VGA Text Mode

Feb 26, 2018

The VGA text mode is a simple way to print text to the screen. In this post, we create an interface that makes its usage safe and simple, by encapsulating all unsafety in a separate module. We also implement support for Rust's formatting macros.

This blog is openly developed on GitHub. If you have any problems or questions, please open an issue there. You can also leave comments at the bottom. The complete source code for this post can be found in the `post-03` branch.

▶ **Table of Contents**

The VGA Text Buffer

To print a character to the screen in VGA text mode, one has to write it to the text buffer of the VGA hardware. The VGA text buffer is a two-dimensional array with typically 25 rows and 80 columns, which is directly rendered to the screen. Each array entry describes a single screen character through the following format:

Bit(s)	Value
0-7	ASCII code point
8-11	Foreground color
12-14	Background color
15	Blink

The first byte represents the character that should be printed in the ASCII encoding. To be exact, it isn't exactly ASCII, but a character set named *code page 437* with some additional characters and slight modifications. For simplicity, we proceed to call it an ASCII character in this post.

The second byte defines how the character is displayed. The first four bits define the foreground color, the next three bits the background color, and the last bit whether the character should blink. The following colors are available:

Number	Color	Number + Bright Bit	Bright Color
0x0	Black	0x8	Dark Gray

Number	Color	Number + Bright Bit	Bright Color
0x1	Blue	0x9	Light Blue
0x2	Green	0xa	Light Green
0x3	Cyan	0xb	Light Cyan
0x4	Red	0xc	Light Red
0x5	Magenta	0xd	Pink
0x6	Brown	0xe	Yellow
0x7	Light Gray	0xf	White

Bit 4 is the *bright bit*, which turns for example blue into light blue. For the background color, this bit is repurposed as the blink bit.

The VGA text buffer is accessible via memory-mapped I/O to the address `0xb8000`. This means that reads and writes to that address don't access the RAM, but directly the text buffer on the VGA hardware. This means that we can read and write it through normal memory operations to that address.

Note that memory-mapped hardware might not support all normal RAM operations. For example, a device could only support byte-wise reads and return junk when an `u64` is read. Fortunately, the text buffer supports normal reads and writes, so that we don't have to treat it in special way.

A Rust Module

Now that we know how the VGA buffer works, we can create a Rust module to handle printing:

```
// in src/main.rs
mod vga_buffer;
```

For the content of this module we create a new `src/vga_buffer.rs` file. All of the code below goes into our new module (unless specified otherwise).

Colors

First, we represent the different colors using an enum:

```
// in src/vga_buffer.rs

#[allow(dead_code)]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(u8)]
pub enum Color {
    Black = 0,
```

```
    Blue = 1,
    Green = 2,
    Cyan = 3,
    Red = 4,
    Magenta = 5,
    Brown = 6,
    LightGray = 7,
    DarkGray = 8,
    LightBlue = 9,
    LightGreen = 10,
    LightCyan = 11,
    LightRed = 12,
    Pink = 13,
    Yellow = 14,
    White = 15,
}
```

We use a C-like enum here to explicitly specify the number for each color. Because of the `repr(u8)` attribute each enum variant is stored as an `u8`. Actually 4 bits would be sufficient, but Rust doesn't have an `u4` type.

Normally the compiler would issue a warning for each unused variant. By using the `#[allow(dead_code)]` attribute we disable these warnings for the `Color` enum.

By deriving the `Copy`, `Clone`, `Debug`, `PartialEq`, and `Eq` traits, we enable copy semantics for the type and make it printable and comparable.

To represent a full color code that specifies foreground and background color, we create a newtype on top of `u8`:

```
// in src/vga_buffer.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(transparent)]
struct ColorCode(u8);

impl ColorCode {
    fn new(foreground: Color, background: Color) -> ColorCode {
        ColorCode((background as u8) << 4 | (foreground as u8))
    }
}
```

The `ColorCode` struct contains the full color byte, containing foreground and background color. Like before, we derive the `Copy` and `Debug` traits for it. To ensure that the `ColorCode` has the exact same data layout as an `u8`, we use the `repr(transparent)` attribute.

Text Buffer

Now we can add structures to represent a screen character and the text buffer:

```
// in src/vga_buffer.rs

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
#[repr(C)]
struct ScreenChar {
    ascii_character: u8,
    color_code: ColorCode,
}

const BUFFER_HEIGHT: usize = 25;
const BUFFER_WIDTH: usize = 80;

#[repr(transparent)]
struct Buffer {
    chars: [[ScreenChar; BUFFER_WIDTH]; BUFFER_HEIGHT],
}
```

Since the field ordering in default structs is undefined in Rust, we need the `repr(C)` attribute. It guarantees that the struct's fields are laid out exactly like in a C struct and thus guarantees the correct field ordering. For the `Buffer` struct, we use `repr(transparent)` again to ensure that it has the same memory layout as its single field.

To actually write to screen, we now create a writer type:

```
// in src/vga_buffer.rs

pub struct Writer {
    column_position: usize,
    color_code: ColorCode,
    buffer: &'static mut Buffer,
}
```

The writer will always write to the last line and shift lines up when a line is full (or on `\n`). The `column_position` field keeps track of the current position in the last row. The current foreground and background colors are specified by `color_code` and a reference to the VGA buffer is stored in `buffer`. Note that we need an explicit lifetime here to tell the compiler how long the reference is valid. The `'static` lifetime specifies that the reference is valid for the whole program run time (which is true for the VGA text buffer).

