Learn Rust programming and hack the NanoPi Neo

Pramode C.E

25 September, 2017

A language that doesn't affect the way you think about programming, is not worth knowing.

`Alan Perlis.`

Why Rust?

> *"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety."*

– From www.rust-lang.org

Why Rust?

- A modern, memory-safe replacement for C/C++ with excellent tooling.
- Memory safety achieved using innovative type system concepts (affine types) and *not* using Garbage Collection.
- Many high level features (mostly borrowed from statically typed functional programming languages) without any run time overhead (so-called "zero cost abstractions").

Why Not Rust?

- Complex language with a steep learning curve. May not be suitable for beginners. (Rust community is aware of the problem . . . there are efforts to improve the language ergonomics and create high quality learning materials).

Why Not Rust?

- Difficult to express many data structure patterns in “safe” Rust.
- Very young . . . library ecosystem not as mature as that of older languages (but this will improve with time).
- Long compilation time (this will improve).

Why Not C/C++?

What comes to your mind when you think of C/C++?

Figure 1: Speed

Figure 2:

C - Undefined behaviours

Compile using clang: “clang -O3 undef1.c”.

```
// undef1.c
static void (*Do)();

static void EraseAll() {
  printf("remove all files ...\n");
}
void NeverCalled() {
  Do = EraseAll;
}
int main() {
  Do();
}
```

Ref: https://www.reddit.com/r/cpp/comments/6xeqr3/compiler_undefined_behavior_calls_nevercalled/

C - undefined behaviours

Compile with “gcc -O3 undef3.c”

```
// undef3.c
#include <limits.h>
#include <stdio.h>
main()
{
    int i = INT_MAX;

    if (i > i + 1) {
        printf("hello\n");
    }
}
```

C - undefined behaviours

- http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html
- http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html
- http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html

C - Memory safety issues

```c
// mem1.c
#include <string.h>
#include <stdio.h>

int main()
{
    char *c = strstr("hello", "mo");
    printf("%c\n", *c);
}
```

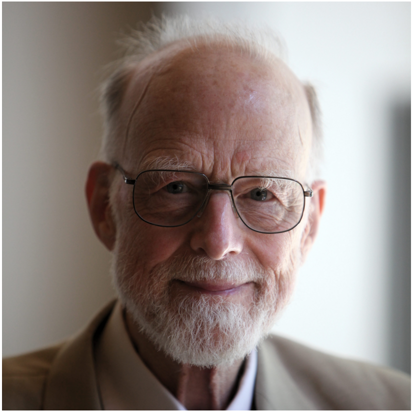

Figure 3:

The billion dollar mistake

> *I call it my billion-dollar mistake. It was the invention of the null reference in 1965 . . . . This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

– Tony Hoare, inventor of QuickSort

C - Memory safety issues

```c
// mem2.c
int* fun()
{
    int x = 1;
    return &x;
}
```

C - Memory safety issues

```c
// mem3.c
#include <string.h>
#include <stdio.h>
int main()
{
    char a[5];

    strcpy(a, "hello");
}
```

C - Memory safety issues

```c
// mem4.c
#include <stdlib.h>
void fun()
{
    int *p = malloc(10 * sizeof(int));
    /* use the block */
    /* forget to free */
}
```

C - Memory safety issues

```c
// mem6.c
#include <stdlib.h>
void fun(int *p)
{
    // use p

    free(p);
}
int main()
{
    int *p = malloc(10 * sizeof(int));
    fun(p);
    p[0] = 10;
}
```

C - Memory safety issues

```c
// mem7.c
#include <stdlib.h>
void fun(int *p)
{
    // do some stuff with p

    free(p);
}
int main()
{
    int *p = malloc(10 * sizeof(int));
    fun(p);
    free(p);
}
```

C - Memory safety issues

October 2, 2017: Yet more DNS and DHCP vulnerabilities: https://security.googleblog.com/2017/10/behind-masq-yet-more-dns-and-dhcp.html

Almost all of them are memory safety issues! Example:

- CVE-2017-14493 Stack based overflow
- CVE-2017-14495 Lack of "free()"
- CVE-2017-14492 Heap based overflow

C/C++ programming: what a beginner thinks it is

Figure 4:

C/C++ programming: what it really is!

