BOOT LOADER

Architecture Design Document

2022-06-07

ST

This document is an Architecture Design Document for developing **BOOT LOADER**.

REVISION HISTORY

Version	Date	Author	Description
0.1	2022-06-07	ST	Initial document creation
0.2	2022-06-14	ST	Document for pre final submission
1.0	2022-07-05	ST	Document for final submission

1. Overview

1.1. Introduction

A **Bootloader** is a program that runs when a device is powered on and is responsible for loading the OS or main application that runs the device. A general bootloader performs the following tasks:

1. Testing all essential storage and peripheral devices are working
2. Loading the operating system Kernel into memory
3. Executing the Kernel `init` process
4. Providing a way to change bootloader configuration

1.2. System Definition

The purpose of this project is to create a bootloader for a Linux based OS which runs on a product line of Smart Devices and appliances such as TVs, Tablets, Mobile Phones, Refrigerator Display, Microwave Display, Smart Speakers, Smart Monitors etc.

Below Figure 1 depicts the system boundary and how it will interact with outside components and actors:

1. A Boot ROM embedded in the smart device firmware will execute when the smart device is powered on by the user.
2. The primary actor on the system is the user of the device, but the user interacts with the bootloader through some peripheral or network input devices to provide input and interrupts or through the Boot ROM to power on the device
3. The bootloader developer is responsible for configuring the bootloader to interact with Boot ROM and Kernel.

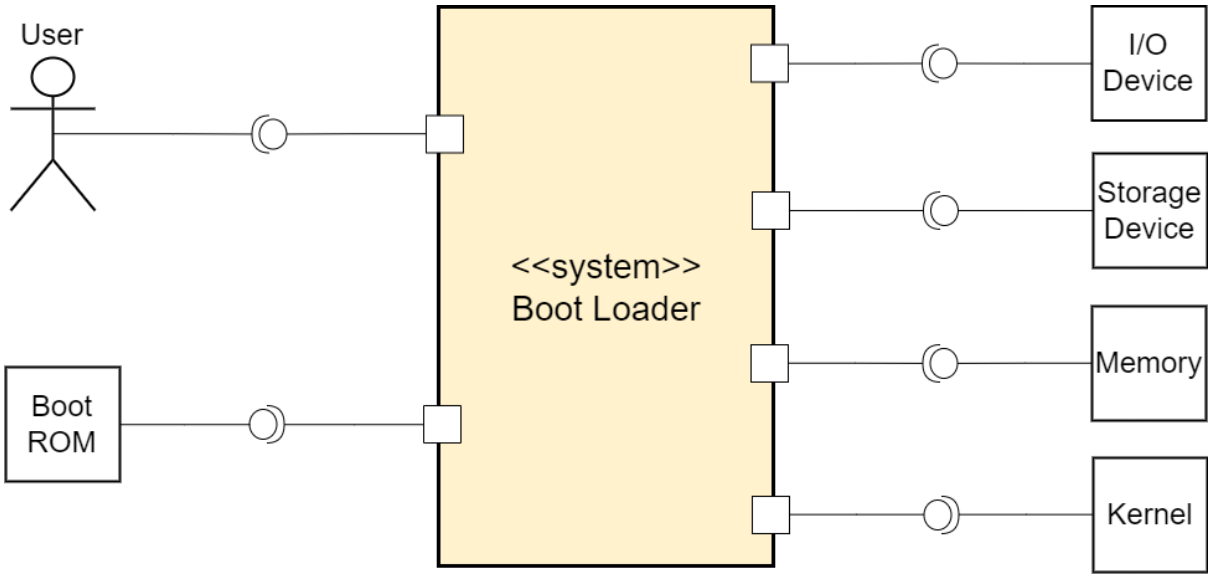


Figure 1: System Definition

User refers to the end user of the device to be booted. User interacts with the system in the following ways:

1. Power on the system, which executes the code in Boot ROM to initialize bootloader
2. Provide Interrupt to the Bootloader to load shell.
3. Provider CLI Input to the bootloader to perform shell commands

Boot ROM refers to the code in the firmware of the device which are executed when it is powered on. The Boot ROM specifies a location where the bootloader must be stored. When it runs, it executes the bootloader from the predefined location. Bootloader developer is responsible for placing bootloader in the location specified by Boot ROM.

I/O Device is a peripheral or network I/O or Display device connected to the system which can be used to display the shell or by the user to provide inputs.

Storage Device is a storage location which can be a physical storage device connected to the smart device like Flash or SSD, a peripheral storage device like USB or a network storage device on a server. It holds the image of the kernel that is to be loaded by the bootloader.

Memory refers to the various types of RAM available to the system which it can utilize for its

operation.

Kernel refers to the entry point of the operating system or application that the bootloader needs to execute.

1.3. Business Context

The operating system which the bootloader loads is a Linux based OS runs on a diverse product line that includes TVs, Tablets, Mobile Phones, Refrigerator Display, Microwave Display, Smart Monitors etc.

These devices have varied capabilities in terms of onboard storage as well as having different peripherals such as displays and input devices based on their use case. For example, TVs have remote controllers as primary input and built in display but microwave will have on device buttons or voice command as input and no built in display. Therefore, our bootloader should be adaptable to all these environments present in various devices.

Regardless of the device, bootloader operations should be conducted in a way that the user does not have to wait long in order to use their device.

In addition it is essential for us to provide a way for the user to interact with the bootloader in case some configuration changes are needed.

It is also important for our business to ensure that our devices use the OS or applications we provide to ensure device integrity and prevent revenue loss from some other OS being used on our devices.

Based on the above context, we can extract the following key business drivers for our system:

- Fast boot time that is in line with or exceeds industry standards of each product in the product line.
- Easy to add support for new devices
- Bootloader should be easily update-able if needed

- The bootloader should ensure only valid OS images are loaded on the smart devices to prevent misuse

2. Requirements

2.1. Functional Requirements

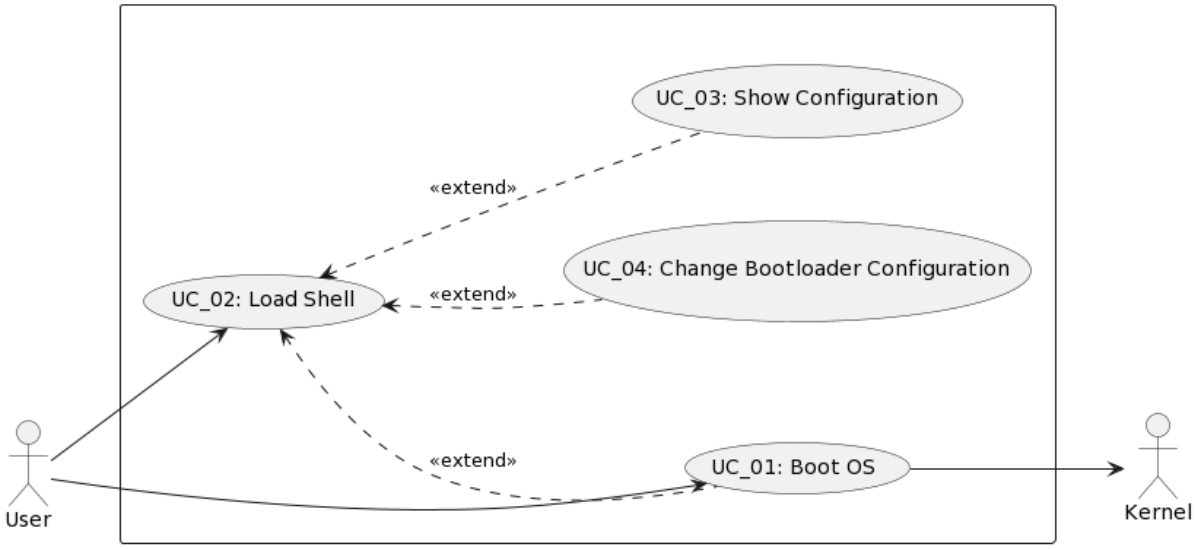


Figure 2: Bootloader - Use Case Diagram

UC_01	**Boot OS**
Description	When the device is powered on, load and execute the OS Kernel
Actor	User
Pre-condition	Device is off
Post-condition	OS Kernel entry point is executed
Basic Flow	1. User power on the device 2. Bootloader initializes hardware 3. Bootloader performs POST (Power On Self-Test) 4. Bootloader waits for shell entry interrupt from user 5. Bootloader loads kernel image from storage 6. Bootloader decompresses kernel image 7. Bootloader validates kernel image 8. Bootloader executes kernel
Additional Flow	If there is an interrupt in step 4: Bootloader loads shell (UC_02)

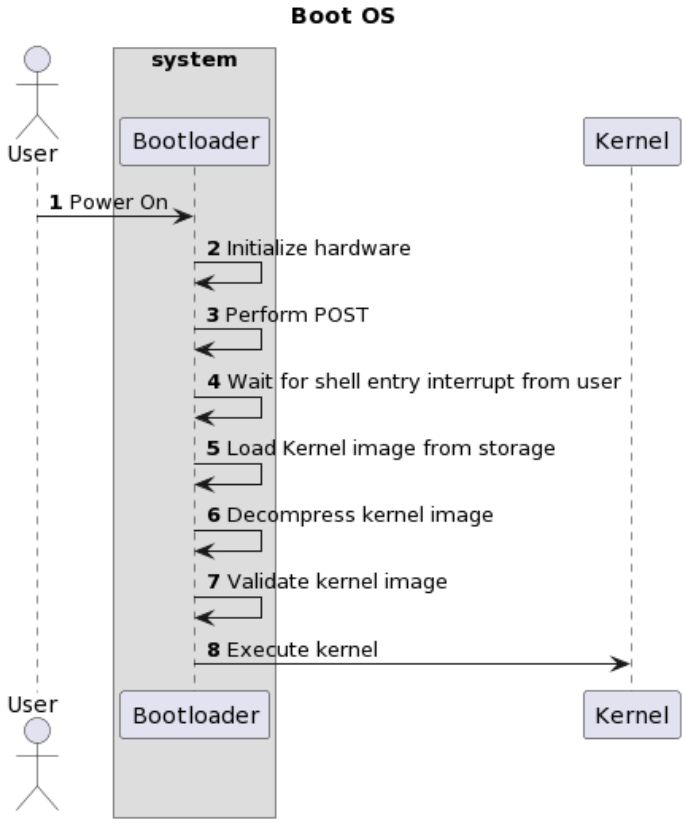


Figure 3: UC_01 - Boot OS

UC_02	**Load Shell**
Description	Load a shell to allow user to view/change configuration options or perform actions on the bootloader
Actor	User
Pre-condition	Bootloader is booting the OS kernel
Post-condition	Shell is displayed and bootloader waits for user command
Basic Flow	1. User asks bootloader to load shell 2. Bootloader suspends boot process 3. Bootloader initializes shell process 4. Bootloader displays shell to the user 5. Bootloader waits for user shell command
Additional Flow	User can perform the following commands after shell is loaded: 1. Show Configuration (UC_03) 2. Change Bootloader Configuration (UC_04)

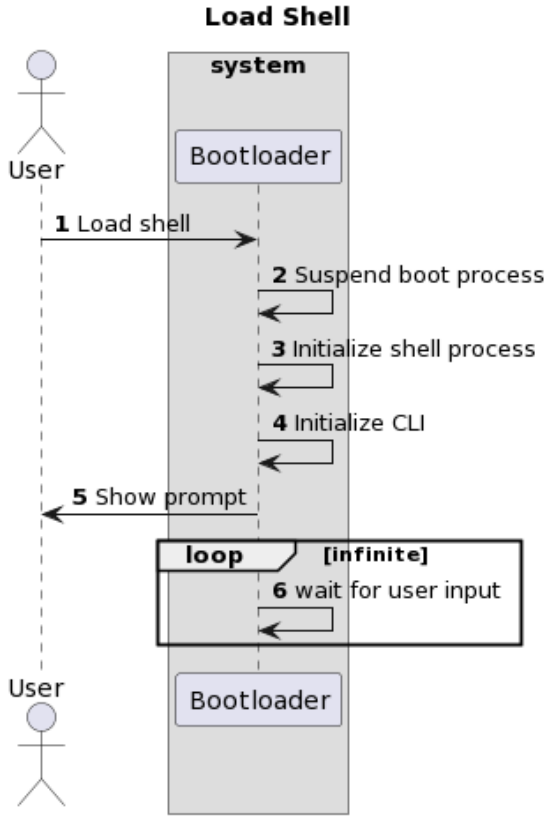


Figure 4: UC_02 - Load Shell

UC_03	**Show Configuration**
Description	Show the bootloader configuration to the user on shell output
Actor	User
Pre-condition	Shell is displayed and bootloader waits for user shell command
Post-condition	Bootloader Configuration is displayed on shell and bootloader waits for next user shell command
Basic Flow	1. User asks bootloader to change bootloader configuration 2. Bootloader parses the command 3. Bootloader reads configuration 4. Bootloader shows bootloader configuration to the user 5. Bootloader waits for user shell command
Additional Flow	If invalid command is detected at step 2: 1. Bootloader displays error message to the user 2. Bootloader waits for user shell command

UC_04	**Change Bootloader Configuration**
Description	Change/Set bootloader configuration like boot device, kernel image location on file system, display device etc.
Actor	User
Pre-condition	Shell is displayed and bootloader waits for user shell command
Post-condition	Boot configuration is changed and bootloader waits for next shell command
Basic Flow	1. User asks bootloader to change bootloader configuration 2. Bootloader parses the command 3. Bootloader changes configuration 4. Bootloader shows updated bootloader configuration to the user 5. Bootloader waits for user shell command
Additional Flow	If invalid command is detected at step 2: 1. Bootloader displays error message to the user 2. Bootloader waits for user shell command

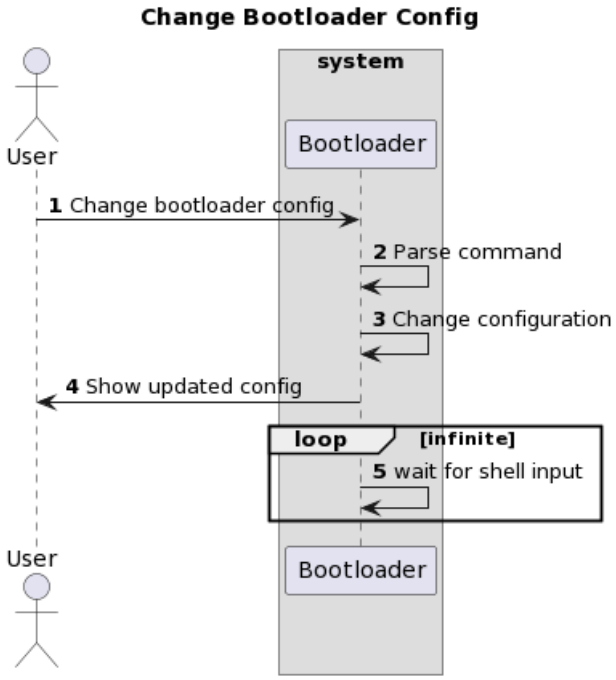


Figure 5: Change Bootloader Configuration

2.2. Non-functional Requirements

NFR_01, QA_01	Performance	Booting Time
Description	Boot time needs to be optimized as much as possible as booting is the main use case of our system and a slow booting time will lead to a bad user experience.	
Environment	Device is off	
Stimulus	User turns on the device and bootloader is executed	
Response	1. Bootloader initializes hardware 2. Bootloader performs POST (Power On Self-Test) 3. Bootloader waits for shell entry interrupt from user 4. Bootloader loads kernel image from storage	

	5. Bootloader decompresses kernel image 6. Bootloader validates kernel image 7. Bootloader executes kernel
Measure	[Response Time] = [Time when bootloader executes kernel] - [Time when user turns on the device]
Allowance	[Response Time]<=N milliseconds

NFR_02	**Performance**	**Bootloader Configuration Time**
Description	Time taken to show bootloader configuration and apply changes to it should after the user has requested it from the shell should be within a reasonable time as the user	
Environment	Shell is shown to the user and a prompt for user input is displayed	
Stimulus	User provides command to either view or change bootloader configuration	
Response	1. Bootloader parses the command 2. Bootloader reads configuration 3. Bootloader shows bootloader configuration to the user 4. Bootloader waits for user shell command	
Measure	[Response Time] = [Time when command response is displayed to user] - [Time when command is requested by user]	
Allowance	[Response Time]<=N milliseconds	

NFR_03	**Reliability**	**Kernel Image Recovery Time**
Description	Bootloader should be able to load kernel image even if kernel image is corrupted or storage device storing kernel image suffers hardware failure.	
Environment	Bootloader is running	
Stimulus	Bootloader tries to load kernel image	
Response	1. Kernel image fails to load 2. Bootloader identifies backup kernel loading method 3. Bootloader loads kernel image from backup	
Measure	[Response Time] = [Time when kernel image fails to load] - [Time when bootloader loads kernel image from backup]	

Allowance	[Response Time]<=N seconds

2.3. Quality Attributes

QA_02	Modifiability	Multi Core Modifiability
Description	We will use multi-core setup to improve QA_01. No. of cores in the CPU can differ for different devices. It should be easy for developers to change functionality without affecting multi core support and also add support for new type of multi core setup.	
Environment	During development or maintenance of the bootloader	
Stimulus	Developer receives request for supporting new core setup (single core, 2 core, 8 core etc.).	
Response	1. Developer plans changes to bootloader code 2. Developer makes required code changes 3. Developer analyzes impact on existing code 4. Developer tests and releases the code for new core setup 5. Developer creates binary for new core setup 6. Binary is deployed on device with new core setup	
Measure	[Development Cost]	

QA_03	Modifiability	Storage Modifiability
Description	Since we have a product line of devices we need to support they can have different storage devices where the kernel is stored. It should be easy and fast for developers to add support for a new storage device. Different storage devices can also be formatted by different file systems, this also needs to be managed by the bootloader.	
Environment	During development or maintenance of the bootloader	
Stimulus	Developer receives request for supporting new Storage Device	
Response	1. Developer plans changes to bootloader code 2. Developer makes required code changes 3. Developer analyzes impact on existing code	

	4. Developer tests and releases the code for supporting new storage device 5. Developer creates binary with support for new storage device 6. Binary is deployed on all devices which need to support the new storage device
Measure	[Development Cost]

QA_04	**Modifiability**	**I/O Modifiability**
Description	We can have different I/O devices which we need to connect to our bootloader for displaying shell or taking user inputs. It should be easy and fast for developers to add support for a new I/O device.	
Environment	During development or maintenance of the bootloader	
Stimulus	Developer receives request for supporting new I/O Device	
Response	1. Developer plans changes to bootloader code 2. Developer makes required code changes 3. Developer analyzes impact on existing code 4. Developer tests and releases the code for supporting new I/O device 5. Developer creates binary with support for new I/O device 6. Binary is deployed on all devices which need to support the new I/O device	
Measure	[Development Cost]	

QA_05	**Modifiability**	**Shell Modifiability**
Description	Shell should be modifiable relatively independently of main booting code and changes in shell should not affect booting code.	
Environment	During development or maintenance of the bootloader	
Stimulus	Developer receives request for changing shell code	
Response	1. Developer plans changes to bootloader code 2. Developer makes required code changes 3. Developer analyzes impact on existing code 4. Developer tests and releases the code 5. Developer creates binary with requested shell code changes	

	6. Binary is deployed on all devices which need to support the new shell menu.
Measure	[Size of affected modules]

QA_06	**Performance**	**Bootloader Storage and Loading Efficiency**
Description	Bootloader binary size should be minimized so that it can run on devices that have limited storage capacity and can be loaded quickly. We cannot directly control bootloader loading time as it is dependent on the device and Boot ROM, but we if we control bootloader binary size it will indirectly improve loading time.	
Environment	During Deployment	
Stimulus	Developer is requested to deploy bootloader on Device	
Response	1. Developer analyzes device storage 2. Developer creates binaries according to device storage capabilities 3. Developer deploys binaries in storage device(s)	
Measure	[Size of binary in bytes] = [Sum of size of binaries loaded to fulfill UC_01]	

3. Architecture

// A8. Architecture Documentation

// C8-1. Is allocation of processes, etc. appropriate? (deployment)

// C8-2. Is grouping appropriate in terms of components? (component & connector)

// C8-3. Is the description of the system architecture appropriate?