Printing

Now we can use the `Writer` to modify the buffer's characters. First we create a method to write a single ASCII byte:

```rust
// in src/vga_buffer.rs

impl Writer {
    pub fn write_byte(&mut self, byte: u8) {
        match byte {
            b'\n' => self.new_line(),
            byte => {
                if self.column_position >= BUFFER_WIDTH {
                    self.new_line();
                }

                let row = BUFFER_HEIGHT - 1;
                let col = self.column_position;

                let color_code = self.color_code;
                self.buffer.chars[row][col] = ScreenChar {
                    ascii_character: byte,
                    color_code,
                };
                self.column_position += 1;
            }
        }
    }

    fn new_line(&mut self) {/* TODO */}
}
```

If the byte is the newline byte `\n`, the writer does not print anything. Instead it calls a `new_line` method, which we'll implement later. Other bytes get printed to the screen in the second match case.

When printing a byte, the writer checks if the current line is full. In that case, a `new_line` call is required before to wrap the line. Then it writes a new `ScreenChar` to the buffer at the current position. Finally, the current column position is advanced.

To print whole strings, we can convert them to bytes and print them one-by-one:

```rust
// in src/vga_buffer.rs

impl Writer {
    pub fn write_string(&mut self, s: &str) {
        for byte in s.bytes() {
            match byte {
                // printable ASCII byte or newline
                0x20..=0x7e | b'\n' => self.write_byte(byte),
                // not part of printable ASCII range
                _ => self.write_byte(0xfe),
            }
```

```
        }
    }
}
```

The VGA text buffer only supports ASCII and the additional bytes of code page 437. Rust strings are UTF-8 by default, so they might contain bytes that are not supported by the VGA text buffer. We use a match to differentiate printable ASCII bytes (a newline or anything in between a space character and a `~` character) and unprintable bytes. For unprintable bytes, we print a `■` character, which has the hex code `0xfe` on the VGA hardware.

Try it out!

To write some characters to the screen, you can create a temporary function:

```
// in src/vga_buffer.rs

pub fn print_something() {
    let mut writer = Writer {
        column_position: 0,
        color_code: ColorCode::new(Color::Yellow, Color::Black),
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    };

    writer.write_byte(b'H');
    writer.write_string("ello ");
    writer.write_string("Wörld!");
}
```

It first creates a new Writer that points to the VGA buffer at `0xb8000`. The syntax for this might seem a bit strange: First, we cast the integer `0xb8000` as an mutable raw pointer. Then we convert it to a mutable reference by dereferencing it (through `*`) and immediately borrowing it again (through `&mut`). This conversion requires an `unsafe` block, since the compiler can't guarantee that the raw pointer is valid.

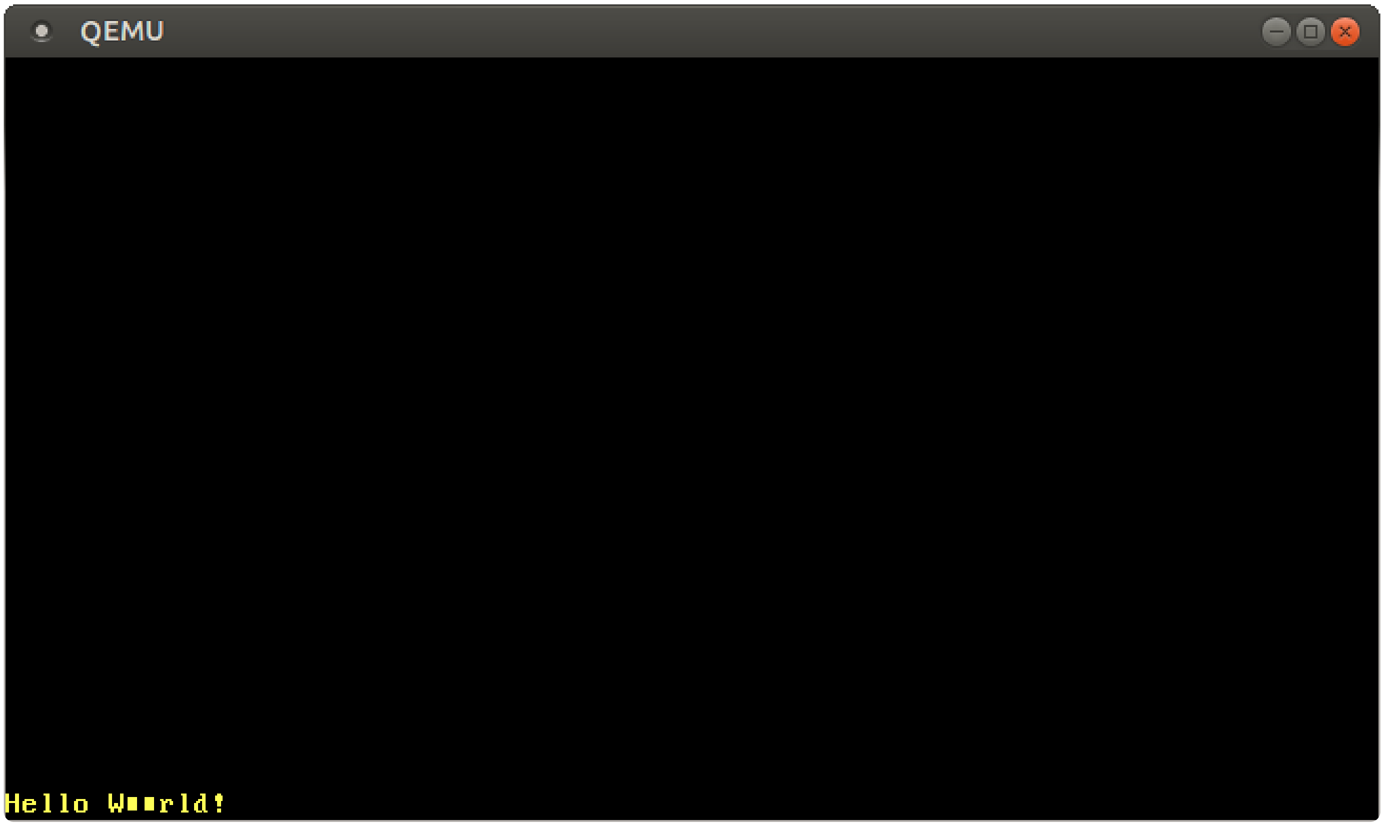
Then it writes the byte `b'H'` to it. The `b` prefix creates a byte literal, which represents an ASCII character. By writing the strings `"ello "` and `"Wörld!"`, we test our `write_string` method and the handling of unprintable characters. To see the output, we need to call the `print_something` function from our `_start` function:

```
// in src/main.rs

#[no_mangle]
pub extern "C" fn _start() -> ! {
    vga_buffer::print_something();

    loop {}
}
```

