Altus Metrum

Software Design Document v1.0

AltOS Rust Project v1.3.0

AltosRust Team
15 March 2017

1 Table Of Contents

2 Introduction

2.1 Purpose

The purpose of this document is to provide an overview of the AltOS Rust software design based on the AltOS Rust requirements specification.

2.2 Scope

This document outlines the implementation details of the AltOS Rust operating system. The software will consists of two major pieces; The core operating system, and the peripheral drivers which interface with the operating system.

2.3 Intended Audience

This document is intended for use by developers, to outline and describe the design of the system being built.

2.4 Reference Material

https://www.rust-lang.org
Cortex-M0 Technical Reference Manual
STM32F0xx Reference Manual (RM0091)

2.5 Glossary

2.5.1 Definitions

AltOS: The current operating system for Altus Metrum components implemented in the C programming language.
AltOS Rust: The working name of the software system developed. This also refers to the project as a whole. This is also referred to as the *target system.*
Condition Variables: A variable associated with a Condition. Typically used for synchronization.
Context: The current state of a task, including all its register values, stack values, and the next instruction to be executed.
Context Switch: Saving the context of the current task and switching to a new task's context so it can start running.
Cortex-M0: The ARM processor used in the STM32F0xx family of microcontrollers.

Mutex Locks: A mutually exclusive lock that may only be held by a single task.
Run Ratios: A scheduling strategy that tries to prevent starvation by allowing lower priority tasks to be scheduled ahead of higher priority task via a defined ratio.
Rust: A systems programming language that supports compile-time memory safety, speed and concurrency.
STM32F0xx: A microcontroller family produced by STMicroelectronics.
Serial Port: Interface in which information transfers in or out, one bit at a time.
Scheduler: The entity in charge of choosing which task is going to be run next.
Synchronization Primitive: A tool to ensure multiple tasks behave in a synchronized manner when accessing a shared resource in order avoid data races.
Task: A self contained unit of execution isolated from other tasks, except through the access of shared resources.
Time Slice: A unit of time given to each task by the scheduler.

3 Required Core Software Implementations

This section covers required core software components including: operating system services for tasks, scheduling, memory management, and the serial communication driver.

3.1 Core Operating System

The AltOS operating system consists of a layered architecture, with each layer of functionality using the services provided by the layers below it.

