Project Oxide

Real Time Operating System

Background

While Internet of Things (iOT) technologies have advanced greatly in recent years, the embedded technologies that many of these iOT devices still run on operating systems built using the C programming language.

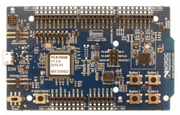
Our project, OSxide, aims to modernize real time operating systems by using Rust, a modern systems programming language that focuses on speed and security. By using Rust, we hope to mitigate many of the security vulnerabilities that have historically come with systems build using C.

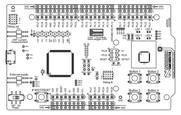
Scope and Objectives

While the motivation for a real time operating system (RTOS) in Rust comes from the need for secure iOT platforms, the scope of this project is narrower, with the main objective being the construction of the most basic RTOS in Rust.

Objectives:

• Explore the capabilities of Rust in an embedded device setting. Particularly, in the context of the NRF51 development kit we used for testing.





• Develop a small and basic real time operating system in Rust. This requires the development of a very simple scheduler.

Why Rust?

Rust is a programming language syntactically similar to C++ but it provides better memory safety while maintaining performance. It's main benefits towards embedded devices are:

Memory Safety

• Rust does not permit null pointers, dangling pointers, or data-races in safe code.

Memory Management

• Rust does not use a garbage collector instead manages resources using the programming idiom resource acquisition is Initialization (RAII)

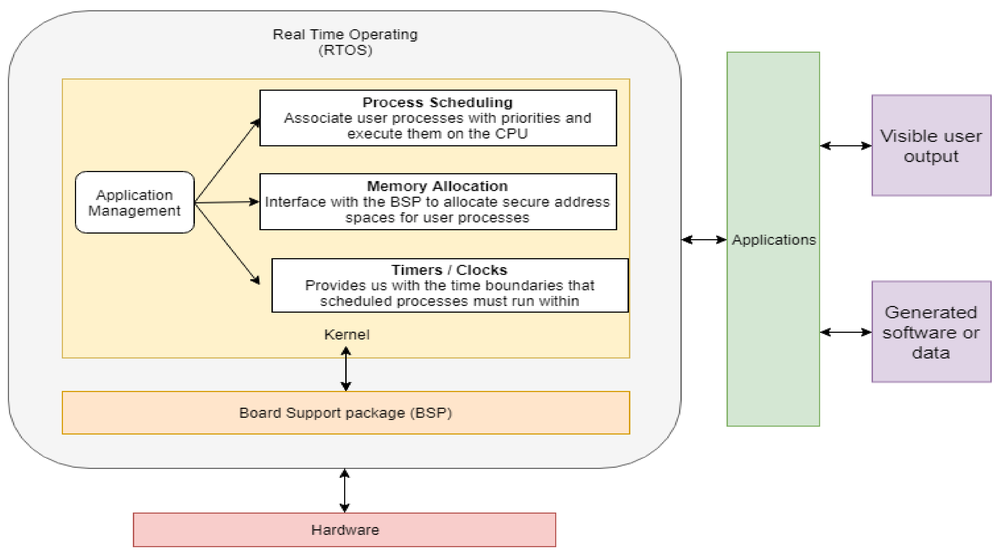
Ownership

• Rust has a unique ownership system where all variables have a single unique owner where the variable and owners scopes overlap

Design and Approach

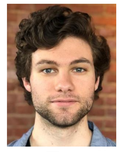
Initially our approach was to design from the ground up creating everything from the board support package (BSP) to the kernel. We quickly determined that this was avoidable as Nordic systems has premade packages available for use; however, these packages are all written in C and we wanted to target Rust. After some digging we came across Jorge Aparicio, an embedded Rust developer, who has been developing tools for embedded Rust development for the last few years. Utilizing his tools we were able to generate some base Rust code which allowed access to our device's peripherals and chip functionality.

Our approach models a typical RTOS setup. The BSP interfaces directly with the hardware and chip registers. The kernal utilizes this hardware interface to allocate memory, schedule process, and ensure time boundaries. The applications then communicate with these layers to perform tasks on the hardware.



Team Members

Doug Flick



Dom Farolino

Codi Burley

Dan Wendelken

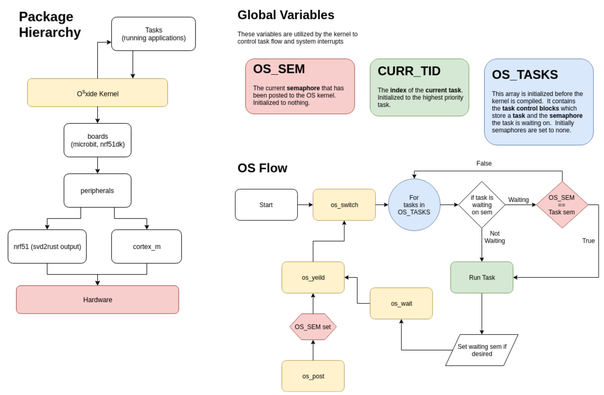
Kernel Design

We Implement a basic RTOS kernel with the following 3 basic components:

• **Scheduler:** Handles task switching based on priority and inter-task signaling with semaphores

• **Task Control Blocks**: The data structures that encapsulates task information

• **Communication**: semaphores allow for inter-task communication



Future Work

We have implmented a bare bones kernel with a basic BSP for the NRF51. Several expansions to these peices can be carried out.

1. Expand our BSP for the NRF51 to add more functionality.

2. Handle erratum in documentation.

3. Implement more board packages.

4. Expand kernel (more advanced task synchronization, I/O, etc.)