When we run our project now, a `Hello W■■rld!` should be printed in the *lower* left corner of the screen in yellow:



Notice that the `ö` is printed as two `■` characters. That's because `ö` is represented by two bytes in UTF-8, which both don't fall into the printable ASCII range. In fact, this is a fundamental property of UTF-8: the individual bytes of multi-byte values are never valid ASCII.

Volatile

We just saw that our message was printed correctly. However, it might not work with future Rust compilers that optimize more aggressively.

The problem is that we only write to the `Buffer` and never read from it again. The compiler doesn't know that we really access VGA buffer memory (instead of normal RAM) and knows nothing about the side effect that some characters appear on the screen. So it might decide that these writes are unnecessary and can be omitted. To avoid this erroneous optimization, we need to specify these writes as *volatile*. This tells the compiler that the write has side effects and should not be optimized away.

In order to use volatile writes for the VGA buffer, we use the volatile library. This *crate* (this is how packages are called in the Rust world) provides a `Volatile` wrapper type with `read` and `write` methods. These methods internally use the read_volatile and write_volatile functions of the core library and thus guarantee that the reads/writes are not optimized away.

We can add a dependency on the `volatile` crate by adding it to the `dependencies` section of our `Cargo.toml`:

```
# in Cargo.toml

[dependencies]
volatile = "0.2.6"
```

Make sure to specify `volatile` version `0.2.6`. Newer versions of the crate are not compatible with this post. The `0.2.6` is the semantic version number. For more information, see the Specifying Dependencies guide of the cargo documentation.

Let's use it to make writes to the VGA buffer volatile. We update our `Buffer` type as follows:

```
// in src/vga_buffer.rs

use volatile::Volatile;

struct Buffer {
    chars: [[Volatile<ScreenChar>; BUFFER_WIDTH]; BUFFER_HEIGHT],
}
```

Instead of a `ScreenChar`, we're now using a `Volatile<ScreenChar>`. (The `Volatile` type is generic and can wrap (almost) any type). This ensures that we can't accidentally write to it through a "normal" write. Instead, we have to use the `write` method now.

This means that we have to update our `Writer::write_byte` method:

```
// in src/vga_buffer.rs

impl Writer {
    pub fn write_byte(&mut self, byte: u8) {
        match byte {
            b'\n' => self.new_line(),
            byte => {
                ...

                self.buffer.chars[row][col].write(ScreenChar {
                    ascii_character: byte,
                    color_code,
                });
                ...
            }
        }
    }
    ...
}
```

Instead of a normal assignment using `=`, we're now using the `write` method. This guarantees that the compiler will never optimize away this write.

Formatting Macros

It would be nice to support Rust's formatting macros, too. That way, we can easily print different types like integers or floats. To support them, we need to implement the `core::fmt::Write` trait. The only required method of this trait is `write_str` that looks quite similar to our `write_string` method, just with a `fmt::Result` return type:

```
// in src/vga_buffer.rs

use core::fmt;

impl fmt::Write for Writer {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        self.write_string(s);
        Ok(())
    }
}
```

The `Ok(())` is just a `Ok` Result containing the `()` type.

Now we can use Rust's built-in `write!` / `writeln!` formatting macros:

```
// in src/vga_buffer.rs

pub fn print_something() {
    use core::fmt::Write;
    let mut writer = Writer {
        column_position: 0,
        color_code: ColorCode::new(Color::Yellow, Color::Black),
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    };

    writer.write_byte(b'H');
    writer.write_string("ello! ");
    write!(writer, "The numbers are {} and {}", 42, 1.0/3.0).unwrap();
}
```

Now you should see a `Hello! The numbers are 42 and 0.3333333333333333` at the bottom of the screen. The `write!` call returns a `Result` which causes a warning if not used, so we call the `unwrap` function on it, which panics if an error occurs. This isn't a problem in our case, since writes to the VGA buffer never fail.

Newlines

Right now, we just ignore newlines and characters that don't fit into the line anymore. Instead we want to move every character one line up (the top line gets deleted) and start at the beginning of the last line again. To do this, we add an implementation for the `new_line` method of `Writer`:

```
// in src/vga_buffer.rs

impl Writer {
    fn new_line(&mut self) {
        for row in 1..BUFFER_HEIGHT {
            for col in 0..BUFFER_WIDTH {
                let character = self.buffer.chars[row][col].read();
                self.buffer.chars[row - 1][col].write(character);
            }
        }
        self.clear_row(BUFFER_HEIGHT - 1);
        self.column_position = 0;
    }

    fn clear_row(&mut self, row: usize) {/* TODO */}
}
```

We iterate over all screen characters and move each character one row up. Note that the range notation (`..`) is exclusive the upper bound. We also omit the 0th row (the first range starts at `1`) because it's the row that is shifted off screen.

To finish the newline code, we add the `clear_row` method:

```
// in src/vga_buffer.rs

impl Writer {
    fn clear_row(&mut self, row: usize) {
        let blank = ScreenChar {
            ascii_character: b' ',
            color_code: self.color_code,
        };
        for col in 0..BUFFER_WIDTH {
            self.buffer.chars[row][col].write(blank);
        }
    }
}
```

This method clears a row by overwriting all of its characters with a space character.

