# Asynchronous I/O in Rust

[[](https://medium.com/@technocrat21?source=post_page-----8096b5a58c1a--------------------------------)](https://medium.com/@technocrat21?source=post_page-----8096b5a58c1a--------------------------------)

[[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----8096b5a58c1a--------------------------------)](https://medium.com/coderhack-com?source=post_page-----8096b5a58c1a--------------------------------)

[Amay B](https://medium.com/@technocrat21?source=post_page-----8096b5a58c1a--------------------------------)

·

Follow

Published in

[CoderHack.com](https://medium.com/coderhack-com?source=post_page-----8096b5a58c1a--------------------------------)

·

3 min read

·

Sep 15

25

1

Rust has excellent support for asynchronous I/O through features like Futures and the async/await syntax. In this article, we’ll explore how to write asynchronous Rust code using these capabilities.



Photo by [Emile Guillemot](https://unsplash.com/@emilegt?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

## The futures crate

The [futures](https://docs.rs/futures/) crate provides the Future trait, the basic building block for asynchronous tasks in Rust. A future represents an asynchronous computation that may complete at some point in the future.

We can define a basic future that returns a value asynchronously like this:

use futures::Future;  
  
struct MyFuture {  
 // ...  
}  
  
impl Future for MyFuture {  
 type Output = i32;  
  
 fn poll(self: Pin<&mut Self>, cx: &mut Context<'\_>) -> Poll<Self::Output> {  
 Poll::Ready(5)  
 }  
}

We can chain multiple futures together using combinators like .and\_then():

let future = MyFuture.and\_then(|x| async move { x + 1 });

This will yield a new future that returns x + 1 once MyFuture completes.

## The tokio crate

The [tokio](https://tokio.rs/) crate provides an asynchronous runtime for executing futures. We can spawn a future onto the tokio runtime using tokio::spawn():

tokio::spawn(my\_future);

This will run the future concurrently in the background.

We can also run a future to completion synchronously within the tokio::run\_main runtime:

tokio::run\_main(my\_future);

## Asynchronous I/O with async/await

The async/await syntax provides a clean way to write asynchronous Rust code. We can define an async function like this:

async fn my\_async\_function() {  
 // ...  
}

Inside an async function, we can call .await on asynchronous values:

async fn my\_async\_function() {  
 let value = other\_future().await;  
 // ...  
}

This will pause the async function and resume it once other\_future completes, binding its return value to value.

We can call asynchronous code from synchronous code using runtime::spawn\_blocking():

tokio::spawn\_blocking(|| {  
 my\_async\_function();  
});

This will spawn my\_async\_function onto a dedicated thread pool and return a future representing its completion.

## Working with asynchronous I/O objects

The AsyncRead and AsyncWrite traits abstract over asynchronous I/O. For example, TcpStream is a wrapper around a TCP socket that implements these traits. We can read from and write to a TcpStream asynchronously like this:

use tokio::io::{AsyncReadExt, AsyncWriteExt};  
use tokio::net::TcpStream;  
  
let stream = TcpStream::connect("127.0.0.1:7891").await?;  
let mut buf = [0; 1024];  
stream.read(&mut buf).await?; // Read from the socket  
stream.write(&buf).await?; // Write to the socket

We can also use these traits to read from and write to files asynchronously.

## Using channels for asynchronous communication

Channels are useful for asynchronous communication between tasks. We can create an unbounded channel using tokio::sync::mpsc::unbounded\_channel():

let (tx, rx) = tokio::sync::mpsc::unbounded\_channel();

We can then send values on the tx handle from one task:

tx.send(42);

And receive them on the rx handle from another task:

let value = rx.recv().await; // 42

## Testing asynchronous code

We can use the tokio\_test crate to synchronously run asynchronous code in tests. For example:

#[tokio::test]  
async fn my\_async\_test() {  
 // Async test logic here  
}

This will run the test asynchronously and complete synchronously.

We can also use the #[tokio::test] attribute:

#[tokio::test]   
fn my\_async\_test() {  
 tokio::runtime::Runtime::new().unwrap().block\_on(async {  
 // Async test logic here   
 });  
}

This will run the async closure on the tokio runtime, blocking until it completes.

## Conclusion

Rust provides excellent support for asynchronous programming through crates like futures and tokio, as well as language features like async/await. In this article, we explored asynchronous I/O, futures, running futures with tokio, and more. Asynchronous Rust code powers many high performance and scalable systems and these capabilities allow Rust to be a leader in the fields of servers, network infrastructure, and more.