Deployment View

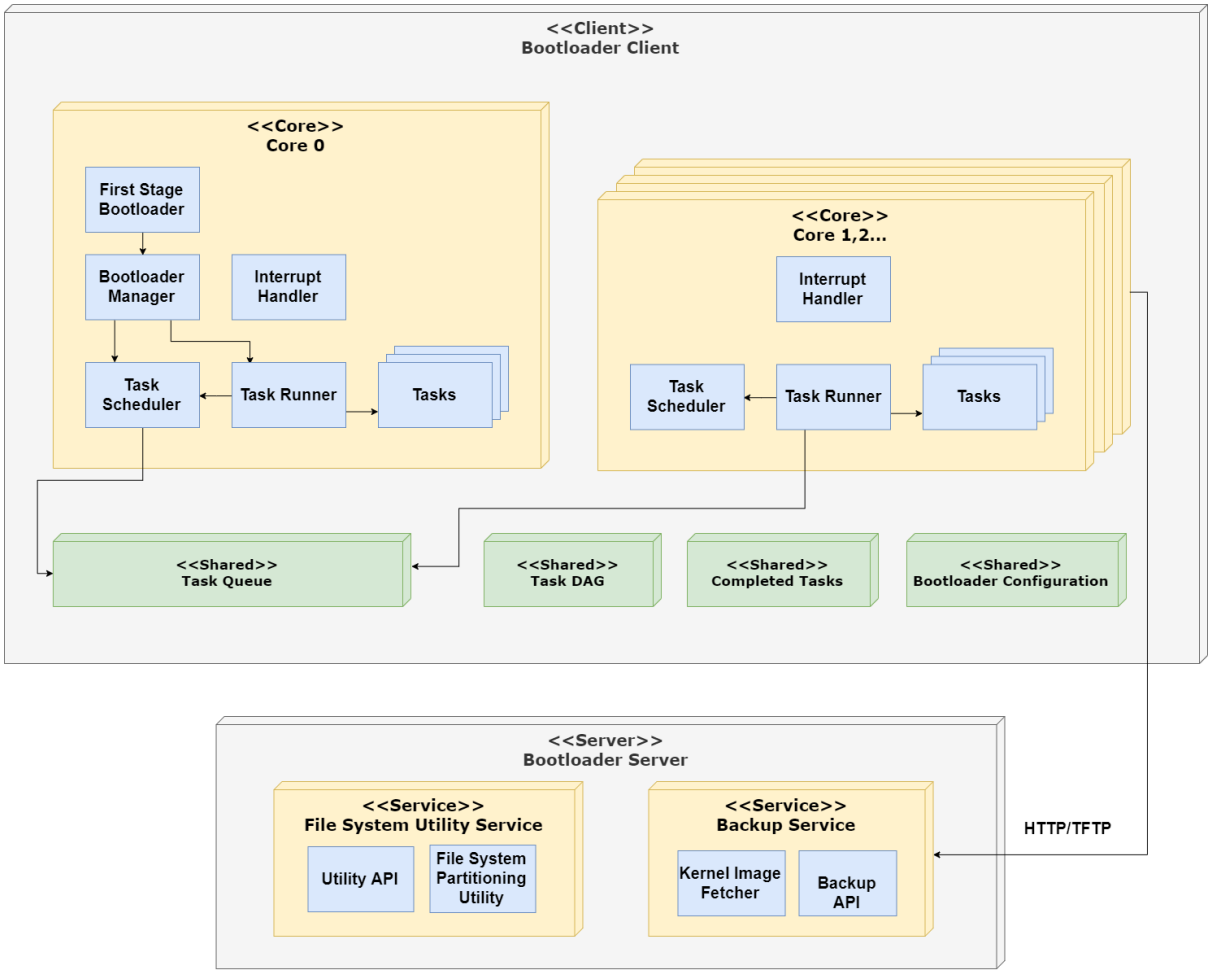


Figure 6: Deployment View for Multi Core Deployment

Component and Connector View for Kernel Image Reovery Task

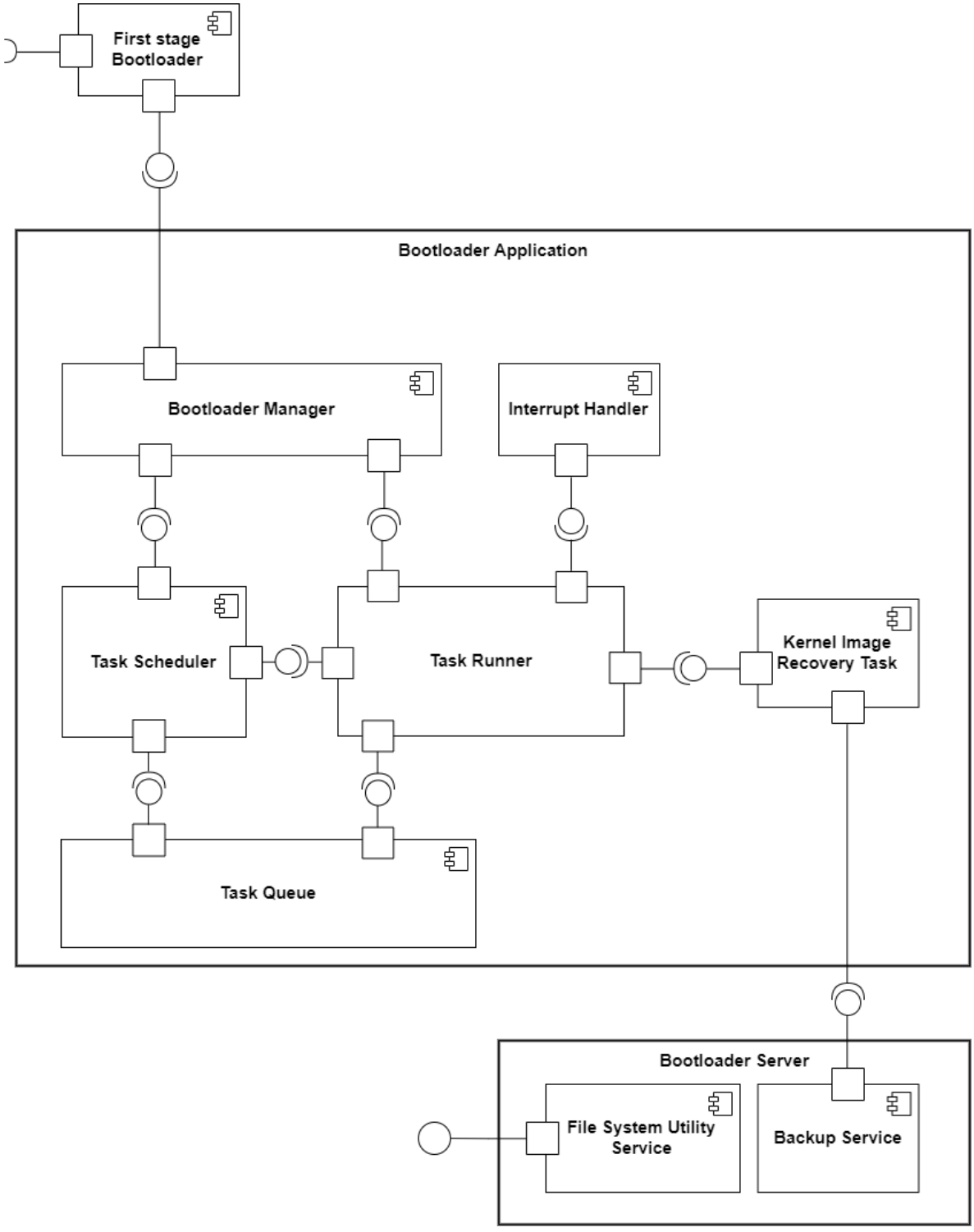


Figure 7: Component and Connector View for Kernel Image Recovery Task

Deployment Units:

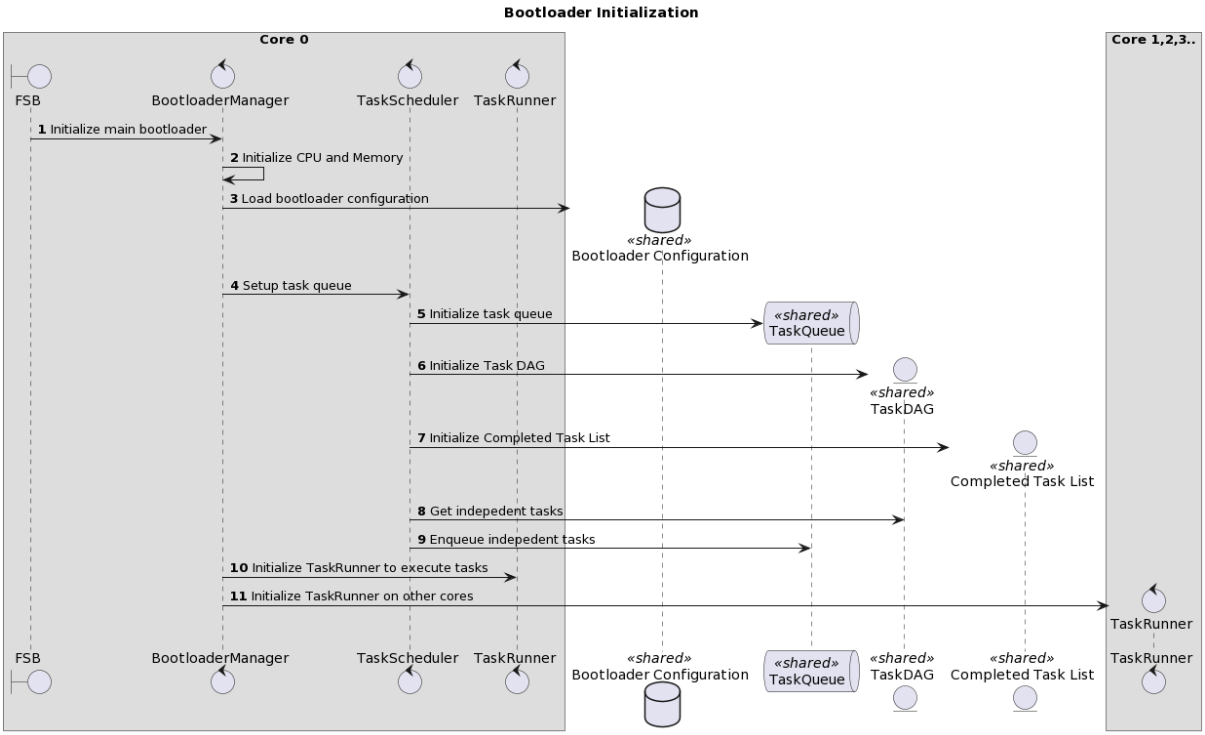
- **Client** represents the client device on which the bootloader is running
- **Core** represents a core on the client device where the bootloader is running. By using cores, we are using hardware concurrency during execution.
- **Shared** represents shared memory on the client device that all cores can access.
- **Server** represents the Bootloader server which the clients interacts with.

Components:

- **First Stage Bootloader:** Entry point for our code which is executed by the Boot ROM. It initializes Bootloader Manager in the Main Bootloader and exits.
- **Bootloader Manager:** High level manager for our bootloader which initializes the bootloader. It initializes basic hardware and sets up the all the cores and shared data.
- **Tasks:** Group of components representing functionalities to be performed by the bootloader such as Memory POST, Validation, Shell loading etc.
- **Task Queue:** Component managing a queue containing the tasks described above. It is a Multi-Producer Multi-Consumer (MPMC) queue to which all cores will enqueue and dequeue from.
- **Task Scheduler:** Responsible for adding Tasks to the Task Queue. It uses some sub-components which are deployed in shared memory for its functioning:
    - **Task DAG:** A topologically sorted Directed Acyclic Graph of all the dependencies between tasks.
    - **Completed Tasks:** The set of Tasks completed by the bootloader. It is implemented as a Hash Set.
- **Task Runner:** Responsible for taking Tasks from the Task Queue and executing them.
- **Interrupt Handler:** Interrupts the running Task on Task Runner when the Core receives an interrupt to do so.

- **Backup Service:** Service which provides a way to recover Kernel image when there is an issue in retrieving kernel image from the storage device. The Recovery Task will call the Backup API, which will use the Kernel Image Fetcher to get the relevant Kernel image from an online repository (provided by the OS provider).
- **File System Utility Service:** Service which provides a Utility to partition a Storage Device with a custom 'Bootloader File System' which is used by our bootloader.

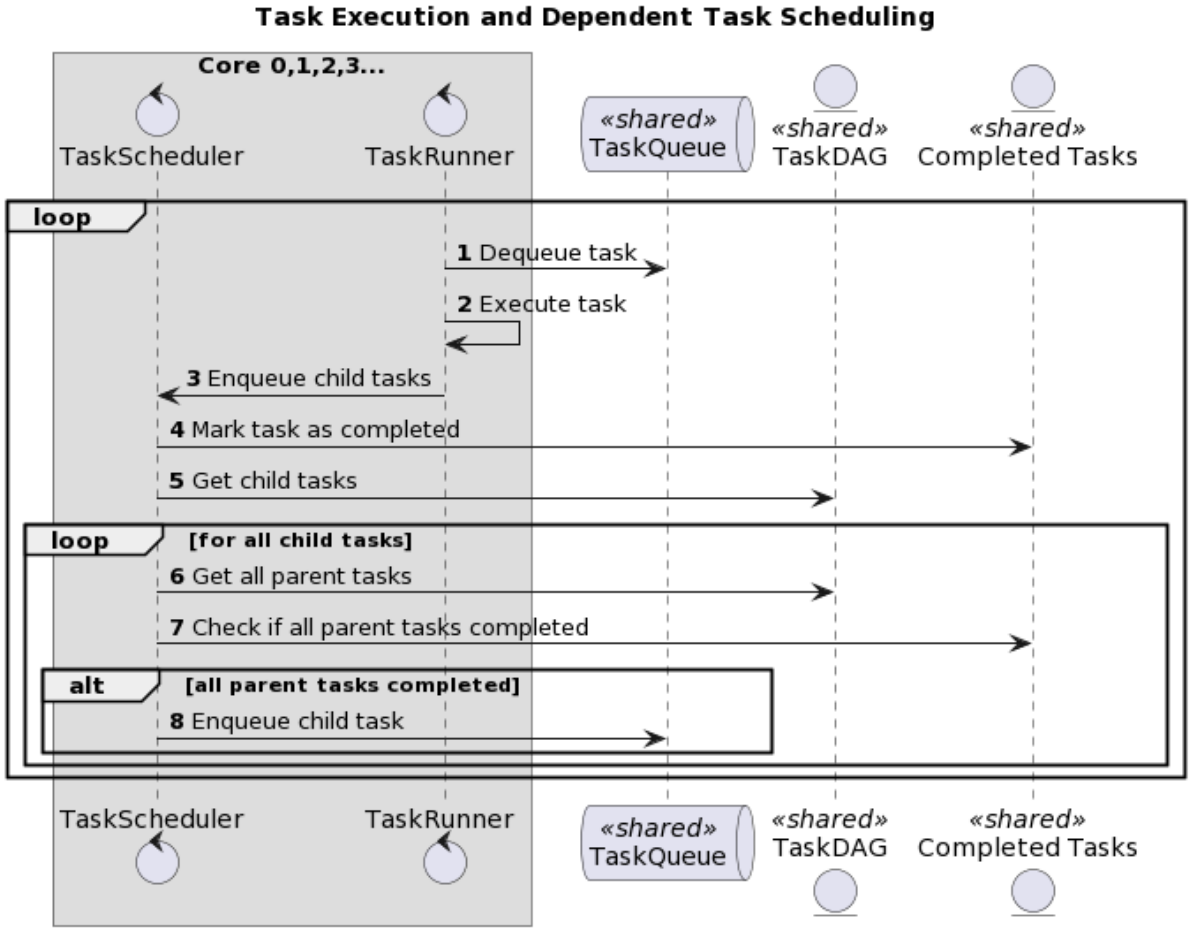
Detailed Operational Flow of Bootloader Initialization:



1. First Stage Bootloader initializes Main Bootloader
2. Bootloader Manager initializes CPU and Memory
3. Bootloader Manager loads Bootloader Configuration in shared memory.
4. Bootloader Manager asks Task Scheduler to setup task queue
5. Task Scheduler initializes the Task Queue in shared memory.

6. Task Scheduler initializes the Task DAG in shared memory.
7. Task Scheduler initializes the Completed Task List in shared memory and returns control to Bootloader Manager.
8. Bootloader Manager initializes Task Runner on current core and all other cores.