Figure 5:

A bit of Rust history

- Started by Graydon Hoare as a personal project in 2006
- Mozilla foundation started sponsoring Rust in 2010
- Rust 1.0 released in May, 2015
- Regular six week release cycles

Firefox Quantum: Rust in action

Figure 6: Firefox Quantum

Core language features

- Memory safety without garbage collection
  - Ownership
  - Move Semantics
  - Borrowing and lifetimes
- Static Typing with Type Inference
- Algebraic Data Types (Sum and Product types)
- Exhaustive Pattern Matching
- Trait-based generics
- Iterators
- Zero Cost Abstractions
- Concurrency without data races
- Efficient C bindings, minimal runtime

Structure of this workshop

- Session 1: Understand basic Rust concepts
- Session 2: Programming an embedded Linux system (the NanoPi Neo) with Rust

Hello, world!

```
$ cargo new myproj --bin
$ cd myproj
$ cargo run
Hello, world!
$ cd src
$ cat main.rs
fn main() {
    println!("Hello, world!");
}
$ rustc main.rs
$ ./main
Hello, world!
$
```

Using external libraries

```
// folder: random
$ cd random
$ cat Cargo.toml
[package]
name = "random"
version = "0.1.0"
authors = ["Pramode <mail@pramode.in>"]

[dependencies]
rand = "0.3"
$ cargo run
...(a random number)
$
```

Integrated unit tests

```rust
// folder: tests
#[test]
fn sqr_0_test() {
    assert_eq!(sqr(0), 0);
}

fn sqr(x: i32) -> i32 {
    x * x
}
fn main() {
    println!("Hello, world!");
}
```

```
$ cargo test # runs the test functions
```

Static typing, type inference, Immutability

Exercise: make the code compile and run correctly!

```
// missing "mut"; also a missing line
// infer1.rs
fn factorial(n: i32) -> i32 {
    let f = 1;
    while n > 0 {
        f = f * n;
        // missing line
    }
    f
}
fn main() {
    let r = factorial(4);
    println!("{}", r);
}
```

Expression oriented programming

```rust
// expr1.rs
fn main() {
    let a = 1;
    let b = 2;

    let r = if a > b {
                a
            } else {
                b
            };

    println!("{}", r);
}
```

Expression oriented programming

```
Exercise: implement max3
//expr2.rs
fn max2(a: i32, b: i32) -> i32 {

    if a > b { a } else { b}
}

fn max3(a: i32, b: i32, c: i32) -> i32 {

    // define max3 using max2
}

fn main() {
    println!("{}", max3(10, 12, 8));
}
```

Expression oriented programming

Even loops are expressions!

```
// expr3.rs
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("{}", result);
}
```

Generic data structures: Vectors

Exercise: make the “push” operation work.

```
// vec1.rs
fn main() {
    let v1:Vec<i32> = vec![1,2,3];
    let mut v2 = vec![1.2, 2.3, 4.5];
    let v3 = vec!["cpp", "python", "perl"];

    v2.push(5);

    println!("v1 = {:?}, v2 = {:?}, v3 = {:?}",
             v1, v2, v3);

}
```

Pop an element from a vector

Why does this not compile?

```
// vec2.rs
fn main() {
    let mut v1 = vec![1,2,3];
    let x = v1.pop() + 10;
    println!("{}", x);

}
```

Pop an element from a vector

- Popping an element is a tricky operation
- What do you do if the vector is empty?
- Python/Java etc: Raise an exception
- Rust: Use special types: Option / Result

Using the Option type

```rust
// vec3.rs
fn main() {
    let mut v1 = vec![1,2,3];
    let mut v2:Vec<i32> = vec![];

    let x1 = v1.pop();
    let x2 = v2.pop();
    println!("x1={:?}, x2={:?}", x1, x2);

}
```

Using the Option type

The Option type can assume two values:

- None.
- Some(x) where x is a value of some type T, say for example i32.

Some(5), Some("hello"), Some(1.2) etc are all possible values for a variable of type Option.

Option type and pattern matching

```rust
// option1.rs
fn main() {
    let v1:Option<i32> = None;
    let v2:Option<i32> = Some(10);

    match v1 {
        None => println!("Option value is None"),
        Some(x) => println!("x = {}", x),
    }
    match v2 {
        None => println!("Option value is None"),
        Some(x) => println!("x = {}", x),
    }
}
```

Unwrapping an Option

```rust
// option2.rs
fn main() {
    let v1:Option<i32> = None;
    let v2:Option<i32> = Some(10);

    println!("v2: x={}", v2.unwrap());
    println!("v1: x={}", v1.unwrap());
}
```

Note: Calling unwrap in production code is *not* a good idea.

Popping from a vector: continued

Exercise: The program should print the popped value; it should print 'stack empty' if the stack is empty.

```
// option3.rs
fn main() {
    let mut v = vec![1,2,3];

    let x = v.pop();
    // fill up the missing parts
    match  {
         => println!("stack empty"),
        Some() => println!("{}", ),
    }
}
```

Creating Option-like types: using enums

There is nothing special about the Option type. You can create your own “option-like” types using Rust “enums”.

Using enums

- An Option can assume one of two values, either None or Some(x).
- A Color can assume one of three values: Red, Green, Blue
- A shape can assume one of three values, Circle(r), Square(x), Rectangle(x, y).
- Here, r represents radius, x represents a side of the Square and (x,y) represents two sides of a Rectangle.

This "exactly-one-of-many" pattern is *very* common in programming. Rust "enums" are used to represent this pattern in code.