A Global Interface

To provide a global writer that can be used as an interface from other modules without carrying a `Writer` instance around, we try to create a static `WRITER`:

```rust
// in src/vga_buffer.rs

pub static WRITER: Writer = Writer {
    column_position: 0,
    color_code: ColorCode::new(Color::Yellow, Color::Black),
    buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
};
```

However, if we try to compile it now, the following errors occur:

```
error[E0015]: calls in statics are limited to constant functions, tuple structs and
 --> src/vga_buffer.rs:7:17
  |
7 |     color_code: ColorCode::new(Color::Yellow, Color::Black),
  |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

error[E0396]: raw pointers cannot be dereferenced in statics
 --> src/vga_buffer.rs:8:22
  |
8 |     buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ dereference of raw pointer

error[E0017]: references in statics may only refer to immutable values
 --> src/vga_buffer.rs:8:22
  |
8 |     buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ statics require immutable v

error[E0017]: references in statics may only refer to immutable values
 --> src/vga_buffer.rs:8:13
  |
8 |     buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
  |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ statics require immutable
```

To understand what's happening here, we need to know that statics are initialized at compile time, in contrast to normal variables that are initialized at run time. The component of the Rust compiler that evaluates such initialization expressions is called the “const evaluator”. Its functionality is still limited, but there is ongoing work to expand it, for example in the “Allow panicking in constants” RFC.

The issue about `ColorCode::new` would be solvable by using `const` functions, but the fundamental problem here is that Rust's const evaluator is not able to convert raw pointers to references at compile time. Maybe it will work someday, but until then, we have to find another solution.

Lazy Statics

The one-time initialization of statics with non-const functions is a common problem in Rust. Fortunately, there already exists a good solution in a crate named lazy_static. This crate provides a `lazy_static!` macro that defines a lazily initialized `static`. Instead of computing its value at compile time, the `static` laziliy initializes itself when it's accessed the first time. Thus, the initialization happens at runtime so that arbitrarily complex initialization code is possible.

Let's add the `lazy_static` crate to our project:

```
# in Cargo.toml

[dependencies.lazy_static]
version = "1.0"
features = ["spin_no_std"]
```

We need the `spin_no_std` feature, since we don't link the standard library.

With `lazy_static`, we can define our static `WRITER` without problems:

```
// in src/vga_buffer.rs

use lazy_static::lazy_static;

lazy_static! {
    pub static ref WRITER: Writer = Writer {
        column_position: 0,
        color_code: ColorCode::new(Color::Yellow, Color::Black),
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    };
}
```

However, this `WRITER` is pretty useless since it is immutable. This means that we can't write anything to it (since all the write methods take `&mut self`). One possible solution would be to use a mutable static. But then every read and write to it would be unsafe since it could easily introduce data races and other bad things. Using `static mut` is highly discouraged, there were even proposals to remove it. But what are the alternatives? We could try to use a immutable static with a cell type like RefCell or even UnsafeCell that provides interior mutability. But these types aren't Sync (with good reason), so we can't use them in statics.

Spinlocks

To get synchronized interior mutability, users of the standard library can use Mutex. It provides mutual exclusion by blocking threads when the resource is already locked. But our basic kernel does not have any blocking support or even a concept of threads, so we can't use it either. However there is a really basic kind of mutex in computer science that requires no operating system features: the spinlock. Instead of blocking, the threads simply try to lock it again and again in a tight loop and thus burn CPU time until the mutex is free again.

To use a spinning mutex, we can add the spin crate as a dependency:

```
# in Cargo.toml
[dependencies]
spin = "0.5.2"
```

Then we can use the spinning Mutex to add safe interior mutability to our static `WRITER` :

```
// in src/vga_buffer.rs

use spin::Mutex;
...
lazy_static! {
    pub static ref WRITER: Mutex<Writer> = Mutex::new(Writer {
        column_position: 0,
        color_code: ColorCode::new(Color::Yellow, Color::Black),
        buffer: unsafe { &mut *(0xb8000 as *mut Buffer) },
    });
}
```

Now we can delete the `print_something` function and print directly from our `_start` function:

```
// in src/main.rs
#[no_mangle]
pub extern "C" fn _start() -> ! {
    use core::fmt::Write;
    vga_buffer::WRITER.lock().write_str("Hello again").unwrap();
    write!(vga_buffer::WRITER.lock(), ", some numbers: {} {}", 42, 1.337).unwrap();


    loop {}
}
```

We need to import the `fmt::Write` trait in order to be able to use its functions.

Safety

Note that we only have a single unsafe block in our code, which is needed to create a `Buffer` reference pointing to `0xb8000` . Afterwards, all operations are safe. Rust uses bounds checking for array accesses by default, so we can't accidentally write outside the buffer. Thus, we encoded the required conditions in the type system and are able to provide a safe interface to the outside.

A println Macro

Now that we have a global writer, we can add a `println` macro that can be used from anywhere in the codebase. Rust's macro syntax is a bit strange, so we won't try to write a macro from scratch. Instead we look at the source of the `println!` macro in the standard library:

```
#[macro_export]
macro_rules! println {
    () => (print!("\n"));
    ($($arg:tt)*) => (print!("{}\n", format_args!($($arg)*)));
}
```

Macros are defined through one or more rules, which are similar to `match` arms. The `println` macro has two rules: The first rule for is invocations without arguments (e.g `println!()`), which is expanded to `print!("\n")` and thus just prints a newline. the second rule is for invocations with parameters such as `println!("Hello")` or `println!("Number: {}", 4)` . It is also expanded to an invocation of the `print!` macro, passing all arguments and an additional newline `\n` at the end.

The `#[macro_export]` attribute makes the macro available to the whole crate (not just the module it is defined) and external crates. It also places the macro at the crate root, which means that we have to import the macro through `use std::println` instead of `std::macros::println` .