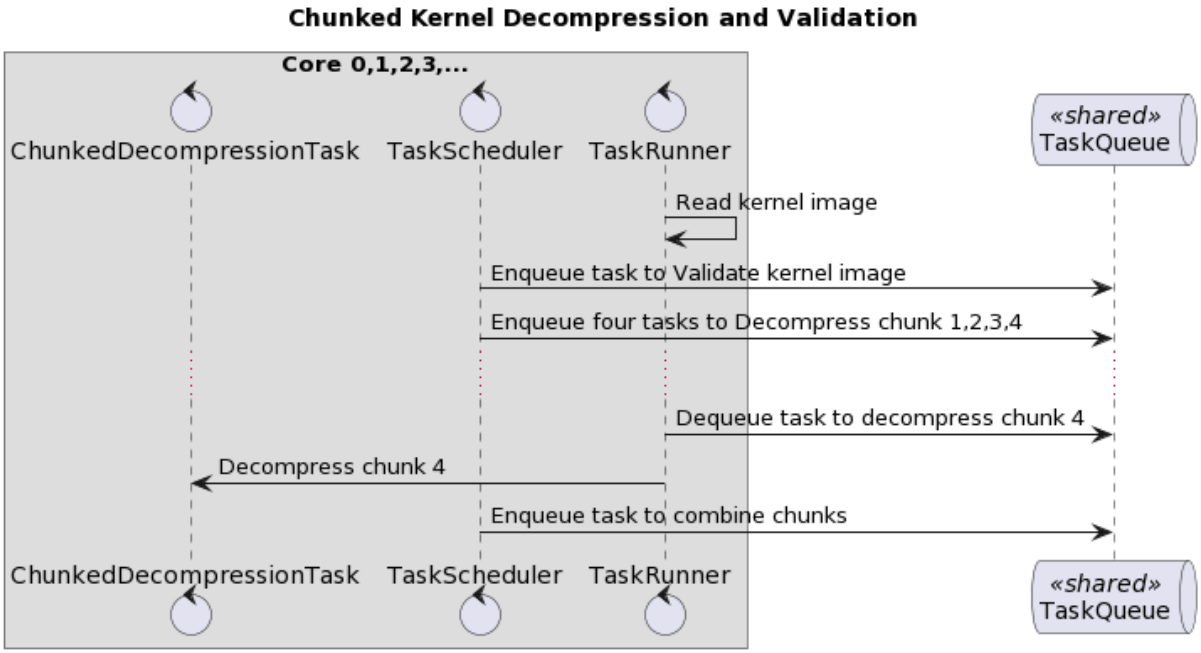
Detailed Operational Flow of Task Execution and Dependent Task Scheduling:



1. Task Runner on each core gets a task from the front of the queue
2. Task Runner executes the task
3. Task Runner asks Task Scheduler to enqueue

4. Task Scheduler adds the task to the Completed Tasks set.
5. Task Scheduler gets the child tasks of the task from the Task DAG.
6. For each child task:
    a. Task Scheduler gets all parent tasks of the child task from the Task DAG.
    b. Task Scheduler checks if all parent tasks of the child task have completed from the Completed Tasks Set.
    c. If all the parent tasks have completed, Task Scheduler enqueues the child task to the Task Queue.
7. Task Scheduler returns control to the Task Runner, which dequeues the next Task from the queue and the process repeats.

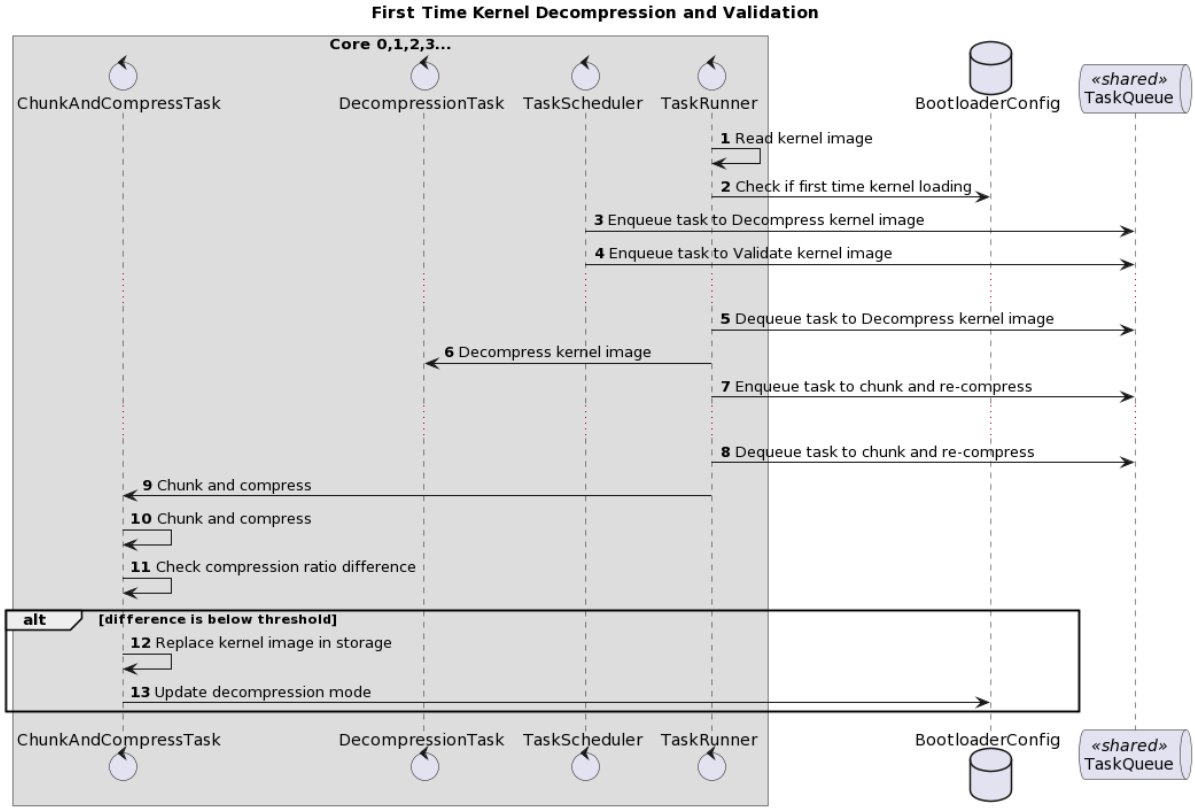
Detailed Operational Flow of Kernel Decompression and Validation:



Kernel Validation will be performed on compressed kernel image so that it can be performed in parallel with Kernel Decompression. Further, Kernel Decompression will be performed in chunks so

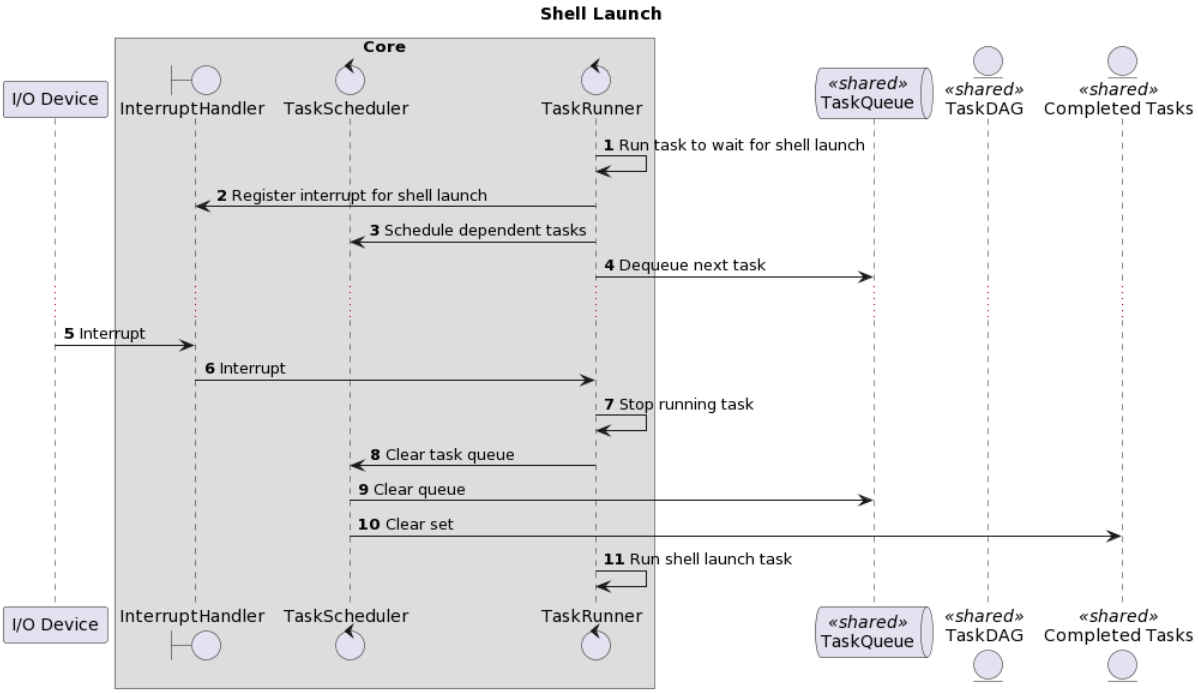
that it can also be parallelized. After all the chunks have been decompressed, they can be combined to get the Kernel Image.

To decompress in chunks, the Kernel image needs to be compressed in chunks as well. To achieve this, we will chunk and compress the Kernel image the first time it is loaded as below.



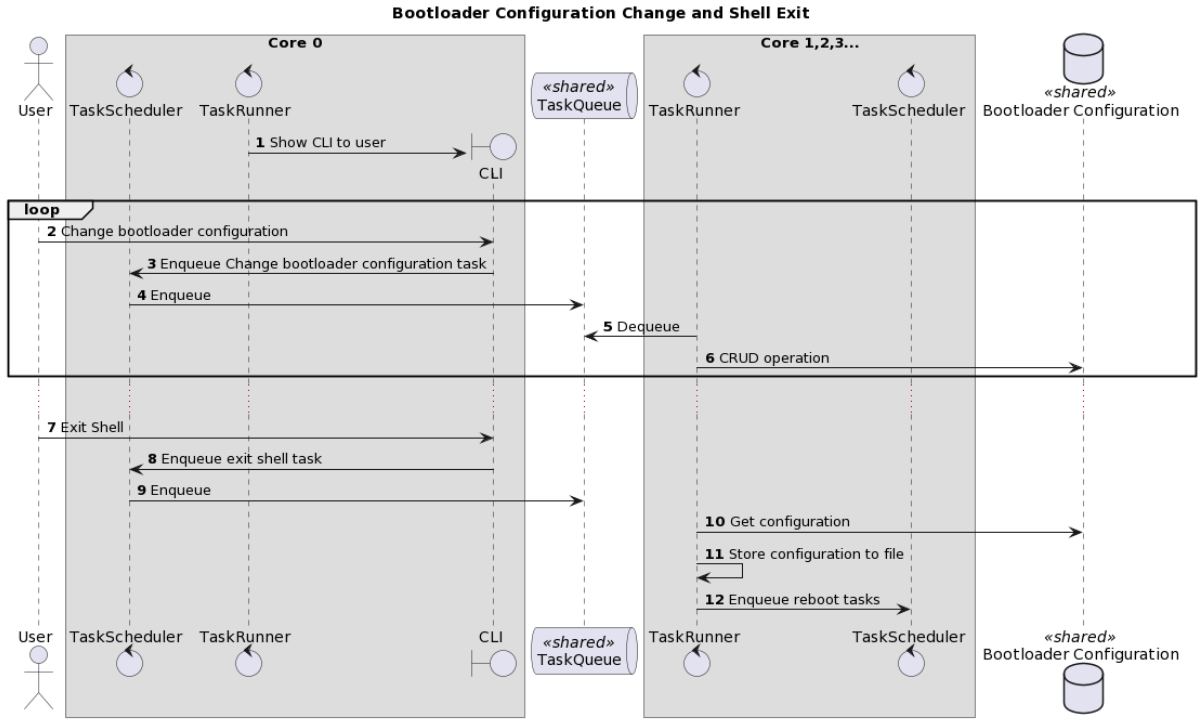
A Bootloader config property is used to decide Decompression Mode. Whenever a kernel image is loaded for the first time, Decompression Mode will be set to non-chunked. The Kernel Image will be decompressed in one task. Then, we will chunk and compress the kernel image and replace it in storage. A further check takes place before replacing the kernel image to ensure that compression ratio has not gotten dramatically worse such that Kernel reading time would increase by a lot.

Detailed Operational Flow of Shell Launch:



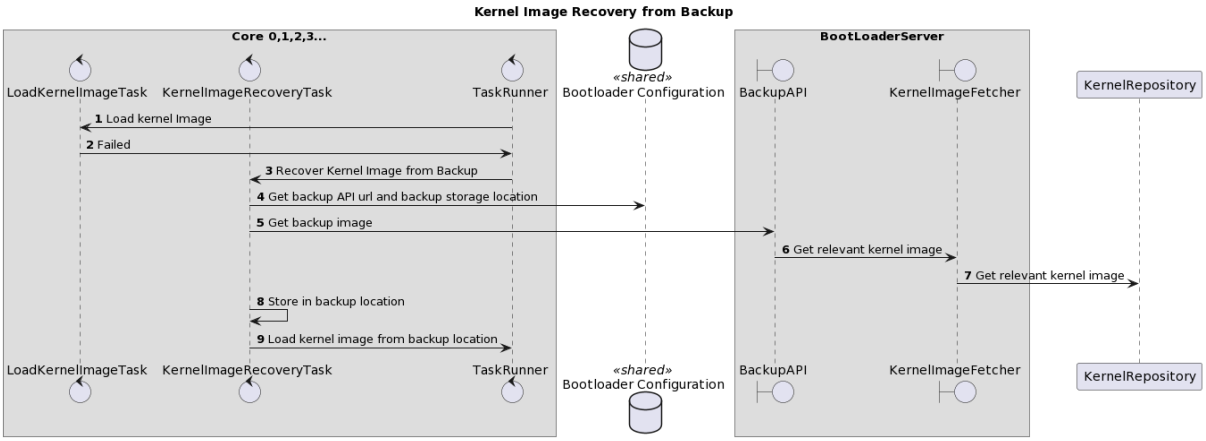
When task for waiting for shell input is executed, the core simply registers an interrupt with the interrupt handler to handle the request and moves on to process other tasks. When an interrupt is received from the User, the interrupt handler interrupts the Task Runner, which stops executing its running task and ask Task Scheduler to clear the Task Queue. Once the Queue and Completed Task Set is clear, the Shell Launch Task is executed.

Detailed Operational Flow of Bootloader Configuration Change and Shell Exit:



Configuration changes are made to the Bootloader Configuration stored in memory. When shell is about to be exited, these changes are persisted to storage at once.

Detailed Operational Flow of Kernel Image Recovery from Backup:

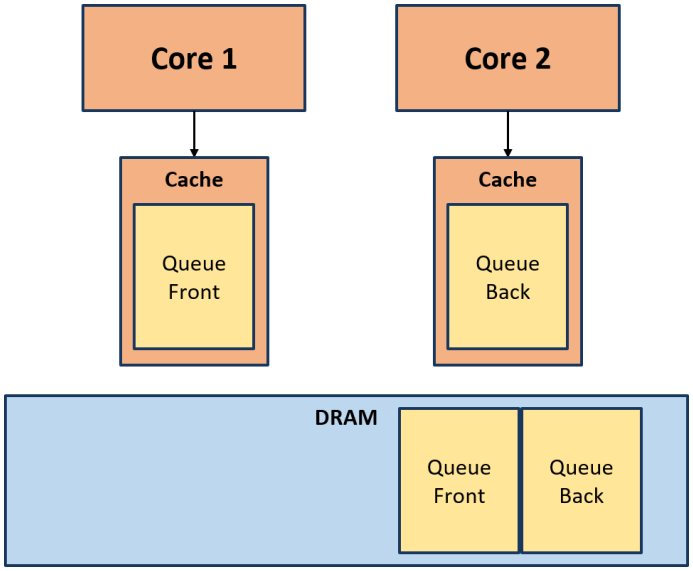


When Load Kernel Image Task fails due to storage failure or kernel corruption, we will load a backup

image using Backup API in the Bootloader server. Backup Image will be fetched by the Bootloader server from a Kernel Repository provided by the OS provider. The backup image will be stored in a backup location and the Load kernel Task will be retried.

Cache Aligned Task Queue Variables:

Task Queue variables will be cache aligned (padded to match core cache size). This is to prevent Core Cache Coherence issues.



For example, variable for queueFront and queueBack might be loaded into Cache together, when queueFront is updated by a core which is taking a Task from the queue, it will cause Cache of other cores to be evicted. Another core which might be accessing queueBack to insert Tasks to the queue, will have to read queueBack from memory instead of from cache.

By aligning Task Queue variables to Cache Size we will prevent such issues.

4. Modules

// A9. Module Specification

// C9-1. Is component specification sufficient to develop?

// C9-2. Is grouping appropriate in terms of module?

// C9-3. Is it appropriate to design dependencies between modules?

// C9-4. Is the work assignment appropriate?

The modules of our Bootloader can represented by a Layered Architecture to help in modifiability and ease of development. The Bootloader can broadly be divided into the following layers from **bottom to top**:

1. **Core Layer:** The bottom layer of our architecture. This is the most stable layer which provides the interfaces our bootloader uses to interact with the hardware devices and components extensively used in our implementation.
2. **Implementation Layer:** This layer contains all the components containing implementation needed to provide all the functionality of our bootloader.
3. **Application Layer:** This layer contains the implementation of all the features of our bootloader which are not needed to fulfill any functionality but are need to run the Bootloader applications.