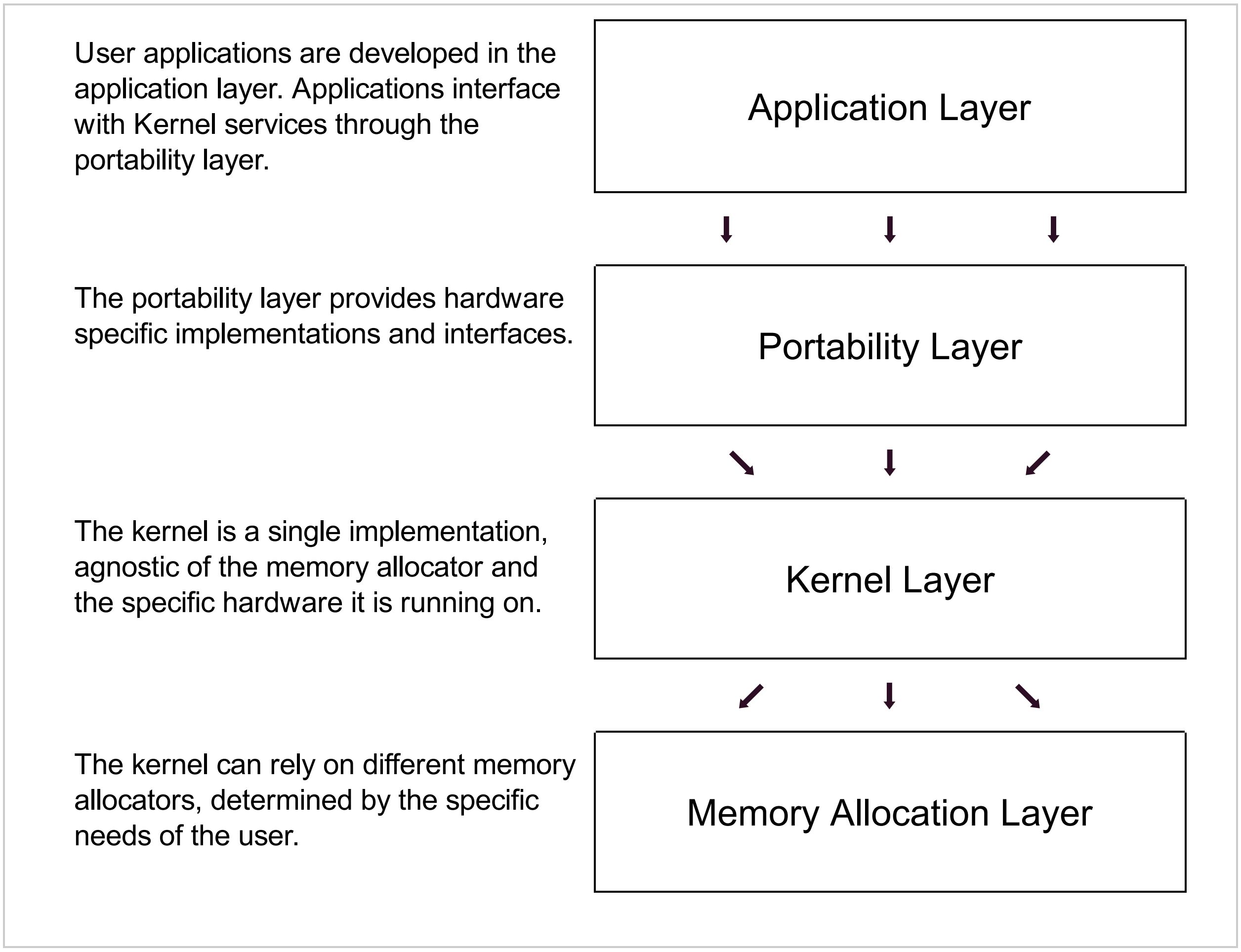
The lowest layer is the **Memory Allocation Layer**. This layer determines how heap space is allocated and reclaimed, and provides a means for dynamic memory allocation for both the kernel and user applications.

The second lowest layer is the **Kernel Layer** which provides platform agnostic operating system abstractions for task creation, task scheduling, synchronization primitives, and system calls.

The second highest layer is the **Portability Layer** which provides platform specific implementations and interfaces into the kernel. Interrupt handlers and device drivers are implemented at this level.

The top layer is the **Application Layer** which uses services exposed by the Portability and Kernel Layers in their implementations.

AltOS' Layered Architecture



User applications are developed in the application layer. Applications interface with Kernel services through the portability layer.

The portability layer provides hardware specific implementations and interfaces.

The kernel is a single implementation, agnostic of the memory allocator and the specific hardware it is running on.

The kernel can rely on different memory allocators, determined by the specific needs of the user.

3.1.1 Tasks

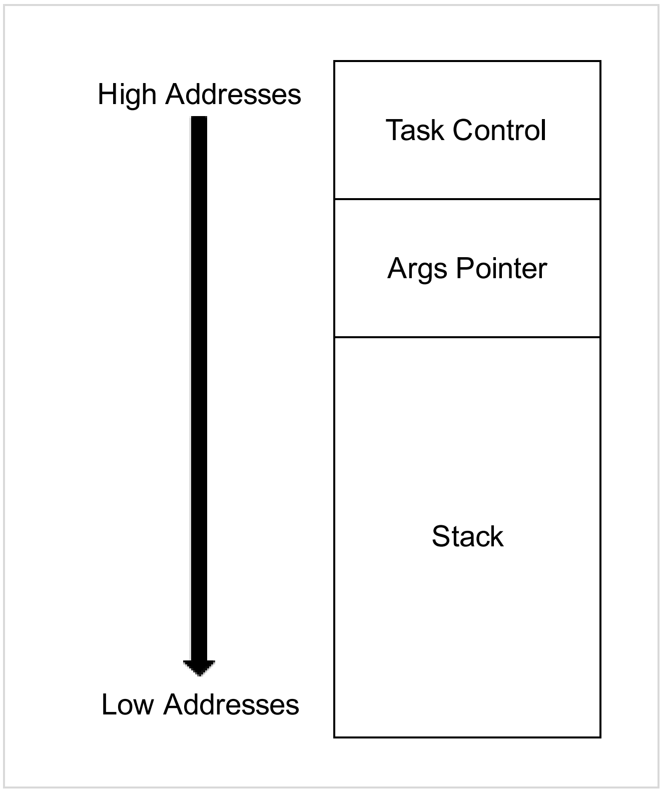
Tasks are process-like abstractions. Each task has an independent, isolated register and memory context, and a single thread of execution on the processor. The task's context and the amount of memory allocated to it is controlled and monitored by the operating system through a task data structure.