The `print!` macro is defined as:

```
#[macro_export]
macro_rules! print {
    ($($arg:tt)*) => ($crate::io::_print(format_args!($($arg)*)));
}
```

The macro expands to a call of the `_print` function in the `io` module. The `$crate` variable ensures that the macro also works from outside the `std` crate by expanding to `std` when it's used in other crates.

The `format_args` macro builds a fmt::Arguments type from the passed arguments, which is passed to `_print` . The `_print` function of libstd calls `print_to` , which is rather complicated because it supports different `Stdout` devices. We don't need that complexity since we just want to print to the VGA buffer.

To print to the VGA buffer, we just copy the `println!` and `print!` macros, but modify them to use our own `_print` function:

```
// in src/vga_buffer.rs

#[macro_export]
```

```
macro_rules! print {
    ($($arg:tt)*) => ($crate::vga_buffer::_print(format_args!($($arg)*)));
}

#[macro_export]
macro_rules! println {
    () => ($crate::print!("\n"));
    ($($arg:tt)*) => ($crate::print!("{}\n", format_args!($($arg)*)));
}

#[doc(hidden)]
pub fn _print(args: fmt::Arguments) {
    use core::fmt::Write;
    WRITER.lock().write_fmt(args).unwrap();
}
```

One thing that we changed from the original `println` definition is that we prefixed the invocations of the `print!` macro with `$crate` too. This ensures that we don't need to have to import the `print!` macro too if we only want to use `println`.

Like in the standard library, we add the `#[macro_export]` attribute to both macros to make them available everywhere in our crate. Note that this places the macros in the root namespace of the crate, so importing them via `use crate::vga_buffer::println` does not work. Instead, we have to do `use crate::println`.

The `_print` function locks our static `WRITER` and calls the `write_fmt` method on it. This method is from the `Write` trait, we need to import that trait. The additional `unwrap()` at the end panics if printing isn't successful. But since we always return `Ok` in `write_str`, that should not happen.

Since the macros need to be able to call `_print` from outside of the module, the function needs to be public. However, since we consider this a private implementation detail, we add the `doc(hidden)` attribute to hide it from the generated documentation.

Hello World using `println`

Now we can use `println` in our `_start` function:

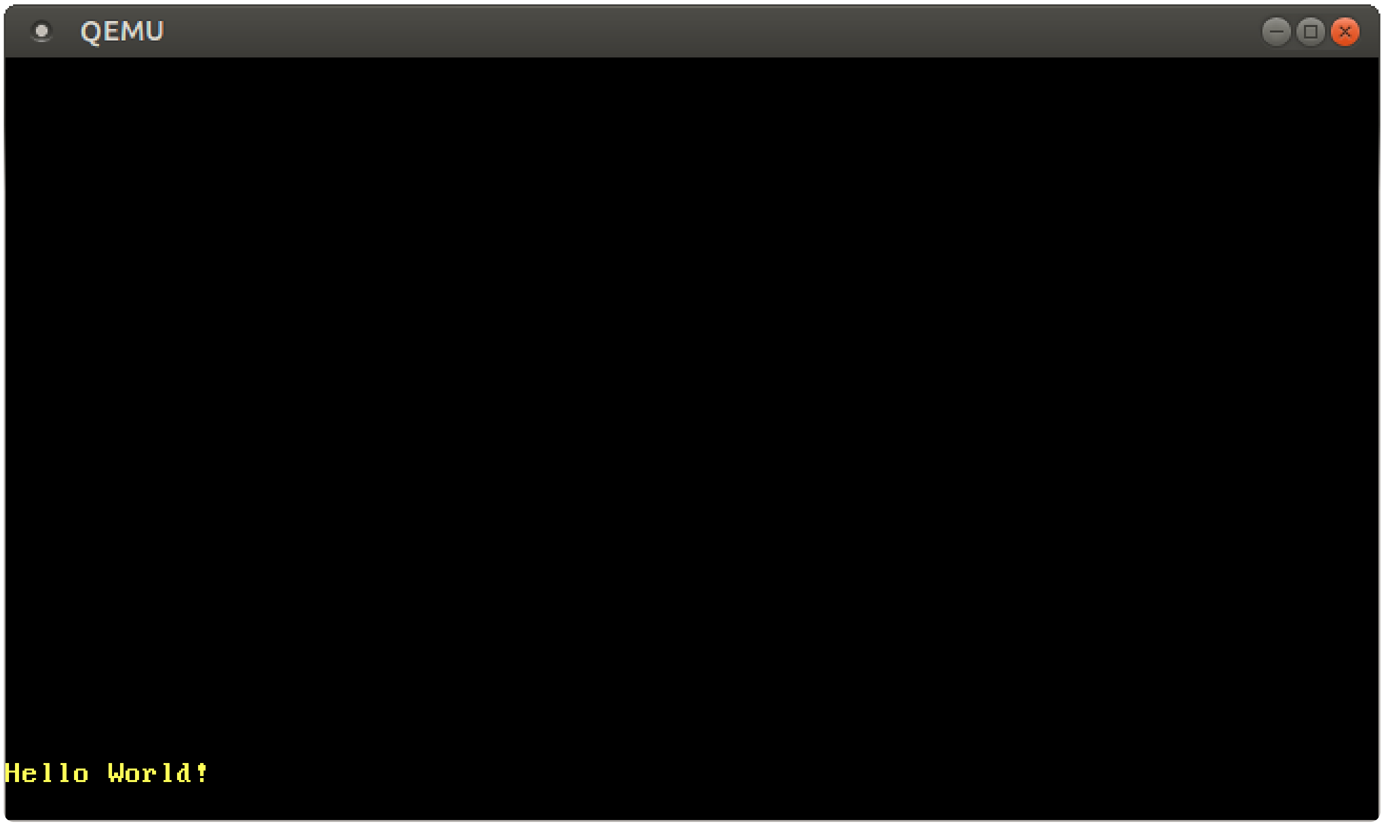
```
// in src/main.rs

#[no_mangle]
pub extern "C" fn _start() {
    println!("Hello World{}", "!");

    loop {}
}
```

Note that we don't have to import the macro in the main function, because it already lives in the root namespace.