Using enums

```rust
// enum1.rs
enum Color {
    Red,
    Green,
    Blue,
}
use Color::*;
fn main() {
    let c1 = Green;
    // you can also write let c1:Color = Green
    // Bug here. Fix it!
    match c1 {
        Red    => println!("Red ..."),
        Green  => println!("Green ..."),
    }
}
```

Using enums

```rust
// enum2.rs
#[derive(Debug)]
enum Shape {
    Circle(i32),
    Square(i32),
    Rectangle(i32, i32),
}
use Shape::*;
fn main() {
    let s = Rectangle(10, 20);
    println!("{:?}", s);
}
```

Using enums

```
// enum3.rs
// complete this function
fn area(s: Shape) -> f32 {
    match s {
        Circle(r) => 3.14 *  ,
        Square(x) => ,
        Rectangle() => ,
    }
}
```

An implementation of Option

```rust
// enum4.rs
#[derive(Debug)]
enum MyOption <T>{
    MyNone,
    MySome(T),
}
use MyOption::*;
fn main() {
    let x = MySome(10);
    let y = MySome("hello");
    let z:MyOption<i32> = MyNone;

    println!("x={:?},y={:?},z={:?}", x, y, z);
}
```

The core of Rust

Time to look at the core ideas:

- Ownership and Move semantics
- Borrowing
- Lifetime

Scope

```rust
// scope1.rs
fn main() {
    let x = 10;
    {
        let y = 20;
    }
    println!("x={}, y={}", x, y);
}
```

Ownership

```rust
// owner1.rs
fn main() {
    let v = vec![10 ,20, 30];

    println!("{:?}", v);
    // how is v deallocated?
}
```

Ownership

Memory representation of a Vector

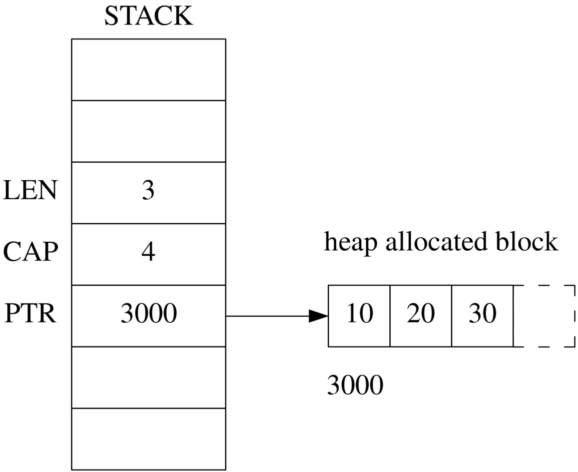

Figure 7: Memory representation of a vector

Ownership

```rust
// owner2.rs
fn fun1() {
    let v = vec![10 ,20 ,30];
} // how is v deallocated?
fn main() {
    fun1();
}
```

Ownership

```rust
// owner3.rs
fn main() {
    let  v1 = vec![10 ,20 ,30];
    let  v2 = v1;
    println!("{:?}", v2);
}
// do we have a double free here?
```

Ownership

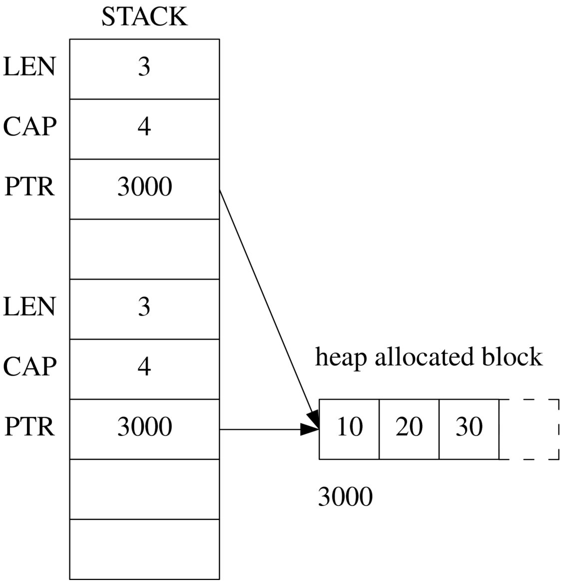

Figure 8: Two pointers

Ownership

```rust
// owner4.rs
fn fun(v2: Vec<i32>)  {
    println!("{:?}", v2);
}
fn main() {
    let  v1 = vec![10, 20, 30];
    fun(v1);
}
// do we have a double free here?
```

Ownership

```rust
// owner5.rs
fn main() {
    let  v1 = vec![10, 20, 30];
    let mut v2 = v1;
    v2.truncate(2);
    println!("{:?}", v2);
}
// what happens if we try to acces the
// vector through v1?
```

Move semantics

```rust
// owner6.rs
fn main() {
    let  v1 = vec![1,2,3];

    let mut v2 = v1;
    v2.truncate(2);
    println!("{:?}", v1);
}
```