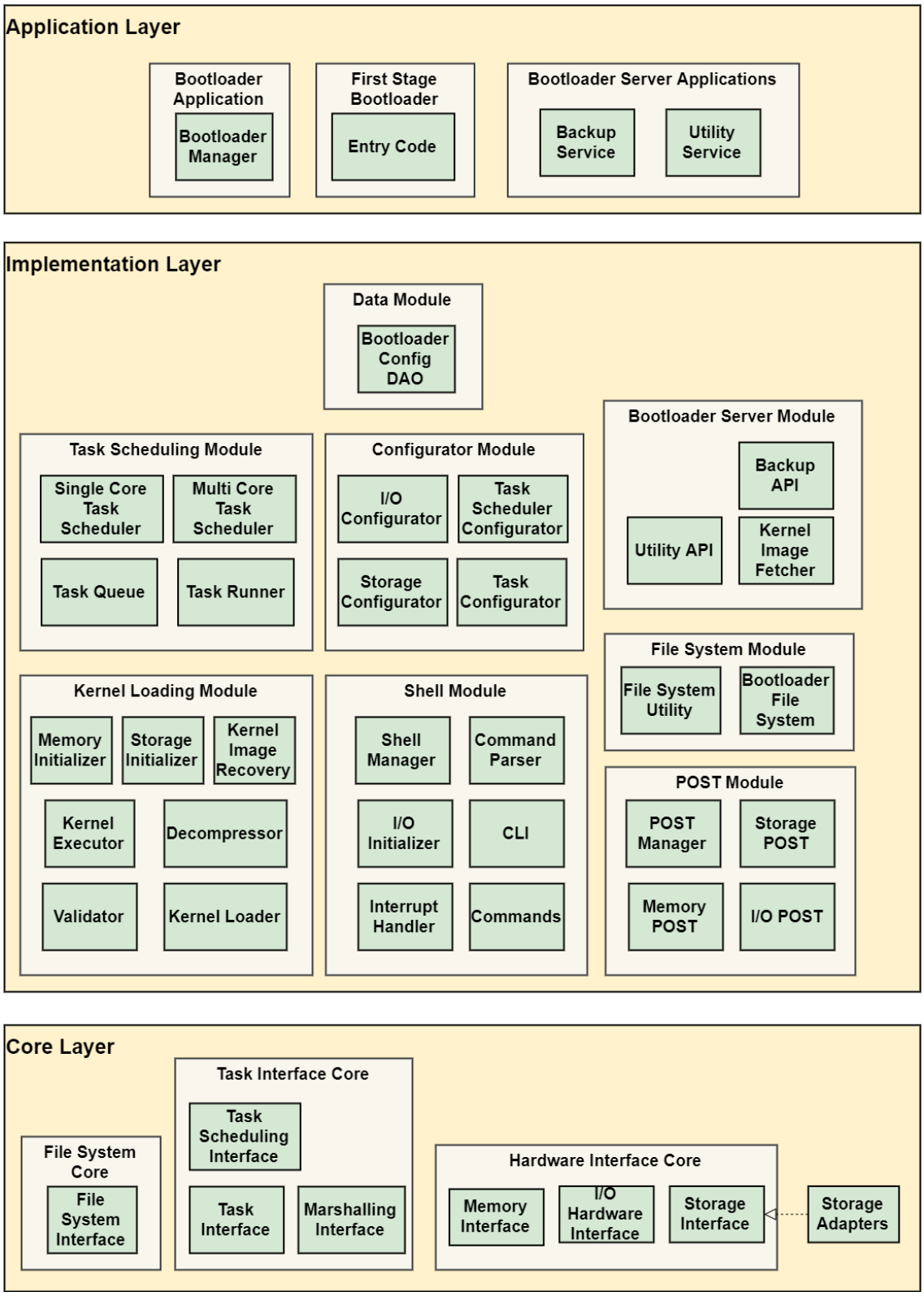


Figure 8: Module View

Core Layer:

Hardware Interface Core Module:

- **Memory Interface:** Stable Interface to interact with memory.
- **I/O Interface:** Stable Interface to interact with I/O devices. Its implementation is provided by I/O device developers in their device drivers.
- **Storage Interface:** Stable interface to interact with Storage devices.

Storage Adapters: Implementations of the Storage Interface. They are maintained as extension of the core hardware interface module. These are not accessible to the implementation, and are only injected into implementation by Storage Configurator.

Task Interface Core Module:

- **Task Interface:** Interface for Task which can be used to represent a single functionality as a Task which is to be performed.
- **Task Scheduler Interface:** Interface for a scheduler which decides how to run Tasks.
- **Marshalling Interface:** Interface for implementing Marshaling an Object to data and Un-marshalling it back into an object.

File System Core Module:

- **File System Interface:** Interface for implementing and using a file system.

Implementation Layer:

Kernel Loading Module:

- **Memory Initializer:** Initializes memory so that the bootloader can utilize it.
- **Storage Initializer:** Initializes storage devices so that they can be used by the bootloader.
- **Kernel Loader:** Loads the Kernel image from storage into memory. Implements the Task and Marshallable interfaces.

- **Kernel Image Recovery:** Recovers the kernel image from a backup location. Implements the Task and Marshallable interfaces.
- **Decompressor:** Used to decompress the kernel image. Implements the Task and Marshallable interfaces.
- **Validator:** Used to validate that the kernel image is genuine. Implements the Task and Marshallable interfaces.
- **Kernel Executor:** Sets up the kernel with the environment it needs and executes the kernel entry point.

Shell Module:

- **I/O Initializer:** Initializes I/O devices so that they can be used by the bootloader.
- **Interrupt Handler:** Handles interrupts to the bootloader, used to handle the interrupt to enter shell.
- **Shell Manager:** Manager for all other shell modules
- **Commands:** Implementations to perform the various commands supported by the shell. Each command implements the Task and Marshallable interfaces.
- **CLI:** Command Line Interface which the User interacts with to enter commands. Implements the Task and Marshallable interfaces.
- **Command Parser:** Parses commands provided to the shell and calls the appropriate Command. Implements the Task and Marshallable interfaces.

POST Module:

- **POST Manager:** Manager for POST modules. Implements the Task and Marshallable interfaces.
- **Storage POST:** Implementation for performing POST on Storage Devices. Implements the Task and Marshallable interfaces.
- **I/O POST:** Implementation for performing POST on I/O devices. Implements the Task and

Marshallable interfaces.

- **Memory POST:** Implementation for performing POST on Memory devices. Implements the Task and Marshallable interfaces.

File System Module:

- **Bootloader File System:** Implementation of a custom file system to be used by our bootloader.
- **File System Utility:** Implementation of a utility to create a Boot partition on a storage device and format it with Bootloader File System.

Data Module:

- **Bootloader Config DAO:** Data Access Object to access the Bootloader configuration.

Task Scheduling Module:

- **Multi Core Task Scheduler:** Implementation of the Task Scheduler Interface for multi core processing using the Task Queue.
- **Single Core Task Scheduler:** Implementation of the Task Scheduler Interface for single core processing.
- **Task Runner:** Components which takes tasks from the Task Queue and runs them
- **Task Queue:** Implementation of the queue used by Multi Core Task Scheduler and Task Runner.

Configurator Module:

- **Task Scheduler Configurator:** Module which injects either Single Core or Multi Core Task Scheduler into our Bootloader Manager. This configurator will use run time Dependency Injection.
- **Task Configurator:** Module which injects the various implementations of the Task interface from our Implementation Layer into the Task Runner in Multi Core Module or the Single Core Task Scheduler in the Single Core Module. This configurator will use compile time

Dependency Injection.

- **Storage Configurator:** Modules which injects the relevant Storage Device Adapters into the various Tasks that use the Storage Interface. This configurator will use compile time Dependency Injection.
- **I/O Configurator:** Modules which injects the relevant I/O Device Drivers into the various Tasks that use the I/O Interface. This configurator will use compile time Dependency Injection.

Bootloader Server Module:

- **Backup API:** API which interacts with Kernel Image Recovery to enable download of backup kernel image.
- **Kernel Image Fetcher:** Implementation of micro service which fetches kernel image from external kernel image repositories.
- **Utility API:** External API to provide File System Utility.

Application Layer:

First Stage Bootloader:

- **Entry Code:** Entry code which is executed by Boot ROM and loads our Main Bootloader

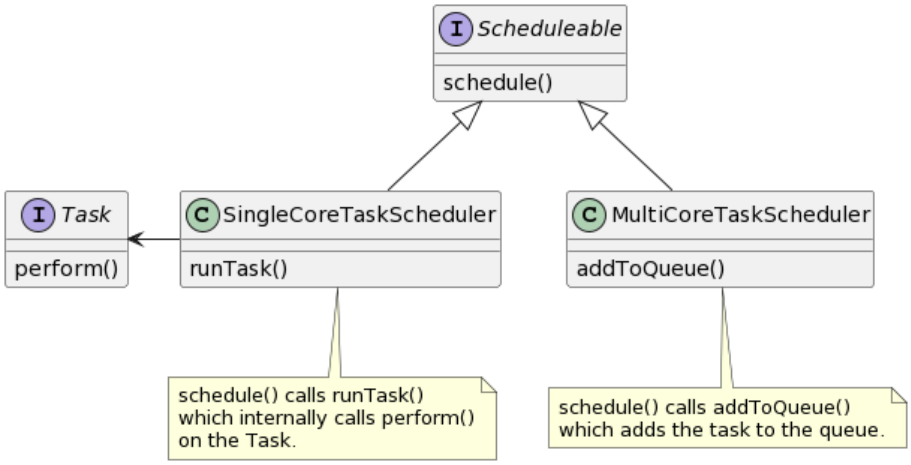
Bootloader Application:

- **Bootloader Manager:** Main manager for our bootloader. It contains implementations all the first level modules needed to run our bootloader such as Memory Initializer, Task Scheduler Task Runner and Bootloader Config DAO.

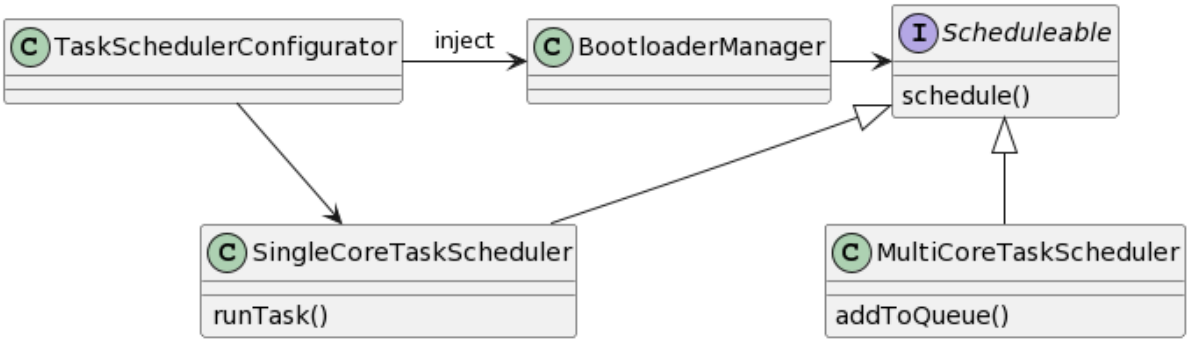
Bootloader Server Module:

- **Backup Service:** Service for the Backup API and Kernel Image fetcher.
- **Utility Service:** Service for the Utility API which uses the File System Utility.

Task Scheduler Implementation:



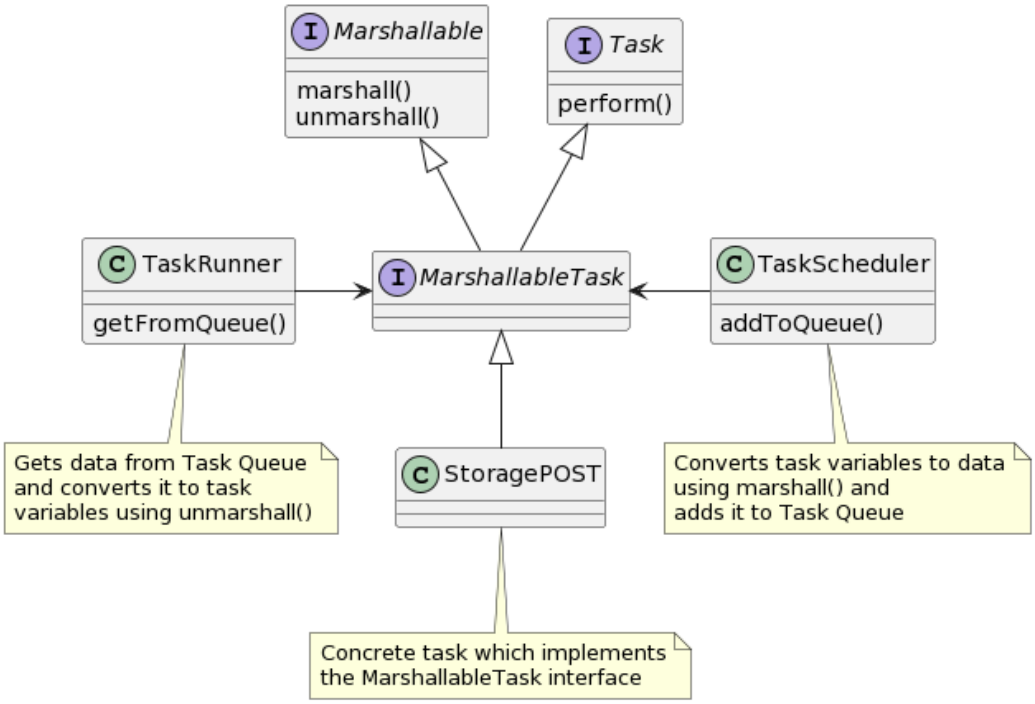
External components using the Task Scheduler call the schedule() function which takes a Task as an argument. Internally, schedule() adds the task to Task Queue for Multi Core Task Scheduler and runs the task for Single Core Task Scheduler.



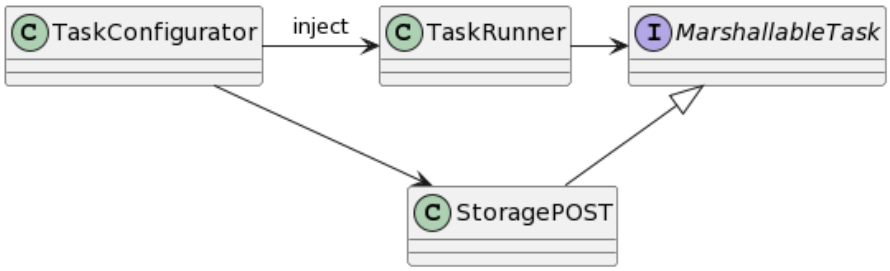
As shown above, the Task Scheduler Configurator injects the relevant Task Scheduler into Bootloader Manager.

Multi Core Task Scheduler contains implementation of a **directed acyclic graph** representing task dependencies, which it loads from a configuration file at run time.

Task Implementation:



Tasks are converted to data to be stored in memory using marshalling. Each Task also implements a Marshalling interface (combined into the Marshallable Task interface), which is used by the Task Scheduler to marshal the Task information into data before inserting it into the Task Queue and by the Task Runner to un-marshal the task information back into the class Object.

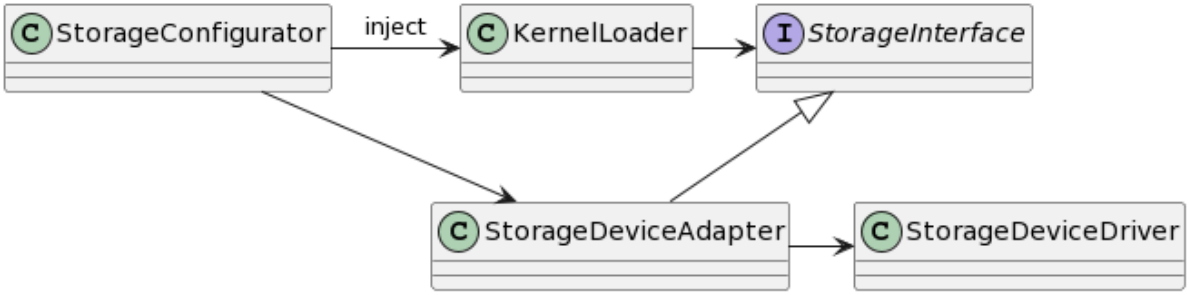


Similar to Task Scheduler Configurator, Task Configurator injects dependency of concrete Task implementations to the Task Runner so that it can run the task.

Hardware Interface Implementation:

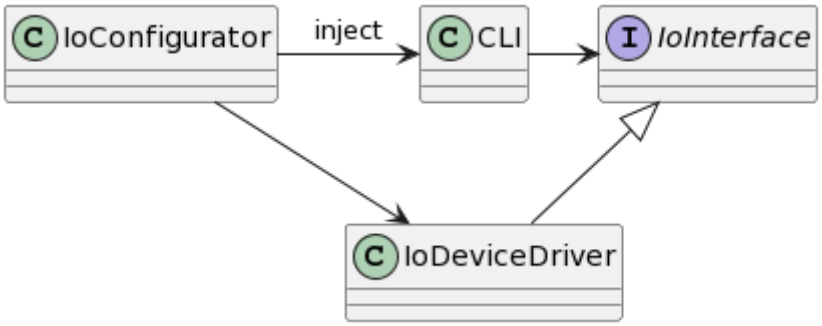
Hardware interfaces are separated into Storage Interface, I/O Interface and Memory Interface.

Storage Interface Implementation



For Storage Interfaces, we will use Adapter pattern to adapt Storage Device Drivers to our Storage Interface. This Adapter will then be injected into Tasks that use the Storage Interface such as Kernel Loader by the Storage Configurator.

I/O Interface Implementation



For I/O interfaces, implementation of the I/O interface will be inside the I/O Device Driver, so Adapter will not be needed in the Bootloader. Effectively, the Adapter to the I/O Interface will be implemented by I/O device developers.

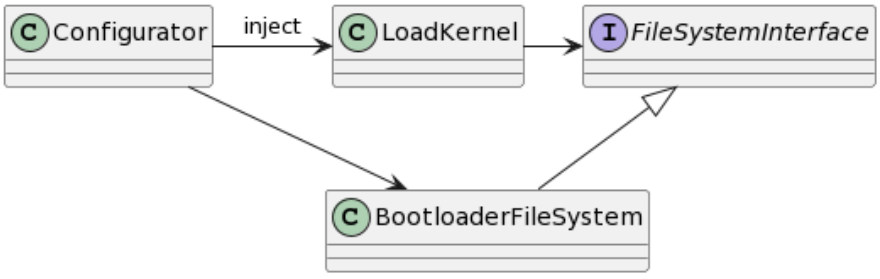
File System Implementation:

A custom simplified Bootloader File System will be used by the Bootloader. The file system will not support directories and store files in contiguous blocks to improve retrieval time.

A separate partition will be maintained in the Storage Device to contain Kernel Image which will be formatted with this file system.

A Utility will also be provided online and within the Bootloader as a Shell Option to create the Boot partition on a Storage Device and format it with the Bootloader File System.