As expected, we now see a *"Hello World!"* on the screen:



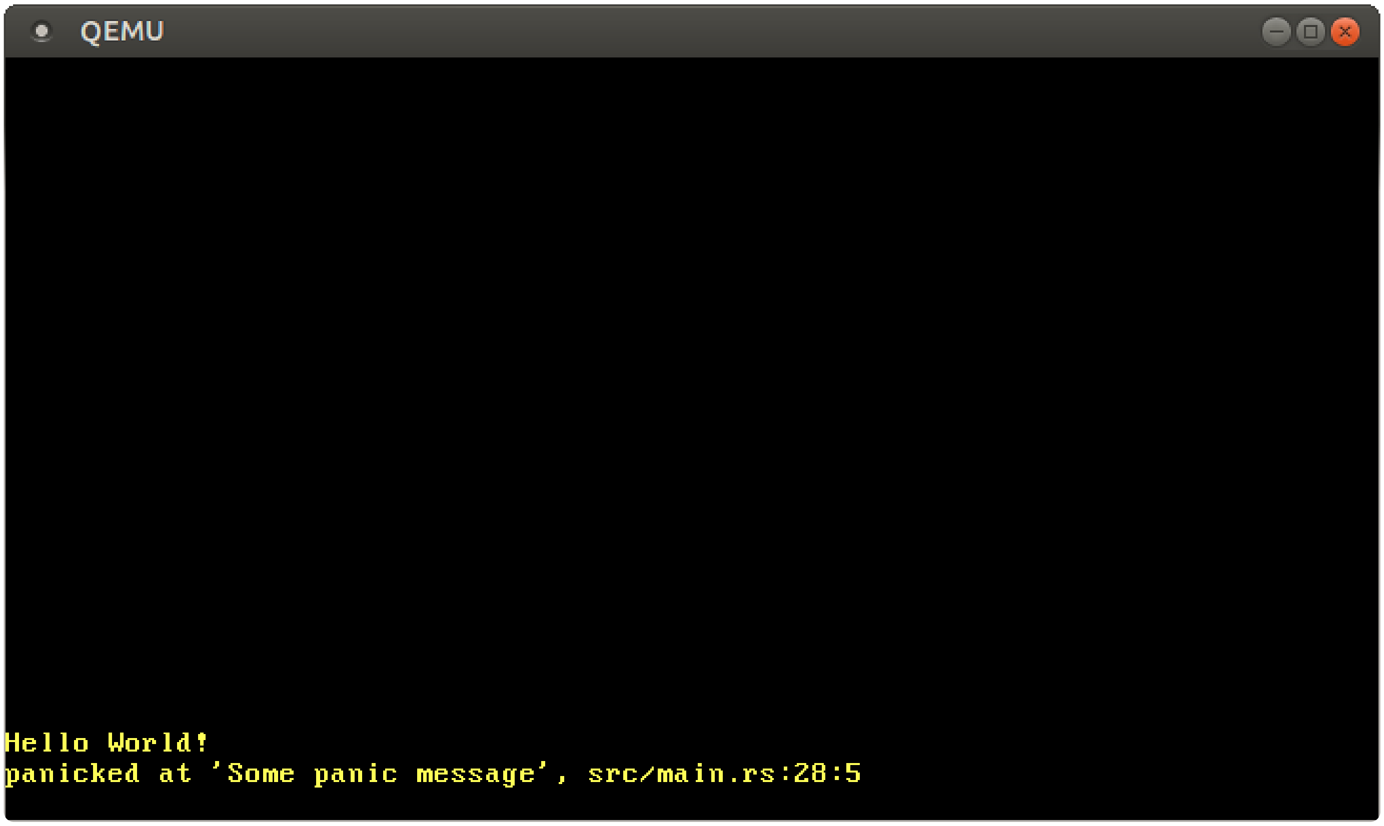
Printing Panic Messages

Now that we have a `println` macro, we can use it in our panic function to print the panic message and the location of the panic:

```
// in main.rs

/// This function is called on panic.
#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    println!("{}", info);
    loop {}
}
```

When we now insert `panic!("Some panic message");` in our `_start` function, we get the following output:



So we know not only that a panic has occurred, but also the panic message and where in the code it happened.

Summary

In this post we learned about the structure of the VGA text buffer and how it can be written through the memory mapping at address `0xb8000`. We created a Rust module that encapsulates the unsafety of writing to this memory mapped buffer and presents a safe and convenient interface to the outside.

We also saw how easy it is to add dependencies on third-party libraries, thanks to cargo. The two dependencies that we added, `lazy_static` and `spin`, are very useful in OS development and we will use them in more places in future posts.

What's next?

The next post explains how to set up Rust's built in unit test framework. We will then create some basic unit tests for the VGA buffer module from this post.

Support Me

Creating and maintaining this blog and the associated libraries is a lot of work, but I really enjoy doing it. By supporting me, you allow me to invest more time in new content, new features, and continuous maintenance.

The best way to support me is to *sponsor me on GitHub*, since they don't charge any fees. If you prefer other platforms, I also have *Patreon* and *Donorbox* accounts. The latter is the most flexible as it supports multiple currencies and one-time contributions.

Thank you!

« A Minimal Rust Kernel Testing »

0 reactions

138 comments – *powered by giscus*

1

gurpreetshanky Mar 10, 2018

Excellent Tutorial series.. As a Rust newbie its a great resource to learn about new things.

👍 8

0 replies

1

danrien Mar 10, 2018

Wow this was all sorts of fun... thanks for putting this together. I've both learned a lot about Rust (in a far gentler fashion than normal) and learned what's involved in building an OS. I look forward to future articles!