When created, a handle to the task data structure is returned. The task handle may be used to check or modify certain aspects of the task including suspending or destroying it. Task handles may be safely passed to any number of tasks, as synchronization is taken care of by the task handle's methods.

A task can be created with a variable number of arguments by passing in an array of pointer-width integers. Each array member can be interpreted as a number type or a pointer to an object. With this design, the task itself must interpret the types of each argument, resulting in the loss of type safety when passing objects to tasks.

A task **SHOULD** be a divergent function. The action taken upon returning from a task is defined in the kernel layer and the return address is determined when initializing the task's stack. If a task can safely be destroyed, it can be marked for destruction, however this does not immediately free memory associated with that task. The operating system will decide when to destroy tasks marked for destruction at its own convenience.

Task Structure in Memory



The **Args Pointer** points to the array of pointer-width integer arguments. It is very likely the arguments will be stored right below the **Task Control**, but not guaranteed.

3.1.2 Scheduler

AltOS Rust is designed with a preemptive scheduler. When a context switch occurs, either through a system interrupt or a voluntarily yield, the next scheduled task will be determined using a Round Robin Strategy that utilizes Run Ratios to prevent starvation. Under this strategy tasks are selected in a Round Robin fashion from the highest priority queue available, but no more than N normal priority tasks can be run successively without first attempting to schedule a low priority task. A task that is not in a ***critical section*** may be interrupted at any time to make room for another task.

AltOS Rust provides synchronization primitives to control access to shared memory. If a block of code exists that absolutely cannot be interrupted, a ***critical section*** can be declared around that block of code. While in a critical section the current running task cannot be preempted until after the ***critical section*** ends. Use of ***critical sections*** should be considered carefully and used sparingly as any errors within a ***critical section*** could lock the CPU, and prevent other tasks from running.

Task can have three different priority levels: ***Critical, Normal,*** and ***Low***

- ***Critical*** priority tasks **SHOULD** be time sensitive, system level tasks, that require immediate attention (such as system drivers). ***Critical Tasks*** are prioritized above all other tasks and should not be used for user applications. ***Critical*** priority tasks do not inherently run inside of a critical section, but critical sections may be declared within ***Critical tasks*** to prevent interrupts.
- ***Normal*** priority tasks **SHOULD** make up a majority of tasks. Each task should do its work and allow the scheduler to handle giving each task a fair time slice. Yielding the CPU is not necessary, however doing so will allow other tasks to utilize that time.
- ***Low*** priority tasks **SHOULD** be tasks that do not need to run consistently. A maximum of 1 ***Low*** priority task will run for every N ***Normal*** tasks, where N is a constant specified before compilation.

A task's priority is set upon its creation and cannot be changed.

Tasks can be in several states: ***Running***, ***Ready***, ***and Blocked***

- A ***Running*** task is the currently executing task.
- A ***Ready*** task is a task that is not running, but is ready to run. When the scheduler is invoked it will select a ready task and move it into the ***Running*** state.
- A ***Blocked*** task is a task that is not running, and it is waiting for an event to unblock it, such as an I/O event, a lock release, or a condition to be fulfilled. When an event occurs, which unblocks a task, that task is set to the **Ready** state and placed at the end of the appropriate priority queue.

The scheduler handles all state changes. The current state of a task can only be observed through a task handle, and cannot be modified.