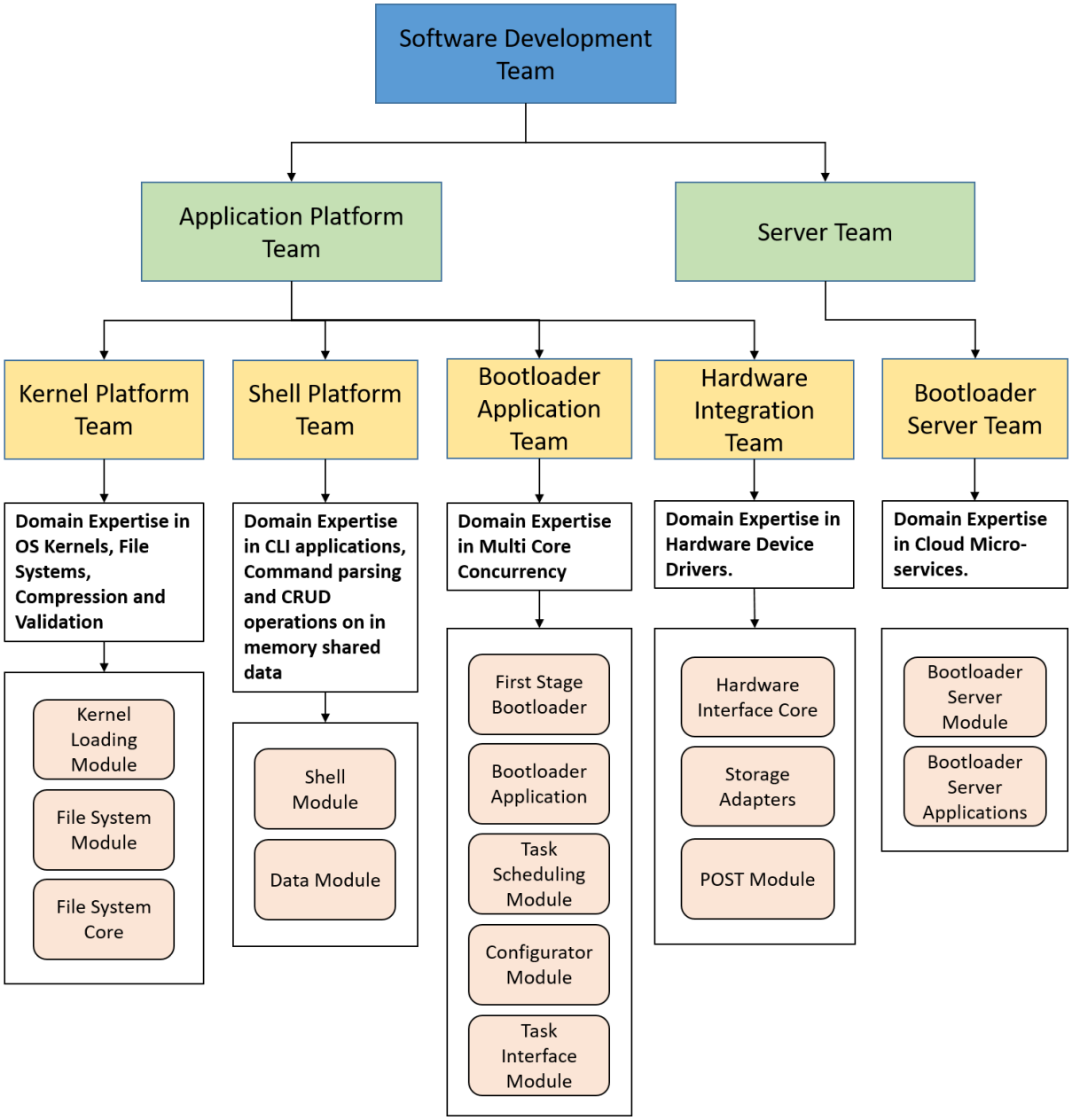
To ensure flexibility to allow for other file systems to be supported in the Future, Implementation Layer will depend on a File System interface, while the Bootloader File System will be injected into Tasks that need it.



Shell Command Implementation:

Each Shell Command will implement the Task interface and will be maintained as a Task. If Shell Commands need to be parallelized, this will be managed through Dependent Tasks of the main Shell Command Task.

Work Assignment



Appendix

A. Domain Model

Entity-Control-Boundary pattern is used to describe the conceptual model of Boot Loader.

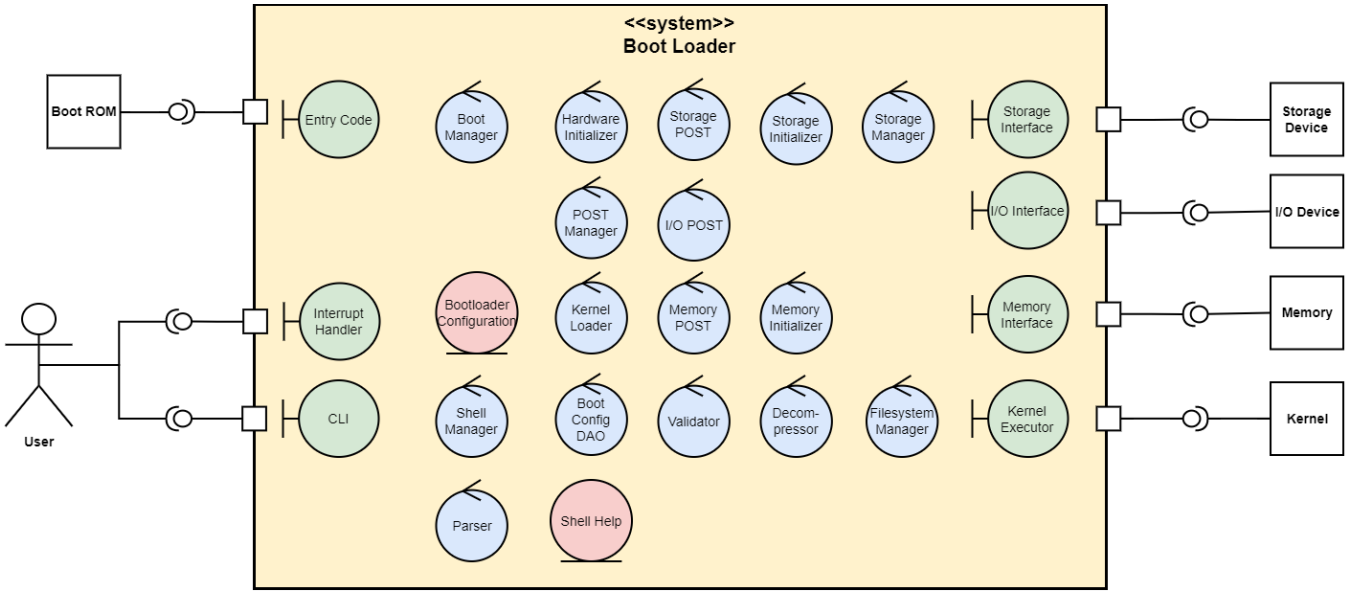


Figure 9: Boot Loader - Domain Model (Conceptual Model)

Boundary Components:

- **Entry Code**: Entry code of the bootloader that is executed by Boot ROM code.
- **Interrupt Handler**: Handler that takes interrupts from user, used to take interrupt to launch shell.
- **CLI**: Command Line Interface created by the Bootloader for the user to operate shell.
- **Storage Interface**: Hardware Interface used to interact with storage devices
- **I/O Interface**: Hardware Interface used to interact with I/O devices
- **Memory Interface**: Hardware Interface used to interact with Memory
- **Kernel Executor**: Interface that executes the kernel entry point at the end of booting process.

Control Components:

- **Boot Manager**: Manager for booting process. It interacts with Hardware Initializer, POST Manager and Kernel Loader to perform booting operations.
- **Hardware Initializer**: Manages initialization of memory and storage hardware.
- **Storage Initializer**: Initializes storage devices.
- **Memory Initializer**: Initializes memory.
- **POST Manager**: Manages Power on Self-Test.
- **Storage POST**: Tests storage devices are working as expected.
- **Memory POST:** Tests memory is working as expected.
- **I/O POST:** Tests I/O devices are working as expected.
- **Kernel Loader:** Loads the kernel into memory from storage
- **Decompressor:** Decompresses the kernel image
- **Validator:** Validates kernel image signature.
- **Storage Manager**: Manages storage device related actions of the boot loader like reading kernel image.
- **File System Manager**: Initializes and manages the file system for the storage device used by Storage Manager and provides storage addresses for file paths to the storage manager
- **Shell Manager**: Manages the shell operations of the bootloader.
- **Parser**: Parses CLI input to shell commands
- **Command**: Performs tasks to fulfill shell commands.

Entity Component:

- **Bootloader Configuration**: Configuration information of the bootloader like boot device,

kernel image details, configured input and display devices, interrupt duration etc. This information will be persisted in storage and the bootloader will need to load it into memory when it starts, but is represented as a part of the system since it is the information store of the bootloader.

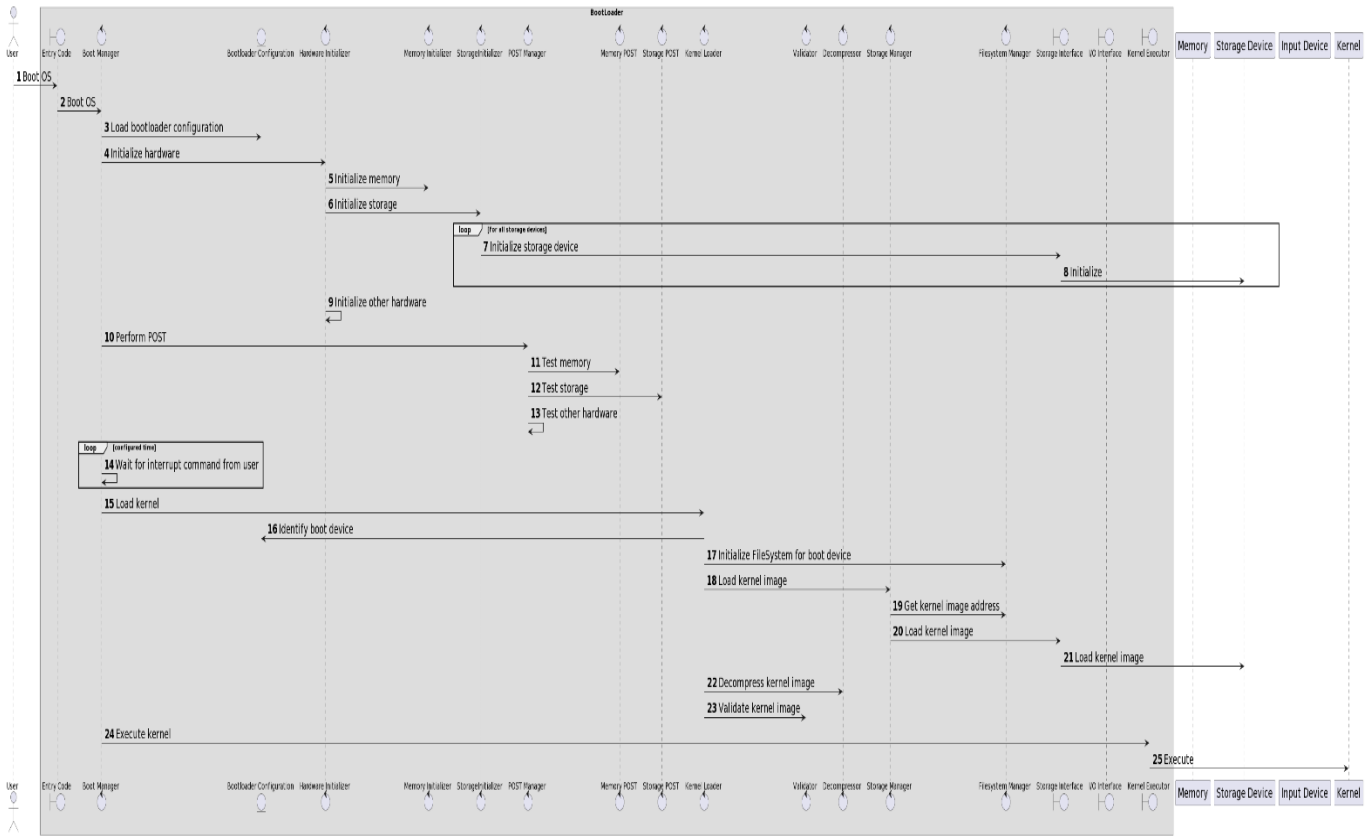


Figure 10: UC_01 - Boot OS Sequence Diagram

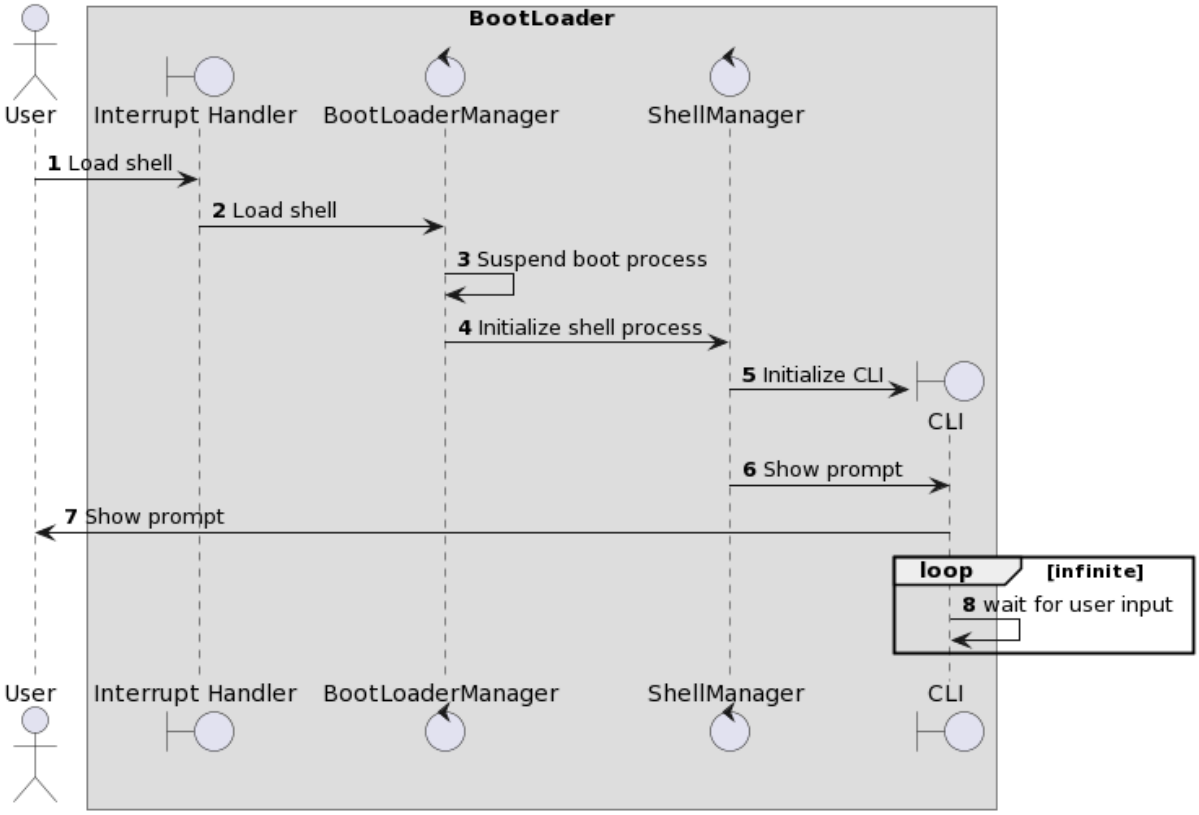


Figure 11: UC_02: Load Shell Sequence Diagram

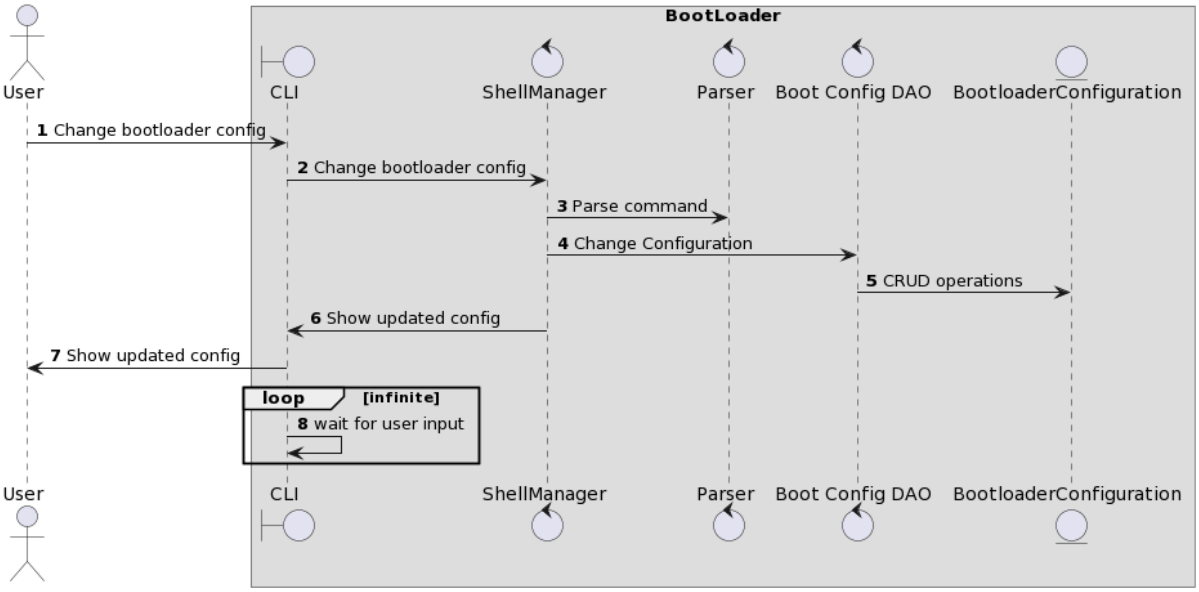


Figure 12: UC_04 – Change Bootloader Configuration Sequence Diagram

B. Quality Scenarios

Category	Sub Category	Quality Scenario	
Performance	Time	**QS_01**	**Boot OS time should be minimized**
		Boot time is the most important quality of our system. We need to ensure our booting is optimized as much as possible **Measure:** Response Time = [Time when user presses power button] - [Time when bootloader executes kernel]	
		QS_02	**Bootloader Configuration time**
		Time taken to show bootloader configuration and apply changes to it should be minimized as much as possible **Measure:** Response Time = [Time when command response is displayed to user] - [Time when command is requested by user]	
	Resource Utilization	**QS_03**	**Bootloader Storage and Loading Efficiency**
		Bootloader binary size should be minimized so that it can run on devices that have limited storage capacity and can be loaded quickly. Measure: [Size of binary in bytes]	
Maintainability	Modifiability	**QS_04**	**Bootloader should be easily updatable**
		Developers should be able to easily push updates to the bootloader code in market released devices in a reasonable time if needed Measure: [Release cycle time]	
		QS_05	**Multi Core Modifiability**
		We will use multi-core setup to improve QS_01. No. of cores in the CPU can differ for different devices. It should be easy for developers to change functionality without affecting multi core support and also add support for new type of multi core setup. **Measure:** [Development cost]	
		QS_06	**Storage Modifiability**

		Since we have a product line of devices we need to support they can have different storage devices where the kernel is stored. It should be easy and fast for developers to add support for a new storage device. Different storage devices can also be formatted by different file systems, this also needs to be managed by the bootloader. **Measure:** [Size of affected module]	
		QS_07	**I/O device Modifiability**
		We will need to support various I/O devices for our different devices. It should be easy to add support for new I/O devices and changes to functionality of one I/O should not have an impact on others. **Measure:** [Size of affected module]	
		QS_08	**Microprocessor Architecture Modifiability**
		Bootloader should be able to run on different microprocessor hardware and changes to code for one architecture should not affect working of other architectures **Measure:** [Size of affected module]	
		QS_09	**Shell Modifiability**
		Shell should be modifiable relatively independently of main booting code and changes in shell should not affect booting code. **Measure:** [Size of affected modules]	
	Testability	**QS_10**	**Bootloader changes should be testable without needing much hardware**
		Developers should be able to test changes in bootloader code without using physical hardware. **Measure:** [Count of external components needed for testing]	
Usability	**User Interface**	**QS_11**	**Viewing and changing bootloader configuration should be easy and intuitive**
		User should easily be able to view and change bootloader config and should not require much knowledge about bootloaders to change these settings. **Measure:** [Count of inputs needed to change config]	
	User Error	**QS_12**	**User should not be allowed set invalid bootloader configuration**

	Protection	User may make a mistake in setting bootloader configuration and enter invalid configuration such as boot device. System should prevent this. **Measure:** [Count of invalid inputs accepted]	
Reliability	**Recoverability**	**QS_13**	**Recovery of Kernel Image**
		System should have a recovery mode to recover Kernel image if it is corrupted **Measure:** [Recovery time]	
	Availability	**QS_14**	**Available despite Hardware Failure**
		Storage or peripherals connected to the device may fail. System should be able to boot the kernel despite these failures as much as possible. **Measure:** [Downtime]	

C. Quality Scenario Analysis

Category	Sub Category	Quality Scenario		Impor-tance	Diffi-culty	Quality Requirement
Performance	**Time**	QS_01	Boot OS time should be minimized	H	H	NFR_01, QA_01
		QS_02	Bootloader Configuration Time	M	M	NFR_02
	Resource Utilization	QS_03	Bootloader Storage and Loading Efficiency	M	M	QA_06
Maintainability	**Modifiability**	QS_04	Bootloader should be easily updatable	M	L	Ignore
		QS_05	Multi Core Modifiability	H	H	QA_02
		QS_06	Storage modifiability	H	H	QA_03
		QS_07	I/O device modifiability	H	H	QA_04
		QS_08	Microprocessor Architecture modifiability	M	L	Ignore
		QS_09	Shell Modifiability	H	M	QA_05
	Testability	QS_10	Bootloader changes should be testable without needing much hardware	L	M	Ignore
Usability	**User Interface**	QS_11	Viewing and changing bootloader configuration should be easy and intuitive	L	L	Ignore
	User Error Protection	QS_12	User should not be allowed set invalid bootloader configuration	L	L	Ignore
Reliability	**Recoverability**	QS_13	Recovery of Kernel Image	H	L	NFR_03 (Kernel Image Recovery Time)
	Availability	QS_14	Available despite Hardware Failure	H	L	

QS_01 Device boot time should be within industry standards

This is the most important quality of our system. We need to optimize booting time as much as

possible as that is the main use case of our system and a slow booting time will lead to a bad user experience. Therefore it is a **high priority quality attribute**. If our booting time is greater than industry standards, our bootloader will fail, therefore we can also maintain it as a **non-functional requirement**.