Move semantics

```rust
// owner7.rs
fn fun(v2: Vec<i32>)  {
    println!("{:?}", v2);
}
fn main() {
    let  v1 = vec![10, 20, 30];
    fun(v1);
    println!("{:?}", v1);
}
```

Move semantics

```rust
// owner8.rs
fn main() {
    let a = (1, 2.3);
    let b = a;
    println!("{:?}", a);
}
```

Move semantics

```
// owner9.rs
fn main() {
    let a = (1, 2.3, vec![10,20]);
    let b = a;
    println!("{:?}", a);
}
```

Ownership / Move: Limitations

```
// owner10.rs
fn vector_sum(v: Vec<i32>) -> i32 {
    //assume v is always a 3 elemnt vector
    v[0] + v[1] + v[2]
}
fn main() {
    let v = vec![1,2,3];
    let s = vector_sum(v);
    println!("{}",s);
}
```

Ownership / Move: Limitations

```rust
// owner11.rs
fn vector_sum(v: Vec<i32>) -> i32 {
    v[0] + v[1] + v[2]
}
fn vector_product(v: Vec<i32>) -> i32 {
    v[0] * v[1] * v[2]
}
fn main() {
    let v = vec![1,2,3];
    let s = vector_sum(v);
    let p = vector_product(v);
    println!("{}",p);
}
// does this code compile?
```

Immutable Borrow

```rust
// borrow1.rs
fn vector_sum(v: &Vec<i32>) -> i32 {
    v[0] + v[1] + v[2]
}
fn vector_product(v: &Vec<i32>) -> i32 {
    v[0] * v[1] * v[2]
}
fn main() {
    let  v = vec![1,2,3];
    let s = vector_sum(&v);
    let p = vector_product(&v);
    println!("v={:?}, s={}, p={}", v, s, p);
}
```

Immutable Borrow

```rust
// borrow2.rs
fn main() {
    let v = vec![1,2,3];
    let t1 = &v;
    let t2 = &v;
    println!("{}, {}, {}", t1[0], t2[0], v[0]);
}
// any number of immutable borrows are ok!
```

Immutable Borrow

```
// borrow3.rs
fn change(t1: &Vec<i32>) {
    t1[0] = 10;
}
fn main() {
    let mut v = vec![1,2,3];
    change(&v);
}
// Does the program compile?
```

Mutable Borrow

```rust
// borrow4.rs
fn change(t1: &mut Vec<i32>) {
    t1[0] = 10;
}
fn main() {
    let mut v = vec![1,2,3];
    change(&mut v);
    println!("{:?}", v);
}
```

Mutable and immutable borrow

```rust
// borrow5.rs
fn change(t1: &mut Vec<i32>) {
    t1[0] = 10;
}
fn main() {
    let mut v = vec![1,2,3];
    let p = &v; // an immutable borrow
    change(&mut v);
    println!("{:?}", v);
}
```

Exercise: Can you make the above code compile without removing either of the borrows?

Why mutable borrows are always exclusive

If a mutable borrow exists, you can't have any more mutable or immutable borrows in the same scope. The reason is a bit tricky.

https://manishearth.github.io/blog/2015/05/17/the-problem-with-shared-mutability/

Borrowing Rules

- Any number of immutable borrows can co-exist.
- A mutable borrow can not co-exist with other mutable or immutable borrows.
- The “borrow checker” checks violations of these rules at compile time.

Borrow checker limitations

- The borrow checker gives you safety by rejecting ALL unsafe programs.
- But it is not perfect in the sense it rejects safe programs also; "fighting the borrow checker" is a common sporting activity among Rust programmers :)
- There are plans to improve the situation: http://smallcultfollowing.com/babysteps/blog/2017/03/01/nested-method-calls-via-two-phase-borrowing/

Borrow checker limitations - an example

```rust
// borrow6.rs
fn main() {
    let mut v = vec![10,20,30];
    v.push(v.len());
}
// this will not compile
```

Borrow checker limitations - an example

```rust
// borrow7.rs
// Same as a35.rs
fn main() {
    let mut v = vec![10,20,30];
    let tmp0 = &v;
    let tmp1 = &mut v;
    let tmp2 = Vec::len(tmp0); //v.len()

    Vec::push(tmp1, tmp2);// v.push(tmp2)
}
```

Lifetimes

```rust
// lifetime1.rs
fn main() {
    let ref1: &Vec<i32>;
    {
        let v = vec![1, 2, 3];
        ref1 = &v;
    }
    // v gets deallocated as it goes out of
    // the scope. What about ref1? Do we have
    // a "dangling pointer" here?
}
```

Lifetimes

```rust
// lifetime2.rs
fn foo() -> Vec<i32> {
    let v = vec![1, 2, 3];
    v // transfer ownership to caller
}
fn main() {
    let p = foo();
    println!("{:?}", p);
}
```