3.1.3 Synchronization

AltOS Rust provides several synchronization primitives to control access to shared resources, including Mutex locks and Condition variables.

Mutex locks ensure that only one task can have access to a shared resource at a time by taking ownership of and strictly controlling access to shared resources. There are two ways to request a lock; **blocking** and **non-blocking**.

- Attempting to take a lock in a blocking manner will block the current task if another task already holds the lock and force a context switch. This locking method is useful when a task's functionality relies on access to a shared resource.
- Attempting to take a lock in a non-blocking manner will not block the current task if the lock is already held, and instead the task will continue executing for the remainder of its time slice. This locking method is useful when a task relies on multiple shared resources, and work can be done on different resources if it fails to acquire the requested lock.

Condition variables are used to signal when a lock becomes available or some event has occurred. Generally condition variables are associated with a **predicate** and a **mutex**. The predicate **MUST** be verified inside the mutex before determining if the task requesting the lock must go to block and wait on the condition variable. Once an event occurs or a lock becomes available, a signal is sent to all blocked tasks waking them. When a task is woken up it will re-attempt to acquire the lock. Notifications are broadcasted to all tasks waiting on a lock, there is no way to signal only one task.

3.1.4 Context Switching

Context switching is platform specific. When a context switch has been initiated, either by a task yielding its time slice or a timer interrupt, the context of the currently running task is saved onto its stack. Control is then passed into the kernel scheduler, which chooses the next task to run. After the scheduler has chosen the next task, the stack pointer is updated to point to the stack of the chosen task, it's context is restored and then it resumes execution.

3.1.5 Interrupts

Interrupts are hardware specific and handled in the portability layer.

3.1.6 Memory Management

The operating system handles dynamic memory allocation and deallocation. Tasks can create a stack with a predetermined size, specified in bytes. Memory management is also used during runtime for dynamic allocation. Tasks are given full control over the memory allocated to them.

The system keeps track of free memory using a linked list structure. The structure keeps track of free blocks of memory which have not been allocated to tasks. The list is kept sorted based on memory addresses to allow for the easy merging of free blocks on deallocation, allowing the system to reduce fragmentation. Nodes representing free blocks of memory are stored in memory within the free space itself, to avoid extra memory overhead.

Initially, the list consists of a single node containing all system memory. When a memory allocation request is serviced, the list is scanned for the first node with enough memory to satisfy the request. The size of the memory managed by the node shrinks when memory is allocated. When memory is deallocated, a new node is created with the size of the deallocation. Where possible, the system merges new free blocks with adjacent free blocks as soon as a deallocation occurs.

To deal with potential memory errors, the allocator keeps the size of every block of free memory and every memory allocation aligned with the size of a free block node itself. This means that allocations may include a few more bytes than the requested amount.

If an amount of memory is requested that is more than what is currently available in any block of free memory, the system returns a null pointer.

3.1.7 Clock Initialization

The operating system contains an enumerator for system clocks, and provides functionality to enable, disable, and select clocks. Variable frequency clocks may also have their frequency adjusted. Clock initialization functionality is hardware dependent and implemented in the portability later.

3.1.8 Device Management

Drivers for the system are modeled as tasks, and are implemented in the portability layer. Each time an interrupt is received, the operating system wakes up the corresponding task that is blocked on that data. Driver tasks are always set to have a critical priority level, in order to ensure that the scheduler will give preference to drivers over all other lower priority tasks.

3.2 Serial Driver

The serial peripheral is comprised of a series of registers that are used for configuration, interrupt handling, and bit management within the serial driver. The serial peripheral uses the portability layer to map the serial registers to hardware memory addresses, using the appropriate offsets. The addresses and offsets are defined within the STM32F0xx reference manual.

The current serial configuration uses the USART2 register to transfer bytes through the serial port. USART2 is configured for 9600 8N1 ─ a baud rate of 9600 Kb/s, wordlength of 8 bits, no parity, and 1 stop bit.

Interrupts are handled through the interrupt vector. There are two main interrupts: one for transmitting data, and one for receiving data. These interrupts are handled by the hardware, and are typically cleared as data is moved in and out of the receive and transmit registers.