QS_02 Bootloader Configuration Time should be minimized

Bootloader configuration time is important for user experience when the user is operating the shell. Configuration should be changed in a reasonable time but since this use case will not be used very often we do not need to optimize it that much. Therefore, we can maintain this as a **non-functional requirement**.

QS_03 Bootloader Storage and Loading Efficiency

Bootloader storage needs to be optimized as in many devices Boot ROM loads bootloader from MBR in the first block of memory, which is 512 bytes. This constraint needs to be managed. Further, greater the bootloader size, longer it will take for the bootloader to load into memory and bootloader performance will also be affected. However, bootloader size does not directly affect any major use case of our system. Therefore, it is a **low priority quality attribute.**

QS_04 Bootloader should be easily updatable

Bootloader updates can be easily pushed either by having the OS download them over the air or by implementing a process in the bootloader itself to check for updates. Therefore, we can **ignore** this quality scenario for now.

QS_05 Multi Core Modifiability

For improving booting performance (QA_01), we will need to use multiple cores in our bootloader. We need to manage how our bootloader will handle different core configurations (single core, two core, four core etc.) efficiently and how to maintain our code to easily support any core configuration. This can be complex to achieve and is also important for our bootloader performance. Therefore, this is a **high priority quality attribute**.

QS_06 Storage Modifiability

Since we have a product line of devices we need to support they can have different storage devices

where the kernel is stored. This is both difficult to achieve because of different storage interfaces provided by different devices and also very important for our system. Therefore, this is a **high priority quality attribute**.

QS_07 I/O Modifiability

Like storage modifiability, our products can have various display and other I/O devices associated to it which we need to support. This is again tough due to each device having its own interface definition and also important for our system. Therefore, this is a **high priority quality attribute**.

QS_08 Microprocessor Architecture Modifiability

We may need to support various microprocessor architectures in the future so we need to write our code such that it is architecture independent. This can be easily achieved as we will be using C to write our code and can avoid using architecture specific code in our core functionality and have extensions for architecture specific code. Therefore, we can **ignore** this scenario.

QS_09 Shell Modifiability

Shell options and shell code can change more often than bootloader code. This should not have an impact on main booting code and should be easily released independent of booting code. This is somewhat important and also requires careful maintenance of shell and boot code. Therefore, this is a **low priority quality attribute**.

QS_10 Bootloader changes should be testable without needing much hardware

This is a useful quality as developers may not always have access to all boards and devices for testing. A virtual environment where the function of these boards and devices can be mimicked will lead to quicker and better development and testing. This can be achieved through function mocking and through sandboxing environments. As this is not very important, we can **ignore** this for now and take it up in further phases of the project.

QS_11 Viewing and changing bootloader configuration should be easy and intuitive

QS_12 User should not be allowed set invalid bootloader configuration

These two user experience scenarios can both be ignored for now. We have provided a help section

in our shell for helping the user change bootloader configuration and can check invalid user in the parser. In the future, a GUI can also be provided instead of the shell if needed. Since these qualities are fulfilled by our current system we can **ignore** them.

QS_13 Recovery of Kernel Image

QS_14 Available despite Hardware Failure

We can ignore other hardware failure as it is either not critical to our main booting use case (like if I/O device fails), or it is critical to the extent that kernel itself may not be able to run (like if memory or CPU fails).

Thus we can consider recovery of kernel image due to either kernel image corruption or kernel storage device failure. This can be achieved by configuring backups from where kernel image can be loaded. Since this is important but not very difficult, we can have this as a **non-functional requirement.**

D. Candidate Architectures

// A6. Candidate Architecture Design

// C6-1. Are quality analysis and solution candidate appropriate?

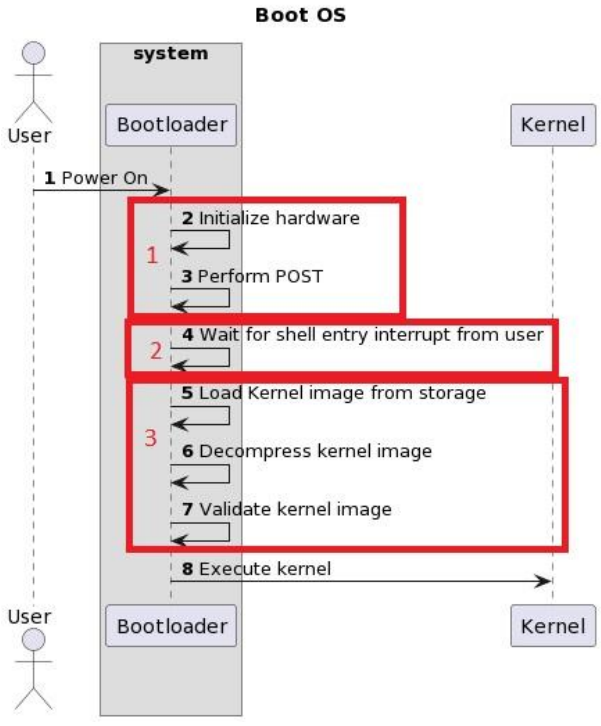
// C6-2. Are performance analysis and solution candidate appropriate?

// C6-3. Are modifiability analysis and solution candidate appropriate?

In this section, we will explore Candidate Architectures targeting each Quality Attribute and Non-Functional Requirement we described in Section 2.2 and 2.3

D1. QA_01 Boot OS Time Optimization

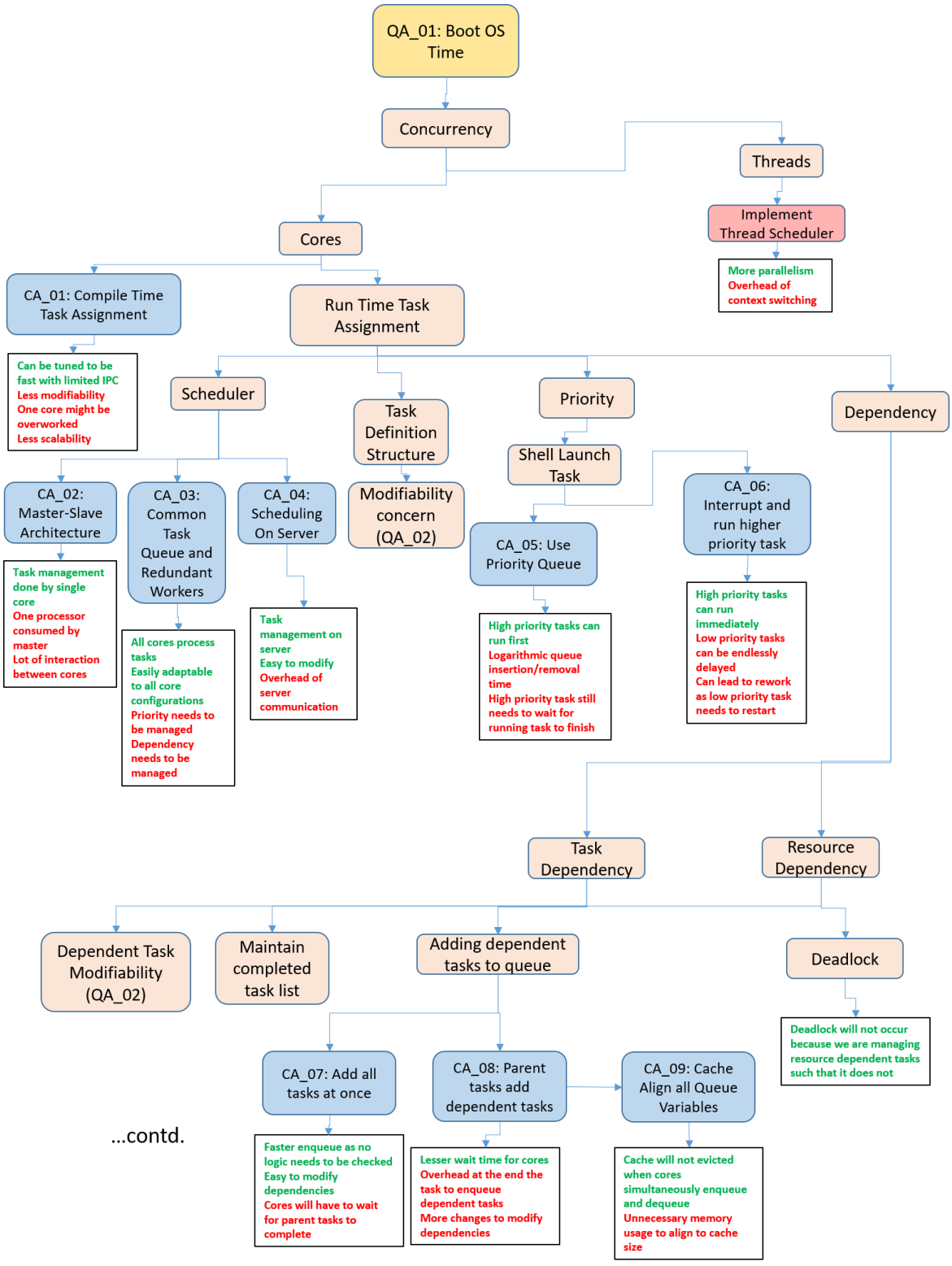
Booting the OS (UC_01) can roughly be divided into three mostly independent parts as shown below:



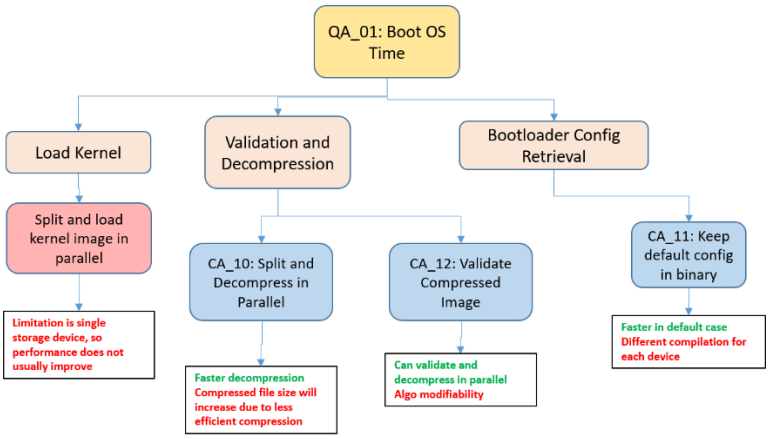
To optimize booting we can consider performing these 3 parts concurrently.

We can also further parallelize by performing tasks within these 3 groups concurrently.

Further, we can think about dividing the HW Initialization and POST so that tasks dependent on POST can be performed in parallel with POST.



...contd.



Concern: Concurrency/Parallelism Model

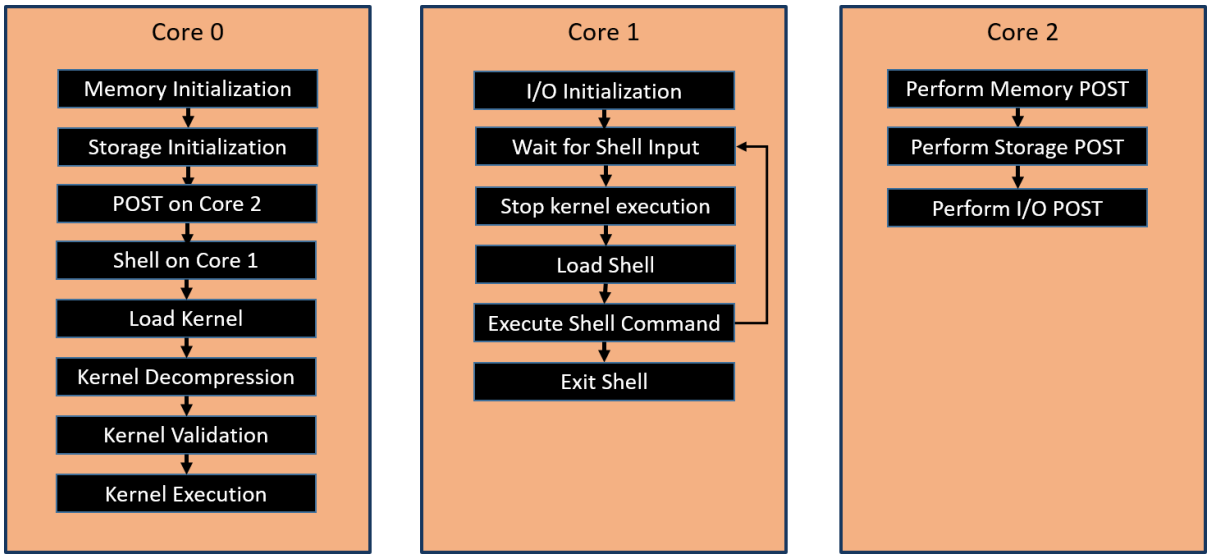
D1.1. CA_01: Compile Time Task Assignment

For multi-core processing, one strategy is to define workload for each core at compile time.

At run-time the process starts on a single core. We can divide the processes to run on each core, when a particular process needs to be started, the running core simply loads the process on a different core and starts the process

Each process proceeds sequentially through its own tasks on single core in parallel.

One possible deployment of cores with the processes they will execute is shown below where booting and shell tasks are parallelized



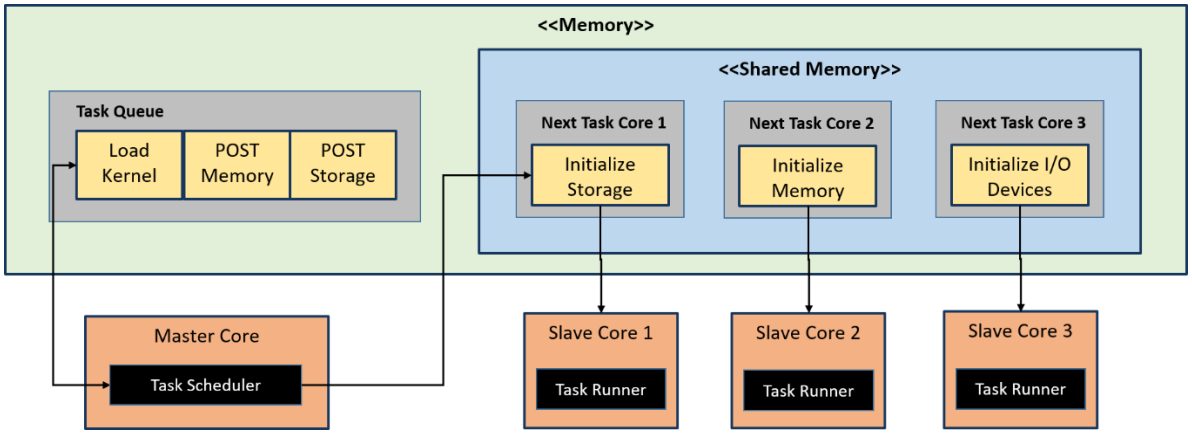
Here a single core performs its defined tasks sequentially and it's communication to other processes is limited to start the processing on another core or stopping it.

When core count decreases the workload of two cores can be combined into one core, instead of loading the process on another core, the running core simply starts the process.

D1.2. CA_02: Master Slave Architecture

Conflicts with CA_01

We can have one master core, which assigns and manages tasks on the slave cores. Master core can communicate with slaves through some inter processor communication method like shared memory or message passing.



As shown in the diagram, the master core takes care of scheduling tasks on the slave cores, while slave cores can perform smaller tasks and communicate results back to the master core. The master will then assign a new task to the slave.

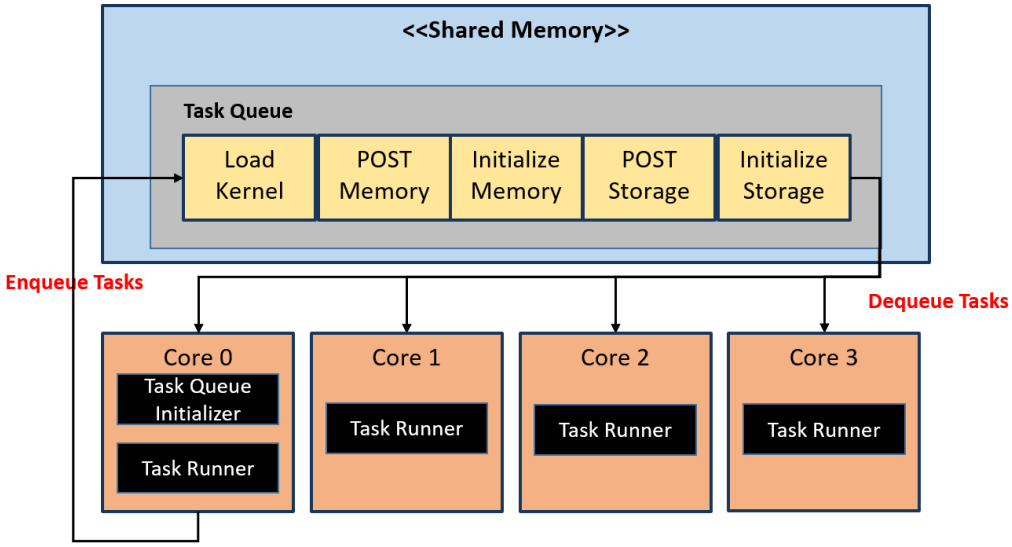
D1.3. CA_03: Common Task Queue and Redundant Workers

Conflicts with CA_01 and CA_02

We can also have a common task queue in memory, and each core can pick up tasks from the queue, execute them and enqueue dependent tasks to the queue.