Lifetimes

```rust
// lifetime3.rs
fn foo() -> &Vec<i32> {
    let v = vec![1, 2, 3];
    &v // Will this compile?
}
fn main() {
    let p = foo();
}
```

Explicit Lifetime Annotations

```rust
// lifetime4.rs
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> &i32 {
    &v1[0]
}
fn main() {
    let v1 = vec![1, 2, 3];
    let p:&i32;
    {
        let v2 = vec![4, 5, 6];
        p = foo(&v1, &v2);
        // How does the compiler know, just by looking at
        // the signature of "foo", that the reference
        // returned  by "foo" will live as long as "p"?
    }
}
```

Explicit Lifetime Annotations

```rust
// lifetime5.rs
fn foo<'a, 'b>(v1: &'a Vec<i32>,
               v2: &'b Vec<i32>) -> &'a i32 {

    &v1[0]
}
fn main() {
    let v1 = vec![1, 2, 3];
    let p:&i32;
    {
        let v2 = vec![4, 5, 6];
        p = foo(&v1, &v2);
    }
}
```

Unsafe

```rust
// unsafe1.rs
fn main() {
    // a is a "raw" pointer intialized to 0
    let a: *mut u32 = 0 as *mut u32;

    *a = 0;
}
```

Unsafe

```rust
// unsafe2.rs
fn main() {
    let a: *mut u32 = 0 as *mut u32;

    unsafe {
        *a = 0;
    }
}
```

Zero Cost Abstractions

```rust
// zerocost1.rs
const N: u64 = 1000000000;
fn main() {
    let r = (0..N)
            .map(|x| x + 1)
            .fold(0, |sum, i| sum+i);

    println!("{}", r);
}
// Compile with optimizations enabled:
// rustc -O zerocost1.rs
```

Zero cost abstractions

```
(0 .. N) => (0, 1, 2, 3, .... N-1) # an "iterator"

(0 .. N).map(|x| x+1) => (1, 2, 3, 4 .... N)

(1, 2, 3, ... N).fold(0, |sum, i| sum + i)

  => ((((0 + 1) + 2) + 3) + 4) + ....
```

Zero cost abstractions

Here is part of the assembly language code produced by the compiler for zerocost1.rs:

```
.Ltmp0:
    .cfi_def_cfa_offset 80
    movabsq $500000000500000000, %rax
    movq    %rax, (%rsp)
    leaq    (%rsp), %rax
    movq    %rax, 8(%rsp)
```

Looks like the expression has been evaluated fully at compile time itself!
Here is the commandline used to produce the above output:

```
rustc -O zerocost1.rs --emit=asm
```

Zero cost abstractions

- You can write confidently using all the high-level abstractions the language has to offer.
- Your code will almost always be as fast as hand-coded low level C!

Generic functions

```
// genericfunction1.rs
fn identity <T> (x: T) -> T {
    x
}
fn main() {
    let a = identity(10);
    let b = identity('A');
    let c = identity("hello");

    println!("{}, {}, {}", a, b, c);
}
```

Rust creates specialized versions of the “identity” function for each argument type. This is called “monomorphization”.

Product Types: Structures

```rust
// struct1.rs
struct Rectangle {
    h: f64,
    w: f64,
}
impl Rectangle {
    fn area(&self) -> f64 {
        self.h * self. w
    }
}
fn main() {
    let r = Rectangle { h: 2.0, w: 3.0 };
    println!("area = {}", r.area());
}
```

Traits

```rust
// trait1.rs
struct Rectangle {
    h: f64,
    w: f64,
}
struct Circle {
    r: f64,
}
// is "a" bigger than "b" in area?
// should work for any shape
fn is_bigger <T1, T2> (a: T1, b: T2) -> bool {
    a.area() > b.area()
}
fn main() {
    let r = Rectangle { h: 3.0, w: 2.0 };
    let c = Circle { r: 5.0 };
    println!("{}", is_bigger(r, c));
}
```

Traits

```
// part of trait2.rs
trait HasArea {
    fn area(&self) -> f64;
}
impl HasArea for Rectangle {
    fn area(&self) -> f64 {
        self.h * self.w
    }
}
impl HasArea for Circle {
    fn area(&self) -> f64 {
        3.14 * self.r * self.r
    }
}
fn is_bigger <T1:HasArea, T2:HasArea>
            (a: T1, b: T2) -> bool {
    a.area() > b.area()
}
```

Tools

- Cargo, the package manager (crates.io holds packages)
- rustfmt, formatting Rust code according to style guidelines
- clippy, a "lint" tool for Rust
- rustup (https://www.rustup.rs/), the Rust toolchain installer/manager

Interesting projects using Rust

- Servo, from Mozilla. The next-gen browser engine.
- Redox OS (https://www.redox-os.org/), an Operating System being written from scratch in Rust.
- ripgrep (https://github.com/BurntSushi/ripgrep), a fast text search tool.
- rocket.rs - a powerful web framework.
- More: https://github.com/kud1ing/awesome-rust