By the way, for whatever reason, when I call my implementation of `write_str`, I don't have to call `.unwrap()`, yet I need to call it from the macro... either I did something wrong or there's some difference I have yet to find with the tutorial and my code. Here's my `write_str` implementation:

```
impl fmt::Write for Writer {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for byte in s.bytes() {
            self.write_byte(byte)
        }
        Ok(())
    }
}
```

0 replies

1

drgmr Mar 11, 2018

@danrien I had the same problem - make sure your `impl Writer` doesn't have a `write_str` method too - you might be using that instead of the one in the `fmt::Write` implementation.

0 replies

1

danrien Mar 11, 2018 edited

Ah thanks, that did the trick!

0 replies

1

phil-opp Mar 13, 2018 Owner edited

@gurpreetshanky **@danrien** Thanks a lot! Glad that you find the posts informative :).

@danrien **@edupc** I just pushed #417 to avoid this confusion. Thanks for the feedback!

0 replies

0

SomeAnotherDude Mar 26, 2018 Contributor

I think it should be useful to add an empty branch to println! macro. (Standard macro has this branch).

```
macro_rules! println {
    () => (print!("\n"));
    ($fmt:expr) => (print!(concat!($fmt, "\n")));
    ($fmt:expr, $($arg:tt)*) => (print!(concat!($fmt, "\n"), $($arg)*));
}
```

👍 1

0 replies

1

fpsiorz Apr 1, 2018

I don't think printing strings byte by byte is good in this case. The characters on screen are encoded in Code Page 437, which is identical to UTF-8 (which Rust strings use) only for a encoded in Code Page 437, which is identical to UTF-8 (which Rust strings use) only for a certain range of characters.

The `write_byte` currently prints symbols for all control characters except '◙' (which is on the code point of '\n'). I think it should do something like this instead:

```
pub fn write_byte(&mut self, byte: u8) {
    match byte {
        b'\n' => self.new_line(),
        b' '...b'~' => {
            // keep current behavior
            ...
        },
        _ => {
            // handle non-ASCII characters
        },
    }
}
```

I added Unicode to Code Page 437 translation to my implementation, but that's a lot of code.

0 replies

1

phil-opp Apr 2, 2018 Owner

@fpsiorz Thanks for the suggestion! I opened #425 to add such a check in the `write_strings` method.

0 replies

1

arjanvaneersel Apr 30, 2018

At first I got this error:

```
error[E0658]: const fn is unstable (see issue #24111)
  --> src/vga_buffer.rs:33:5
   |
33 | /     const fn new(foreground: Color, background: Color) -> ColorCode {
34 | |         ColorCode((background as u8) << 4 | (foreground as u8))
35 | |     }
   | |_____^
   |
   = help: add #![feature(const_fn)] to the crate attributes to enable
```

The reason is obvious: I forgot to add `#![feature(const_fn)]` to main.rs, but it took me a while to figure that out and at first I simply removed the "const" at the function declaration. The code executed fine without the const.

So I wonder: What are const functions exactly for and why do we need to use a const function here instead of a normal function? I found the (RFC)[https://github.com/rust-lang/rfcs/blob/master/text/0911-const-fn.md] on const functions, but that doesn't make me understand the concept behind const functions.

0 replies

1

phil-opp May 1, 2018 Owner

@arjanvaneersel A `const` feature behaves exactly the same as a normal function, but has one additional feature: It can be used for initializing const values, such as a `static` or an array length. In our case, the idea was that it can be used to initialize the static `WRITER`. However, we later use `lazy_static` for initializing it, so the `const` really isn't needed here.

0 replies

1

phil-opp May 1, 2018 Owner

@arjanvaneersel I removed the `const` from the function and added a short explanation of const evaluation in `ba266f3`.

0 replies

1

bugabinga May 10, 2018

Thank you! This is a lot of fun :).

Typical beginner mistakes, that compile, but lead to blank screen:
(these are all errors I made)

- using vga buffer address `0x8000` instead of `0xb8000` (missing 'b')
- in `new_line` putting the `clear_row` call inside the outer for-loop

> small "debugging" tip: put some character into the `blank.ascii_character` field in `clear_row` to see what is actually happening. I chose `b'_'.

Some notes from a beginner perspective:

- I do not understand what is happening in the `unsafe` block. At all. Is this by any chance explained somewhere? I am assuming that it is a well know pattern...?
- The `unwrap` seems suspicious. If it "should" (read: hopefully,maybe) never occur, why is it there? Is there not even one plausible scenario?

Very kool stuff!

If someone else likes to use *cargo-make*, like I do, here is a usable `Makefile.toml`:

```
[tasks.build]
description = "Build no-std rust binary. Combine it with a preconfigured bootloa
command = "bootimage"
args = ["build"]

[tasks.boot]
description="Boot the kernel in a qemu virtual machine."
command="bootimage"
args=["run"]
```

```
# build 'bootimage.bin' via the bootimage tool
$ cargo make build
# run 'bootimage.bin' in qemu
$ cargo make boot
```

👍 1

0 replies

1

phil-opp May 11, 2018 Owner edited

@bugabinga Thanks for your comment!

> I do not understand what is happening in the unsafe block.

You mean `&mut *(0xb8000 as *mut Buffer)`? From the post:

> The syntax for this might seem a bit strange: First, we cast the integer 0xb8000 as an mutable raw pointer. Then we convert it to a mutable reference by dereferencing it (through *) and immediately borrowing it again (through &mut). This conversion requires an unsafe block, since the compiler can't guarantee that the raw pointer is valid.

Basically it's a way of transforming an integer to a memory reference. In C, the equivalent would be the cast `(Buffer *) 0xb8000`.

> The unwrap seems suspicious. If it "should" (read: hopefully,maybe) never occur, why is it there? Is there not even one plausible scenario?

The problem is that the fmt::Write trait is defined that way that `write_str` returns a `Result`. The reason for this is that some implementations (e.g. for Linux stdout) might return an error. In our implementation, we never return an error, so the unwrap is safe.