Initial queue setup can be done by the first core when the bootloader is launched and then all

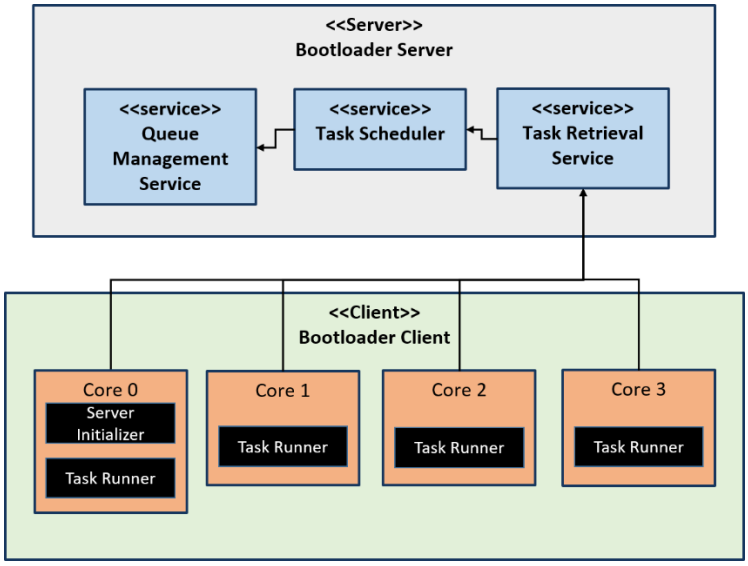
cores listen to the queue for tasks as shown below.



D1.4. CA_04: Scheduling on Server

Conflicts with CA_01, CA_02 and CA_03

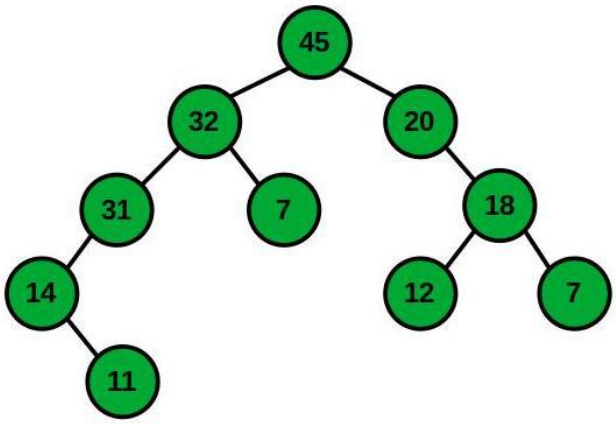
Task scheduling can also be offloaded to the server. The first core initializes connection to the server and sets up the other cores. Then all cores retrieve tasks from the server and perform them.



Concern: Task Priority

D1.5. CA_05: Use Priority Queue for Tasks

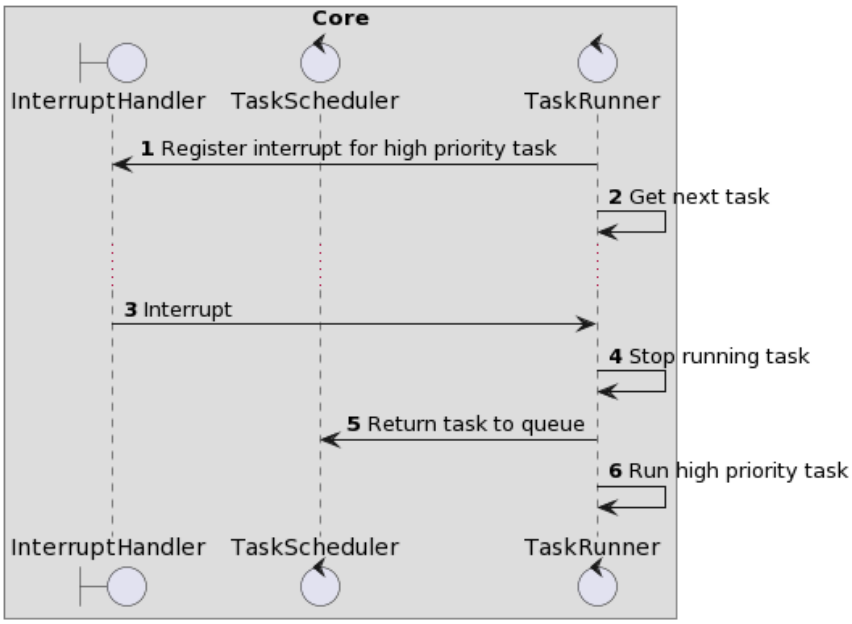
Instead of a queue, we can use a priority queue for tasks. Each task can be assigned a priority at compile time, and higher priority tasks will be picked earlier from the queue.



D1.6. CA_06: Interrupt and run higher priority task

Conflicts with CA_05

When a high priority task needs to be added to the back of the queue, instead of adding it to the queue the core can simply interrupt the running process on that core or interrupt one of the other cores to handle the high priority task first and return the currently running task to the end of the queue.

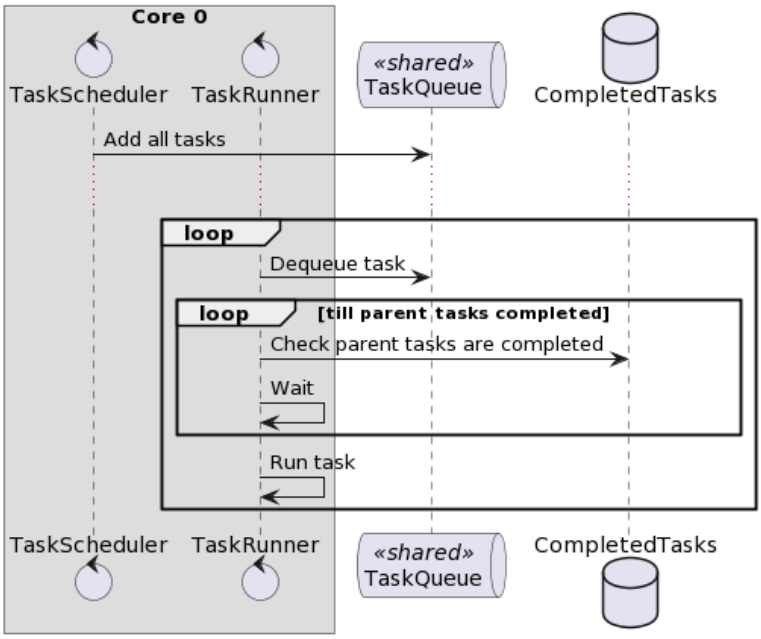


Concern: Task Dependency - Adding Tasks to Queue

D1.7. CA_07: Add all tasks to queue at once

One option is to add all tasks to the queue at once at the start of the booting process and then manage dependencies by checking if parent tasks have completed before running child tasks.

Each task can be given an ID and Completed Task IDs can be maintained in a data structure such as a Hash Set.

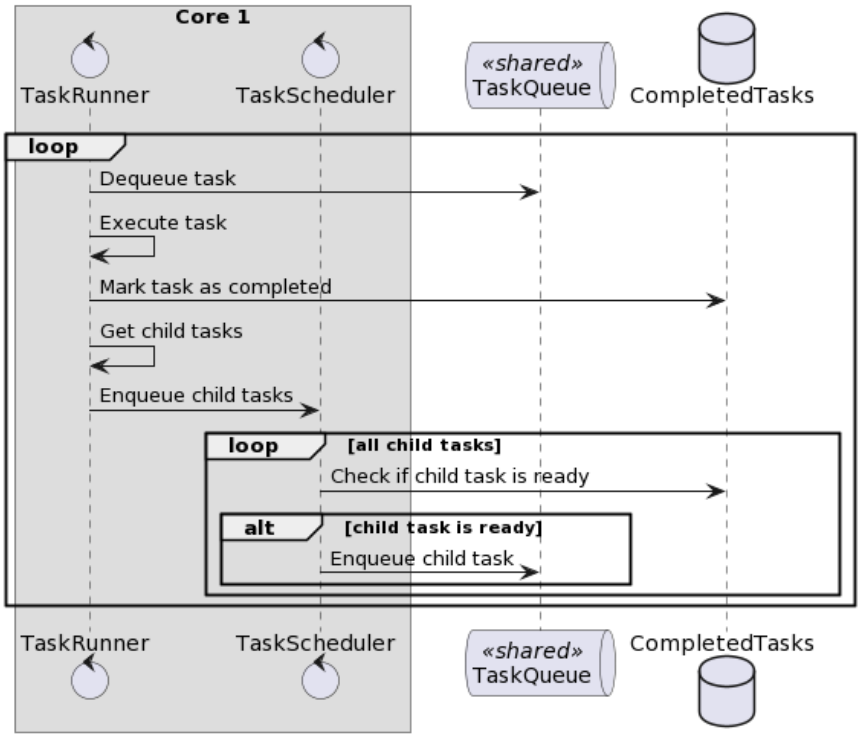


D1.8. CA_08: Parent tasks add dependent tasks

Conflicts with CA_07

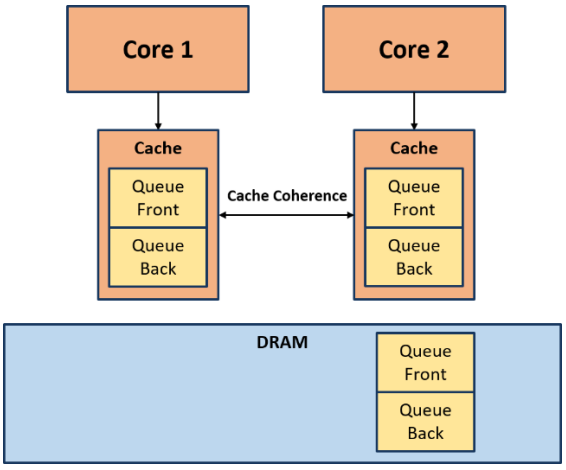
Each task at the end of its execution will check the dependent child tasks it needs to execute and add them to the queue.

For some child tasks, there can be many parent tasks that need to be run before that particular task can run. For example, Kernel Loading requires both Memory and Storage to be initialized. For these tasks, the parent tasks needs to check all other parent tasks have been completed before enqueueing the child task.



D1.9. CA_09: Cache Align all Queue Variables

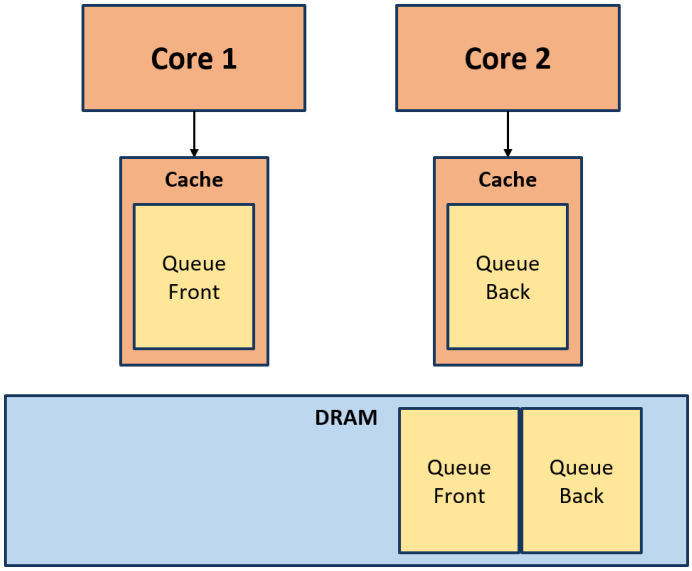
One problem that happens when multiple queues enqueue and dequeue at the same time as in CA_08 is that they would cause cache coherence issues, forcing core caches to be evicted.



For example, variable for queueFront and queueBack might be loaded into Cache together, when

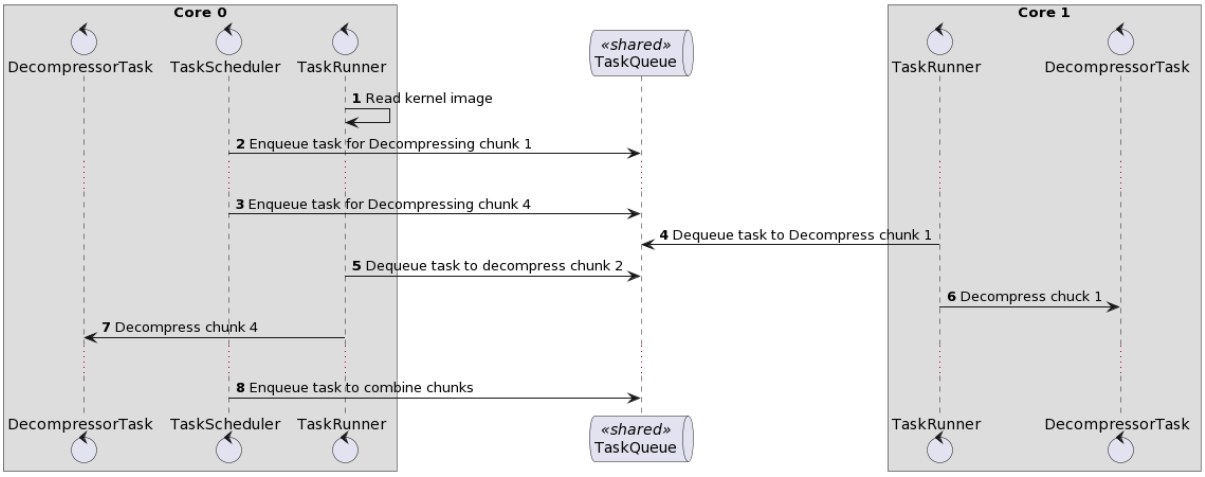
queueFront is updated by a core which is taking a Task from the queue, it will cause Cache of other cores to be evicted. Another core which might be accessing queueBack to insert Tasks to the queue, will have to read queueBack from memory instead of from cache.

We can fix this by Cache Aligning all variables we use in our Queue structure, effectively adding a padding to them so that their size is a multiple of Cache size so that two different variables are not cached together.

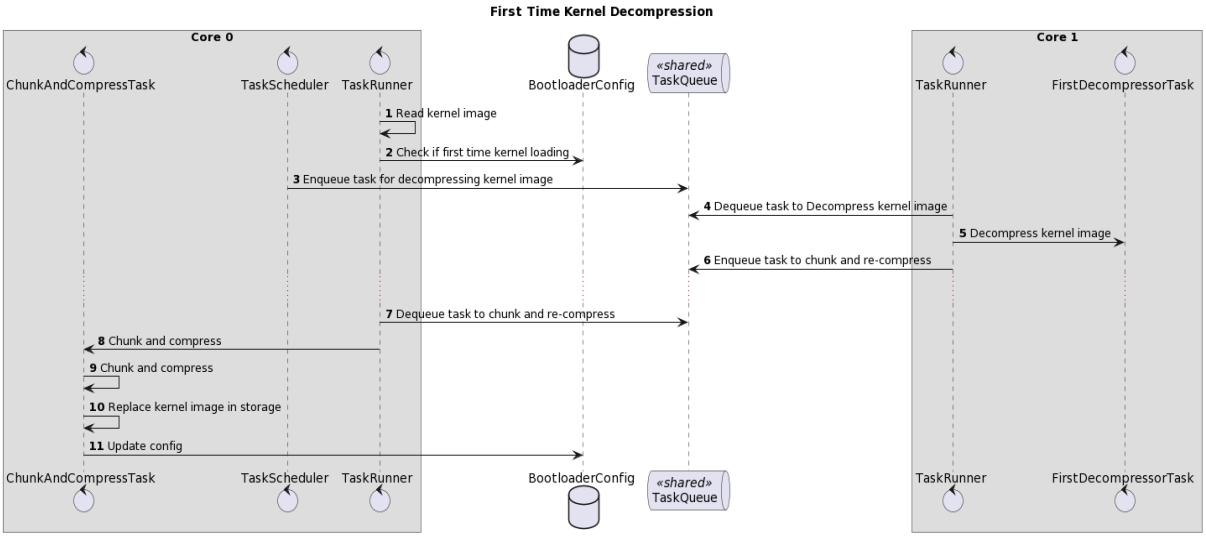


D1.10. CA_10: Split and Decompress in Parallel

We can parallelize Kernel Decompression further by splitting the kernel image into chunks and then decompressing those chucks in parallel on different cores. The decompressed chucks can then be combined to get the Kernel image.



To do this, however, we need to ensure that kernel image is compressed in chunks as well. We can do this the first time a kernel image is read. The first time a new image is read, we will compress it in chunks and replace it in the storage location.

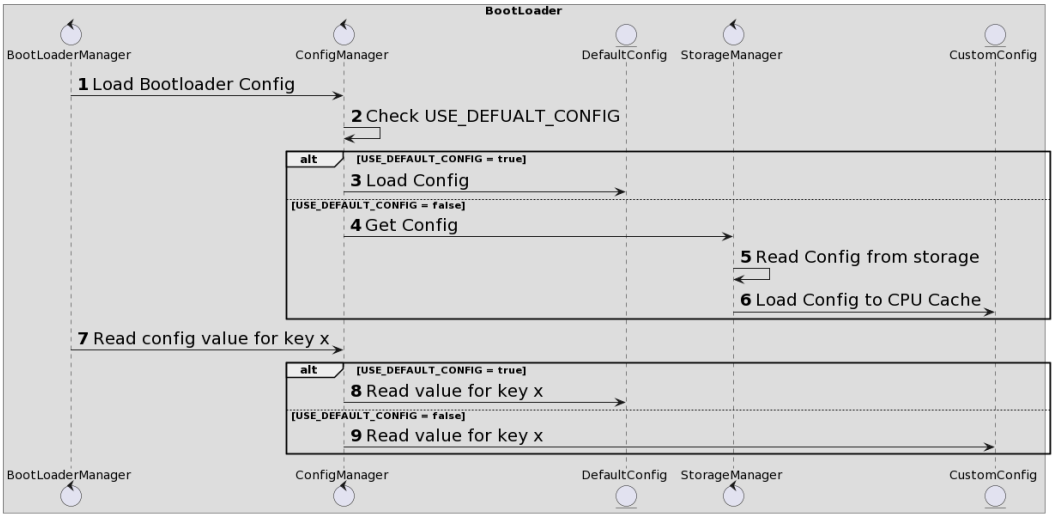


D1.11. CA_11: Set Default Bootloader Configuration in Binary at Compile Time

We can set the default bootloader configuration in the binary at compile time through environment variables. Bootloader will not have to load configuration and will save time taken to load

configuration.