Companies using Rust

- Friends of Rust: https://www.rust-lang.org/vi-VN/friends.html

Documentation

- https://doc.rust-lang.org/book/ (Official Rust book - 2nd edition is far easier to understand)
- Upcoming O'Reilly book: http://shop.oreilly.com/product/0636920040385.do
- http://intorust.com/ (screencasts for learning Rust)

The NanoPi Neo

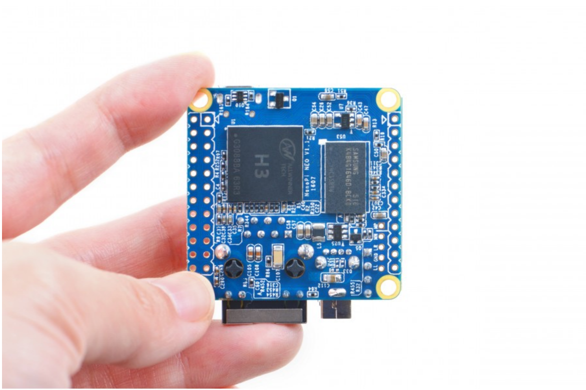

Figure 9: NanoPi Neo

NanoPi Neo Hardware Specifications

- Allwinner H3 SoC; Quad core Cortex A7 upto 1.2GHz
- DDR3 RAM: 256/512Mb
- 10/100 Ethernet
- USB Host Type-A x 1
- MicroSD slot
- 36 pin GPIO
- MicroUSB for power

NanoPi Neo (v1.2) Pinout

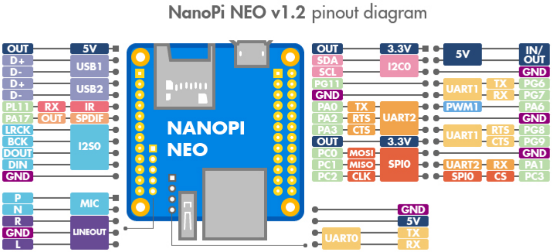

Figure 10: Neo Pinout

Using the GPIO pins

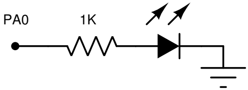

Figure 11: LED Connected to PA0

Using the GPIO pins

- We will use GPIOA0, GPIOA2, GPIOA3.
- These are pins 11, 13, 15 of the 24 pin connector.
- In code, these pins are identified by numbers 0, 2, 3.
- These pins also serve as Tx, RTS, CTS pins of UART2. UART2 is disabled by default.
- We will use GPIOA0 and GPIOA2 as output and GPIOA3 as input.

Using the sysfs interface to GPIO

```
$ echo 0 > /sys/class/gpio/export

$ echo out > /sys/class/gpio/gpio0/direction

# Put on LED
$ echo 1 > /sys/class/gpio/gpio0/value
```

Cross compiling for ARMv7 targets

After installing Rust using *rustup*, run:

```
$ rustup target add armv7-unknown-linux-gnueabihf
```

```
$ sudo apt-get install gcc-4.7-multilib-arm-linux-gnueabihf
```

Add the following to ~/.cargo/config:

```
[target.armv7-unknown-linux-gnueabihf]
linker = "arm-linux-gnueabihf-gcc-4.7"
```

Here is how you compile the code:

```
cargo build --target=armv7-unknown-linux-gnueabihf
```

A Rust program which uses the sysfs interface

```rust
use std::fs::File;
use std::io::prelude::*;
fn main() {
    // The value returned by write_all is ignored.
    let mut f1 =
        File::create("/sys/class/gpio/export")
        .unwrap();
    f1.write_all(b"0");
    let mut f2 =
        File::create("/sys/class/gpio/gpio0/direction")
        .unwrap();
    f2.write_all(b"out");
    let mut f3 =
        File::create("/sys/class/gpio/gpio0/value")
        .unwrap();
    f3.write_all(b"1");
}
```

Another approach: Link to a C library

- WiringNP (https://github.com/wertyzp/WiringNP) is a popular C library for GPIO manipulation. It is a port of the original wiringPi developed for the Raspberry Pi.
- We will write Rust code which interfaces with WiringNP.
- Large C/C++ programs can be incrementally re-written in Rust. Start with those parts which are critical with regards to safety.

Linking Rust code with C: Calling a standard library function

We will call the standard C math library function *fabs* from Rust:

```
#[link(name = "m")]

extern {
    fn fabs(x: f64) -> f64;
}

fn main() {
    let t = unsafe { fabs(-1.2) };
    println!("{}", t);
}

$ rustc a.rs
```

Linking Rust code with C: Calling a standard library function

We will now wrap the *unsafe* code inside a *safe* Rust function:

```
#[link(name = "m")]
extern {
    fn fabs(x: f64) -> f64;
}
fn rust_fabs(x: f64) -> f64 {
    unsafe { fabs(x) }
}
fn main() {
    let t = rust_fabs(-10.23);
    println!("{}", t);
}
```

```
$ rustc a.rs
```

Linking Rust code with our own shared library

```
$ cargo new clink-tougher --bin
$ cd clink-tougher; mkdir sharedlib
$ cd sharedlib
```

Let's first create a shared library under the folder *sharedlib*:

```
int c_add(int x, int y)
{
    return x+y;
}
```

```
$ gcc -fpic -c mylib.c
$ gcc -shared -o libmylib.so mylib.o
```

libmylib.so is our shared library!

