neon-linux-androideabi
armv7-wrs-vxworks-eabihf            thumbv7neon-unknown-linux-gnueabihf
armv7a-none-eabi                    thumbv7neon-unknown-linux-musleabihf
armv7a-none-eabihf                  thumbv8m.base-none-eabi
armv7r-none-eabi                    thumbv8m.main-none-eabi
armv7r-none-eabihf                  thumbv8m.main-none-eabihf
asmjs-unknown-emscripten            wasm32-unknown-emscripten
hexagon-unknown-linux-musl          wasm32-unknown-unknown
i586-pc-windows-msvc                wasm32-wasi
i586-unknown-linux-gnu              x86_64-apple-darwin
i586-unknown-linux-musl             x86_64-fortanix-unknown-sgx
i686-apple-darwin                   x86_64-fuchsia
i686-linux-android                  x86_64-linux-android
i686-pc-windows-gnu                 x86_64-linux-kernel
i686-pc-windows-msvc                x86_64-pc-solaris
i686-unknown-cloudabi               x86_64-pc-windows-gnu
i686-unknown-freebsd                x86_64-pc-windows-msvc
i686-unknown-haiku                  x86_64-rumprun-netbsd
i686-unknown-linux-gnu              x86_64-sun-solaris
i686-unknown-linux-musl             x86_64-unknown-cloudabi
i686-unknown-netbsd                 x86_64-unknown-dragonfly
i686-unknown-openbsd                x86_64-unknown-freebsd
```

```
i686-unknown-uefi                    x86_64-unknown-haiku
i686-uwp-windows-gnu                 x86_64-unknown-hermit
i686-uwp-windows-msvc                x86_64-unknown-hermit-kernel
i686-wrs-vxworks                     x86_64-unknown-illumos
mips-unknown-linux-gnu               x86_64-unknown-l4re-uclibc
mips-unknown-linux-musl              x86_64-unknown-linux-gnu
mips-unknown-linux-uclibc            x86_64-unknown-linux-gnux32
mips64-unknown-linux-gnuabi64        x86_64-unknown-linux-musl
mips64-unknown-linux-muslabi64       x86_64-unknown-netbsd
mips64el-unknown-linux-gnuabi64      x86_64-unknown-openbsd
mips64el-unknown-linux-muslabi64     x86_64-unknown-redox
mipsel-unknown-linux-gnu             x86_64-unknown-uefi
mipsel-unknown-linux-musl            x86_64-uwp-windows-gnu
mipsel-unknown-linux-uclibc          x86_64-uwp-windows-msvc
mipsisa32r6-unknown-linux-gnu        x86_64-wrs-vxworks
```

使用下列的命令你可以打印出某个目标的规范:

```
$ rustc +nightly -Z unstable-options --print target-spec-json --target
thumbv7m-none-eabi
{
  "abi-blacklist": [
    "stdcall",
    "fastcall",
    "vectorcall",
    "thiscall",
    "win64",
    "sysv64"
  ],
  "arch": "arm",
  "data-layout": "e-m:e-p:32:32-i64:64-v128:64:128-a:0:32-n32-S64",
  "emit-debug-gdb-scripts": false,
  "env": "",
  "executables": true,
  "is-builtin": true,
  "linker": "arm-none-eabi-gcc",
  "linker-flavor": "gcc",
  "llvm-target": "thumbv7m-none-eabi",
  "max-atomic-width": 32,
  "os": "none",
  "panic-strategy": "abort",
  "relocation-model": "static",
  "target-c-int-width": "32",
  "target-endian": "little",
  "target-pointer-width": "32",
  "vendor": ""
}
```

如果这些内置的目标都不适合你的目标系统，你将必须通过使用JSON格式编写你自己的目标规范来生成一个自制的目标，在下个章节中会讲到。

`rust-std` 组件

Rust团队通过`rustup`为某些内置的目标分发`rust-std`组件。这个组件是跟`core`和`std`一样的预先编译好的crates的集合，它要求交叉编译。

通过运行下列的命令，你可以找到具有一个`rust-std`组件的目标列表:

```
$ rustup target list | column
aarch64-apple-ios                          mipsel-unknown-linux-musl
aarch64-fuchsia                            nvptx64-nvidia-cuda
aarch64-linux-android                      powerpc-unknown-linux-gnu
aarch64-pc-windows-msvc                    powerpc64-unknown-linux-gnu
aarch64-unknown-linux-gnu                  powerpc64le-unknown-linux-gnu
aarch64-unknown-linux-musl                 riscv32i-unknown-none-elf
aarch64-unknown-none                       riscv32imac-unknown-none-elf
aarch64-unknown-none-softfloat             riscv32imc-unknown-none-elf
arm-linux-androideabi                      riscv64gc-unknown-linux-gnu
arm-unknown-linux-gnueabi                  riscv64gc-unknown-none-elf
arm-unknown-linux-gnueabihf                riscv64imac-unknown-none-elf
arm-unknown-linux-musleabi                 s390x-unknown-linux-gnu
arm-unknown-linux-musleabihf               sparc64-unknown-linux-gnu
armebv7r-none-eabi                         sparcv9-sun-solaris
armebv7r-none-eabihf                       thumbv6m-none-eabi
armv5te-unknown-linux-gnueabi              thumbv7em-none-eabi
armv5te-unknown-linux-musleabi             thumbv7em-none-eabihf
armv7-linux-androideabi                    thumbv7m-none-eabi
armv7-unknown-linux-gnueabi                thumbv7neon-linux-androideabi
armv7-unknown-linux-gnueabihf              thumbv7neon-unknown-linux-gnueabihf
armv7-unknown-linux-musleabi               thumbv8m.base-none-eabi
armv7-unknown-linux-musleabihf             thumbv8m.main-none-eabi
armv7a-none-eabi                           thumbv8m.main-none-eabihf
armv7r-none-eabi                           wasm32-unknown-emscripten
armv7r-none-eabihf                         wasm32-unknown-unknown
asmjs-unknown-emscripten                   wasm32-wasi
i586-pc-windows-msvc                       x86_64-apple-darwin
i586-unknown-linux-gnu                     x86_64-apple-ios
i586-unknown-linux-musl                    x86_64-fortanix-unknown-sgx
i686-linux-android                         x86_64-fuchsia
i686-pc-windows-gnu                        x86_64-linux-android
i686-pc-windows-msvc                       x86_64-pc-windows-gnu
i686-unknown-freebsd                       x86_64-pc-windows-msvc
i686-unknown-linux-gnu                     x86_64-rumprun-netbsd
i686-unknown-linux-musl                    x86_64-sun-solaris
mips-unknown-linux-gnu                     x86_64-unknown-cloudabi
mips-unknown-linux-musl                    x86_64-unknown-freebsd
mips64-unknown-linux-gnuabi64              x86_64-unknown-linux-gnu (default)
mips64-unknown-linux-muslabi64             x86_64-unknown-linux-gnux32
mips64el-unknown-linux-gnuabi64            x86_64-unknown-linux-musl
mips64el-unknown-linux-muslabi64           x86_64-unknown-netbsd
mipsel-unknown-linux-gnu                   x86_64-unknown-redox
```

如果你的目标没有`rust-std`组件，或者你正在使用一个自制的目标，那么你必须要使用一个nightly版的工具链去构建标准库。看下一页，关于构建自制目标。

制造一个自定义的目标

If a custom target triple is not available for your platform, you must create a custom target file that describes your target to rustc.

Keep in mind that it is required to use a nightly compiler to build the core library, which must be done for a target unknown to rustc.

Deciding on a target triple

Many targets already have a known triple used to describe them, typically in the form ARCH-VENDOR-SYS-ABI. You should aim to use the same triple that LLVM uses; however, it may differ if you need to specify additional information to Rust that LLVM does not know about. Although the triple is technically only for human use, it's important for it to be unique and descriptive especially if the target will be upstreamed in the future.

The ARCH part is typically just the architecture name, except in the case of 32-bit ARM. For example, you would probably use x86_64 for those processors, but specify the exact ARM architecture version. Typical values might be `armv7`, `armv5te`, or `thumbv7neon`. Take a look at the names of the built-in targets for inspiration.

The VENDOR part is optional and describes the manufacturer. Omitting this field is the same as using `unknown`.

The SYS part describes the OS that is used. Typical values include `win32`, `linux`, and `darwin` for desktop platforms. `none` is used for bare-metal usage.

The ABI part describes how the process starts up. `eabi` is used for bare metal, while `gnu` is used for glibc, `musl` for musl, etc.

Now that you have a target triple, create a file with the name of the triple and a `.json` extension. For example, a file describing `armv7a-none-eabi` would have the filename `armv7a-none-eabi.json`.

Fill the target file

The target file must be valid JSON. There are two places where its contents are described: `Target`, where every field is mandatory, and `TargetOptions`, where every field is optional. **All underscores are replaced with hyphens**.

The recommended way is to base your target file on the specification of a built-in target that's similar to your target system, then tweak it to match the properties of your target

system. To do so, use the command `rustc +nightly -Z unstable-options --print target-spec-json --target $SOME_SIMILAR_TARGET`, using a target that's already built into the compiler.

You can pretty much copy that output into your file. Start with a few modifications:

- Remove `"is-builtin": true`
- Fill `llvm-target` with the triple that LLVM expects
- Decide on a panicking strategy. A bare metal implementation will likely use `"panic-strategy": "abort"`. If you decide not to `abort` on panicking, unless you tell Cargo to per-project, you must define an eh_personality function.
- Configure atomics. Pick the first option that describes your target:
  - I have a single-core processor, no threads, **no interrupts**, or any way for multiple things to be happening in parallel: if you are **sure** that is the case, such as WASM (for now), you may set `"singlethread": true`. This will configure LLVM to convert all atomic operations to use their single threaded counterparts. Incorrectly using this option may result in UB if using threads or interrupts.
  - I have native atomic operations: set `max-atomic-width` to the biggest type in bits that your target can operate on atomically. For example, many ARM cores have 32-bit atomic operations. You may set `"max-atomic-width": 32` in that case.
  - I have no native atomic operations, but I can emulate them myself: set `max-atomic-width` to the highest number of bits that you can emulate up to 128, then implement all of the atomic and sync functions expected by LLVM as `#[no_mangle] unsafe extern "C"`. These functions have been standardized by gcc, so the gcc documentation may have more notes. Missing functions will cause a linker error, while incorrectly implemented functions will possibly cause UB. For example, if you have a single-core, single-thread processor with interrupts, you can implement these functions to disable interrupts, perform the regular operation, and then re-enable them.
  - I have no native atomic operations: you'll have to do some unsafe work to manually ensure synchronization in your code. You must set `"max-atomic-width": 0`.
- Change the linker if integrating with an existing toolchain. For example, if you're using a toolchain that uses a custom build of gcc, set `"linker-flavor": "gcc"` and `linker` to the command name of your linker. If you require additional linker arguments, use `pre-link-args` and `post-link-args` as so:

```
"pre-link-args": {
    "gcc": [
        "-Wl,--as-needed",
        "-Wl,-z,noexecstack",
        "-m64"
    ]
},
"post-link-args": {
    "gcc": [
        "-Wl,--allow-multiple-definition",
        "-Wl,--start-group,-lc,-lm,-lgcc,-lstdc++,-lsupc++,--end-group"
    ]
}
```

Ensure that the linker type is the key within `link-args`.

- Configure LLVM features. Run `llc -march=ARCH -mattr=help` where ARCH is the base architecture (not including the version in the case of ARM) to list the available features and their descriptions. **If your target requires strict memory alignment access (e.g. `armv5te`), make sure that you enable `strict-align`**. To enable a feature, place a plus before it. Likewise, to disable a feature, place a minus before it. Features should be comma separated like so: `"features": "+soft-float,+neon"`. Note that this may not be necessary if LLVM knows enough about your target based on the provided triple and CPU.
- Configure the CPU that LLVM uses if you know it. This will enable CPU-specific optimizations and features. At the top of the output of the command in the last step, there is a list of known CPUs. If you know that you will be targeting a specific CPU, you may set it in the `cpu` field in the JSON target file.

使用目标文件

一旦你有一个目标规范文件，你可能 Once you have a target specification file, you may refer to it by its path or by its name (i.e. excluding `.json`) if it is in the current directory or in `$RUST_TARGET_PATH`.

Verify that it is readable by rustc:

```
❯ rustc --print cfg --target foo.json # or just foo if in the current directory
debug_assertions
target_arch="arm"
target_endian="little"
target_env=""
target_feature="mclass"
target_feature="v7"
target_has_atomic="16"
target_has_atomic="32"
target_has_atomic="8"
target_has_atomic="cas"
target_has_atomic="ptr"
target_os="none"
target_pointer_width="32"
target_vendor=""
```

Now, you finally get to use it! Many resources have been recommending `xargo` or `cargo-xbuild`. However, its successor, cargo's `build-std` feature, has received a lot of work recently and has quickly reached feature parity with the other options. As such, this guide will only cover that option.

Start with a bare minimum `no_std` program. Now, run `cargo build -Z build-std=core --target foo.json`, again using the above rules about referencing the path. Hopefully, you should now have a binary in the target directory.

You may optionally configure cargo to always use your target. See the recommendations at the end of the page about the smallest `no_std` program. However, you'll currently have to use the flag `-Z build-std=core` as that option is unstable.

Build additional built-in crates

When using cargo's `build-std` feature, you can choose which crates to compile in. By default, when only passing `-Z build-std`, `std`, `core`, and `alloc` are compiled. However, you may want to exclude `std` when compiling for bare-metal. To do so, specify the crated you'd like after `build-std`. For example, to include `core` and `alloc`, pass `-Z build-std=core,alloc`.

Troubleshooting

language item required, but not found: `eh_personality`

Either add `"panic-strategy": "abort"` to your target file, or define an eh_personality function. Alternatively, tell Cargo to ignore it.

undefined reference to `__sync_val_compare_and_swap_#`

Rust thinks that your target has atomic instructions, but LLVM doesn't. Go back to the step about configuring atomics. You will need to reduce the number in `max-atomic-width`. See #58500 for more details.

could not find `sync` in `alloc`

Similar to the above case, Rust doesn't think that you have atomics. You must implement them yourself or tell Rust that you have atomic instructions.

multiple definition of `__(something)`

You're likely linking your Rust program with code built from another language, and the other language includes compiler built-ins that Rust also creates. To fix this, you'll need to tell your linker to allow multiple definitions. If using gcc, you may add:

```
"post-link-args": {
    "gcc": [
        "-Wl,--allow-multiple-definition"
    ]
}
```

error adding symbols: file format not recognized

Switch to cargo's `build-std` feature and update your compiler. This was a bug introduced for a few compiler builds that tried to pass in internal Rust object to an external linker.