0 replies

1

cedws Oct 22, 2018 Contributor edited

Hey, thanks for this excellent blog series. I noticed while following along a small adjustment that could be made. Perhaps it was done like this for clarity, but I think the blog series assumes a familiarity with Rust.

```diff
 let color_code = self.color_code;
 self.buffer.chars[row][col] = ScreenChar {
     ascii_character: byte,
-    color_code: color_code,
+    color_code,
 };
 self.column_position += 1;
```

I was also prompted by the latest nightly compiler to add `#![feature(exclusive_range_pattern)]` due to this chunk of code:

```
match byte {
    0x20..0x7e | b'\n' => self.write_byte(byte),
    _ => self.write_byte(0xfe),
}
```

I recommend changing it to look like this, so that `0x7e` (which is `~`) is included in the pattern. This also means that the feature gate I mentioned above won't be required.

```diff
- 0x20..0x7e
+ 0x20..=0x7e
```

0 replies

1

phil-opp Oct 22, 2018 Owner

@c-edw Thanks, I agree with both suggestions! Could you open pull request that applies these changes to both src/vga_buffer.rs and blog/content/second-edition/posts/03-vga-text-buffer/index.md?

0 replies

108 hidden items
Load more...

1

bjorn3 Sep 15 Contributor

`cargo new` should automatically add `edition = "2018"` cargo new should automatically add edition = "2018".

0 replies

1

martin-bucinskas Oct 5

Might be worth to update the guide as the latest `volatile = "0.4.1"` version will throw an error when compiling, complaining with the following error:

```
error[E0599]: no method named `write` found for struct `Volatile<ScreenChar>` i
  --> src/vga_buffer.rs:98:45
   |
66 | struct ScreenChar {
   | -----------------
   | |
   | doesn't satisfy `<ScreenChar as __Deref>::Target = _`
   | doesn't satisfy `ScreenChar: __Deref`
...
98 |                 self.buffer.chars[row][col].write(ScreenChar {
   |                                             ^^^^^ method not found in `Vola
   |
   = note: the method `write` exists but the following trait bounds were not sat
           `<ScreenChar as __Deref>::Target = _`
           `ScreenChar: __Deref`
```

Might be worth adding the `Deref` and `DerefMut` implementations as they are part of `core::ops` and not standard.

E.g.

```
use core::ops::{Deref, DerefMut};
...
impl Deref for ScreenChar {
    type Target = ScreenChar;

    fn deref(&self) -> &Self::Target {
        self // TODO: Implement
    }
}

impl DerefMut for ScreenChar {
    fn deref_mut(&mut self) -> &mut Self::Target {
        self // TODO: Implement
    }
}
```

👍 2

0 replies

1

phil-opp Oct 5 Owner

@martin-bucinskas The `volatile` crate was completely rewritten in the latest version, so it won't work with the post. Just stick to `volatile = "0.2.6"` for now, then everything should work.

I'm currently working on a completely new version of this post, which will use a pixel-based framebuffer instead of the VGA text mode. I will use the new `volatile` release for this upcoming post.

🎉 1 ❤️ 9

0 replies

1

nurmohammed840 Jan 27

Sorry for this dumb question...

```
// But what does this mean ?
0xb8000 as *mut Buffer
```

It make sense that `0xb8000 as u8` , But `Buffer` !?
New to rust...

0 replies

1

bjorn3 Jan 27 Contributor

This is casting the address `0xb800` to the pointer `*mut Buffer` .

0 replies

1

codic12 Feb 12

`0xb8000` - a hexadecimal literal. it is the start of VGA text mode memory (0xA0000 is for the raw framebuffer access).
`as` - indicates a cast, which tells the compiler to treat the left hand side as if it's type is the type given on the right hand side
`*mut Buffer` - a mutable pointer to the type `Buffer` . you can then use this buffer as memory to write to, like an array, because it is a pointer; same concept as `(buffer_t *)0xb8000` assuming buffer_t is defined.

0 replies

1

ssrlive Feb 19

Compiling error!
In https://os.phil-opp.com/vga-text-mode/#formatting-macros section. I have to stop by the error of `method write_fmt not found for this`, how to resolve it?

0 replies

1

phil-opp Feb 19 Owner

Did you add the `use core::fmt::Write;` import?

0 replies

1

ssrlive Feb 19

Fixed. Just add a line `use core::fmt::Write;`.

0 replies

1

ssrlive Feb 19

Please modify the article to fix the issue.

> Fixed. Just add a line `use core::fmt::Write;`

0 replies

1

drzewiec Feb 20

@ssrlive The article already says that.

0 replies

1

chrisbroome Apr 27

Is there a setting to change/reason why all the test executables get recompiled every time I run `cargo test` even if no code actually changes?

0 replies

0 replies

1

phil-opp Apr 30 Owner

@chrisbroome The `bootloader` needs to be re-linked whenever the kernel executable changes because it currently directly links it. This is something that we plan to fix in the future by making the bootloader load the kernel from disk separately.

👍 1

0 replies

1

maximxlss May 3

I don't really understand why Writer doesn't own the Buffer, but only takes a mutable reference.

0 replies

1

bjorn3 May 3 Contributor

`Buffer` is at a fixed location in memory defined by the hardware/firmware. `Writer` can be stored anywhere and as such can't have `Buffer` as field without indirection. Using `Box` also doesn't make sense as `Buffer` is not actually owned by `Writer`, but by the hardware/firmware.

0 replies

Write Preview

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub

Instead of authenticating the giscus application, you can also comment directly on the on GitHub. Just click the *"X comments"* link at the top — or the date of any comment — to go to the GitHub discussion.

Other Languages

- Persian
- Japanese
- Chinese (simplified)

About Me

I'm a Rust freelancer with a master's degree in computer science. I love systems programming, open source software, and new challenges.

If you want to work with me, reach out on LinkedIn or write me at job@phil-opp.com.