Linking Rust code with our own shared library

We will now link libmylib.so with the following C code and execute the resulting *a.out*:

```
#include <stdio.h>
extern int c_add(int, int);
int main()
{
    printf("%d\n", c_add(10, 20));
    return 0;
}
```

```
$ cc main.c -L. -lmylib
```

```
$ LD_LIBRARY_PATH=. ./a.out
```

Linking Rust code with our own shared library

The next step is to link our shared library with *Rust*, instead of C. First, we need a *build.rs* file in our project's root directory:

```
fn main() {
    let path =
    "/home/recursive/workshops/2017/\
     icefoss/code2/clink-tougher/sharedlib";

    println!("cargo:rustc-link-search={}", path);
    println!("cargo:rustc-link-lib=mylib");

}
```

Linking Rust code with our own shared library

Here is the file *src/main.rs*:

```
extern {
    fn c_add(x:i32, y:i32) -> i32;
}

fn main() {
    let r = unsafe {c_add(1,2)};
    println!("r = {}", r);
}
```

At the project root directory, do:

```
$ cargo build
$ cp target/debug/clink-tougher sharedlib
$ cd sharedlib
$ LD_LIBRARY_PATH=. ./clink-tougher
```

Linking Rust code with our own shared library

What if we need to generate an executable which runs on an ARMv7 board? A few small changes have to be made.

- First, we should build the shared library using a cross compiler, say: *arm-linux-gnueabihf-gcc-4.7*
- Then, we should generate the Rust executable by running:

```
$ cargo build --target=armv7-unknown-linux-gnueabihf
```

The resulting executable as well as the shared library should be copied to the ARM board.

Introducing bindgen

bindgen (https://github.com/rust-lang-nursery/rust-bindgen) takes in a header file containing C function prototypes, structure declaration etc and generates "extern C" declarations automatically. Consider this file *a.h*:

```
int add(int x, int y);

unsigned char upcase(char c);
```

Introducing bindgen

Let's now run:

```
$ bindgen a.h
```

Here is the output:

```
/* automatically generated by rust-bindgen */

#![allow(dead_code,
         non_camel_case_types,
         non_upper_case_globals,
         non_snake_case)]
extern "C" {
    pub fn add(x: ::std::os::raw::c_int,
               y: ::std::os::raw::c_int)
     -> ::std::os::raw::c_int;
    pub fn upcase(c: ::std::os::raw::c_char)
     -> ::std::os::raw::c_uchar;
}
```

Let's now call WiringPi functions from Rust!

[Note: There are better ways to do this. Consider this as a quick-n-dirty hack]

- Copy the wiringPi shared library from the ARMv7 board (NanoPi Neo) itself. It is available as: /usr/lib/libwiringPi.so
- Get the WiringNP source code: https://github.com/wertyzp/WiringNP.
- Run bindgen on wiringPi.h which is part of the above source distribution, cross compile and deploy to the ARM board!

Check out the folder *wiring1* in the workshop source code to see an example.

A wiringPi based LED blinking, I/O pin reading program

GPIOA0 is output and GPIOA3 is input.
Source code: *wiring1/src/main.rs*

```
extern crate wiring1;
use wiring1::*;

const PIN_LED: i32 = 0;
const PIN_INPUT: i32 = 3;
```

A wiringPi based LED blinking, I/O pin reading program

```
fn main() {
    wiringPiSetup();
    pinMode(PIN_LED, cffi::OUTPUT);
    pinMode(PIN_INPUT, cffi::INPUT);
    loop {
        digitalWrite(PIN_LED, cffi::HIGH);
        println!("input = {}", digitalRead(PIN_INPUT));
        delay(500);
        digitalWrite(PIN_LED, cffi::LOW);
        println!("input = {}", digitalRead(PIN_INPUT));
        delay(500);
    }
}
```

A command-line GPIO application

Let's write a small commandline application to control two LED's (GPIOA0 and GPIOA2) and read status of a digital input (GPIOA3). Here is the simplest possible code you can write to access commandline arguments:

```
use std::env;

fn main() {
    let args:Vec<String> = env::args().collect();
    println!("{:?}", args);
}
```

A command-line GPIO application

Here is what we want to do:

```
$ gpioctrl --led1 on # put on LED1 on GPIOA0
$ gpioctrl --led1 on --led2 on # put on both LED's
$ gpioctrl --led2 off # put off LED2
$ gpioctrl --readinput # read the digital input pin
$ gpioctrl --readinput --led2 on --led1 off
```

A command-line GPIO application

We will use the *structopt* library (https://github.com/TeXitoi/structopt) to simplify argument parsing.

```
extern crate structopt;
#[macro_use]
extern crate structopt_derive;

use structopt::StructOpt;
```

A command-line GPIO application

```
#[derive(StructOpt, Debug)]
struct Opt {
  #[structopt(long = "led1", help = "turn led1 ON/OFF")]
  led1: Option<String>,

  #[structopt(long = "led2", help = "turn led2 ON/OFF")]
  led2: Option<String>,

  #[structopt(long="readinput", help = "read GPIOA3")]
  readinput: bool,
}

fn main() {
  let opt = Opt::from_args();
  println!("{:?}", opt);
}
```

A command-line GPIO application

TODO: Extend the code in the previous slide and make it read/write GPIO pins.

A UDP based client/server program

```
//client.rs

use std::net::UdpSocket;

const serv_addr: &str = "127.0.0.1:8000";
const cli_addr: &str = "0.0.0.0:0";

fn main() {
    let mut socket = UdpSocket::bind(cli_addr).unwrap();

    let mut buf = [4; 10];
    socket.send_to(&buf, serv_addr);
}
```

A UDP based client/server program

```
//server.rs

use std::net::UdpSocket;
const addr: &str = "127.0.0.1:8000";

fn main() {
    let mut socket = UdpSocket::bind(addr).unwrap();

    let mut buf = [0; 10];
    let (amt, src) = socket.recv_from(&mut buf).unwrap();
    println!("{}, {}, {:?}", amt, src, buf);

}
```

Control gpio pins over the network

TODO: integrate the command-line gpio control and networking code we have written to control gpio pins over the network.

Interfacing the MCP3008 ADC with the SPI bus

10 bit, 8 channel ADC from Microchip with SPI interface.

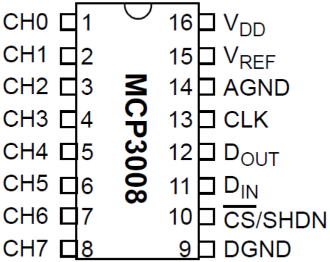

Figure 12: MCP3008

Interfacing the MCP3008 ADC with the SPI bus

Connections (MCP3008 pins on the left):

```
CH0           --> Analog input
Vdd, Vref     --> 3.3V
AGND, DGND    --> GND
CLK           --> SPI0 CLK  (Nanopi)
DOUT          --> SPI0 MISO (Nanopi)
DIN           --> SPI0 MOSI (Nanopi)
CS/SHDN       --> SPI0 CS   (Nanopi)
```

Interfacing the MCP3008 ADC with the SPI bus

Communication logic (from NanoPi to the MCP3008)

- The NanoPi (master) sends 3 bytes of data; the slave (MCP3008) responds with 3 bytes of data.
- First byte of data sent by NanoPi is 1. This is a “start byte”.
- Second byte of data sent by NanoPi is binary: 1 0 0 0 0 0 0 0. Most significant bit should be 1 for single-ended conversion. The next 3 bits select the channel: 0 0 0 selects channel 0. The least significant 4 bits are don’t-care.
- Third byte sent by the NanoPi is again a don’t-care.

Interfacing the MCP3008 ADC with the SPI bus

Communication logic (from MCP3008 to the NanoPi):

- Response from the MCP3008 is 3 bytes of data: first byte is don't care, least significant two bits of the next byte are bits 9 and 8 of the 10 bit value returned by the ADC. The next byte contains all the other bits of converted data.

Interfacing the MCP3008 ADC with the SPI bus: using the spidev crate

We use spidev: https://crates.io/crates/spidev

```
extern crate spidev;
use std::io;
use spidev::{Spidev, SpidevOptions,
             SpidevTransfer, SPI_MODE_0};

fn create_spi() -> Spidev {
    let mut spi = Spidev::open("/dev/spidev0.0").unwrap();
    let options = SpidevOptions::new()
         .bits_per_word(8)
         .max_speed_hz(100_000)
         .mode(SPI_MODE_0)
         .build();
    spi.configure(&options).unwrap();
    spi
}
```

Interfacing the MCP3008 ADC with the SPI bus: using the spidev crate

```
fn full_duplex(spi: &mut Spidev)  {
    let tx_buf = [0x01, 0x80, 0x0];
    let mut rx_buf = [0; 3];
    {
        let mut transfer =
                    SpidevTransfer
                    ::read_write(&tx_buf, &mut rx_buf);
        spi.transfer(&mut transfer).unwrap();
    }
    println!("{:?}", rx_buf);
}

fn main() {
    let mut spi = create_spi();
    println!("{:?}", full_duplex(&mut spi));
}
```

Hope all of you enjoyed playing with Rust! Keep learning more!

Contact me

- Email: mail@pramode.net