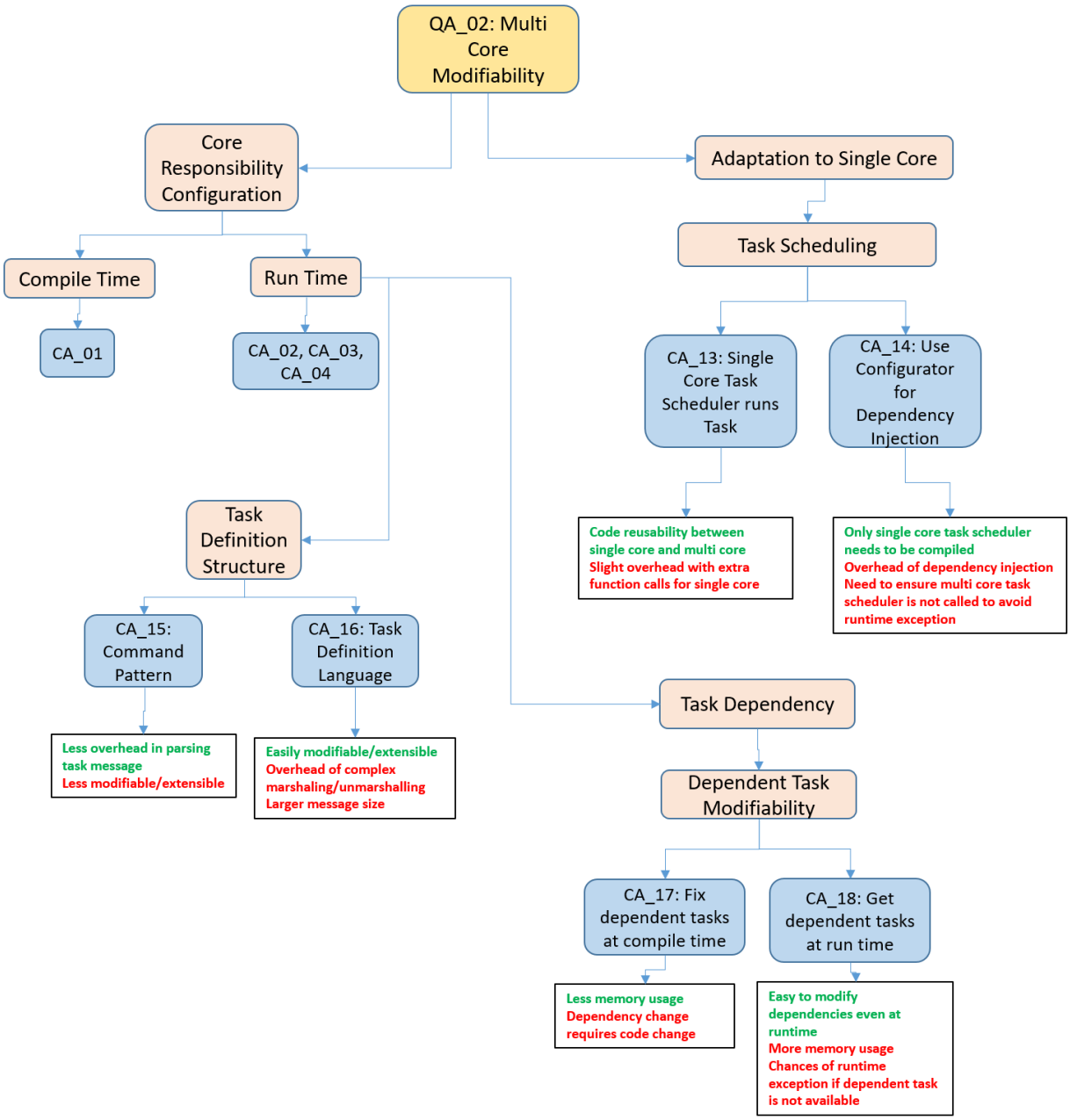
When the user changes configuration, it can be saved in a configuration file which will be stored in Storage. A flag USE_DEFAULT_CONFIG can be maintained to tell the bootloader where it needs to load configuration from. When user changes configuration, it will be set to FALSE.



D1.12. CA_12: Validate Compressed Kernel Image

Instead of first decompressing and then validating Kernel image, we can validate compressed kernel image. This way, we can parallelize decompression and validation.

D2. QA_02: Multi Core Modifiability



Concern: Core Responsibility Configuration

Handled by the following CAs:

CA_01: Compile Time Task Assignment

CA_02: Master Slave Architecture for Cores

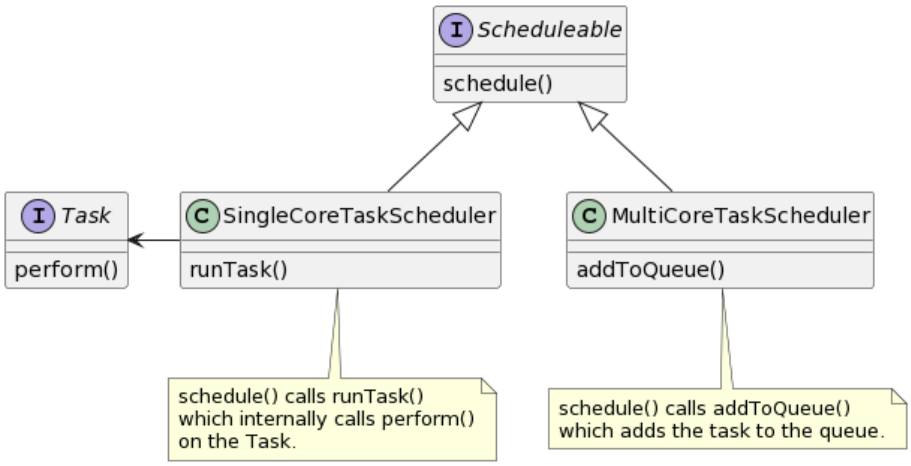
CA_03: Common Task Queue and Redundant Workers

CA_04: Scheduling on Server

Concern: Single Core Adaptation

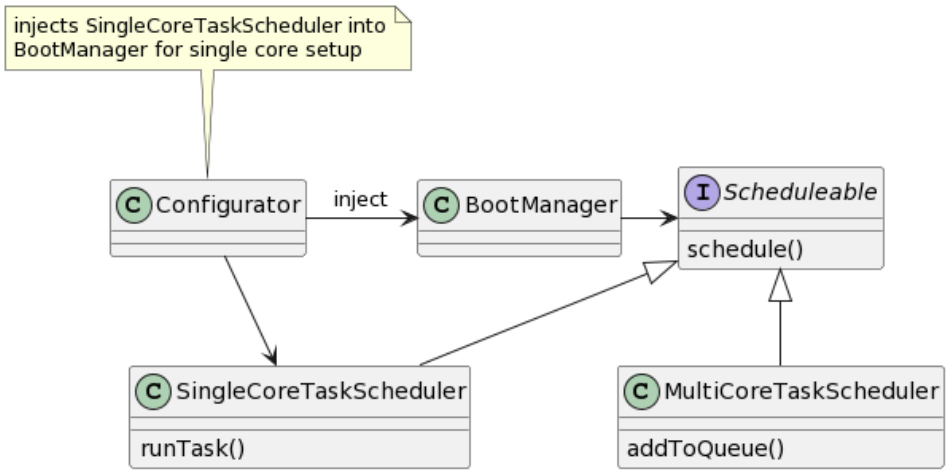
D2.1. CA_13: Single Core Task Scheduler runs Task

We can maintain Task Scheduler as an interface and have a Multi Core Task Scheduler and a Single Core Task Scheduler Implementation. The Multi Core Task Scheduler will add tasks to the queue, whereas the Single Core Task Scheduler will just run them.



D2.2. CA_14: Use Configurator for Dependency Injection

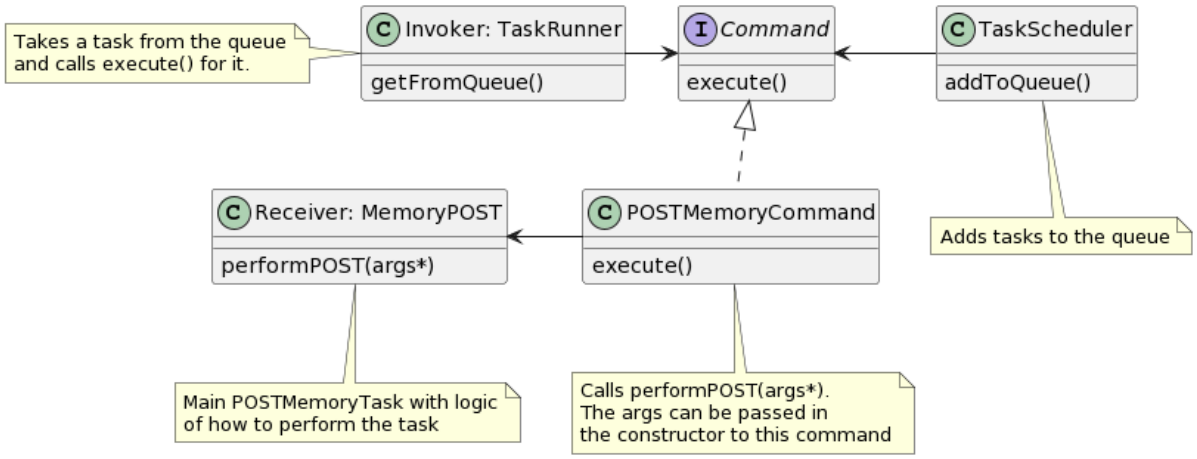
We can use a Configurator for injecting Single Core Task Scheduler or Multi Core Task Scheduler at run time depending on Core configuration. By doing this, we don't need to package multi core task scheduler for single core setup binaries and vice versa.



Concern: Task Definition

D2.3. CA_15: Command Pattern for Tasks

Tasks can be defined using Command Pattern as demonstrated below.

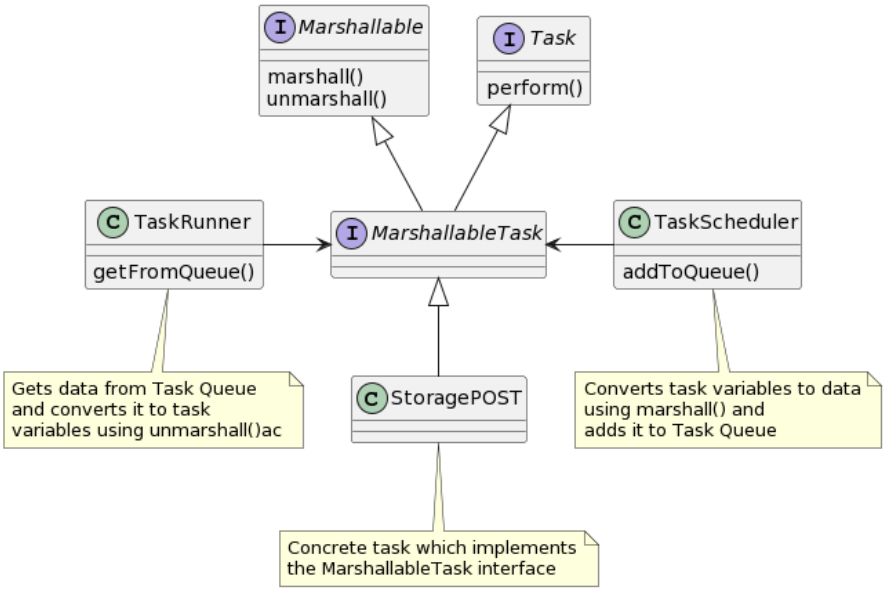


The TaskRunner running on all cores will act as the Invoker of the command, the Receiver will be the actual task that the command will execute, while the TaskScheduler will take a list of Commands and add them to the queue.

D2.4. CA_16: Task Definition Language

Conflicts with CA_15

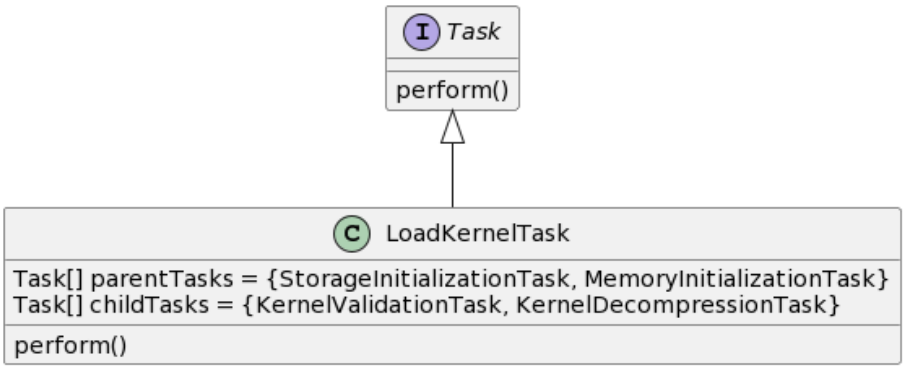
We can have our tasks extend a custom abstract Task Definition and then Marshall/Un-Marshall our concrete tasks to that format for adding to and removing from the queue respectively. This will provide a way to have a smaller data object representing the Task to store in the Task Queue.



Concern: Dependent Task Modifiability

D2.5. CA_17: Fix Dependent Tasks at Compile Time

We can fix dependent tasks at compile time by storing them as a property of each Task. This way, at run time we do not need to find the parent and child tasks of a task.

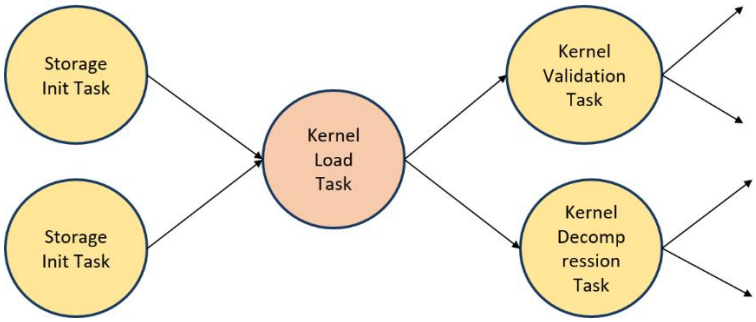


D2.6. CA_18: Get Dependent Tasks at Run Time

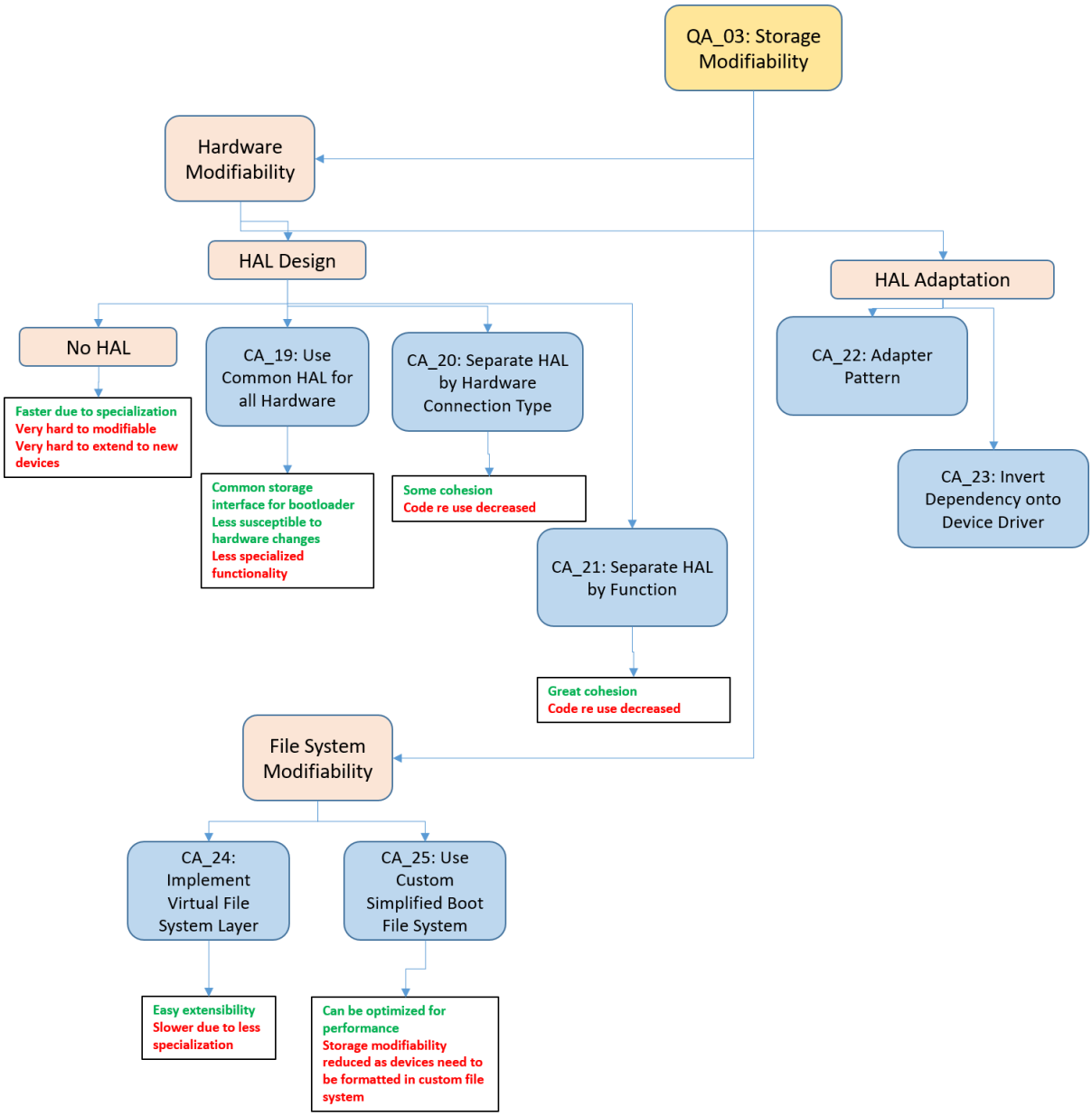
Conflicts with CA_17

We can also get dependent tasks (both child tasks and parent tasks) of a task at runtime.

They can be maintained as a configuration file and can be loaded at the time of task queue setup into a Directed Acyclic Graph.



D3. QA_03: Storage Modifiability

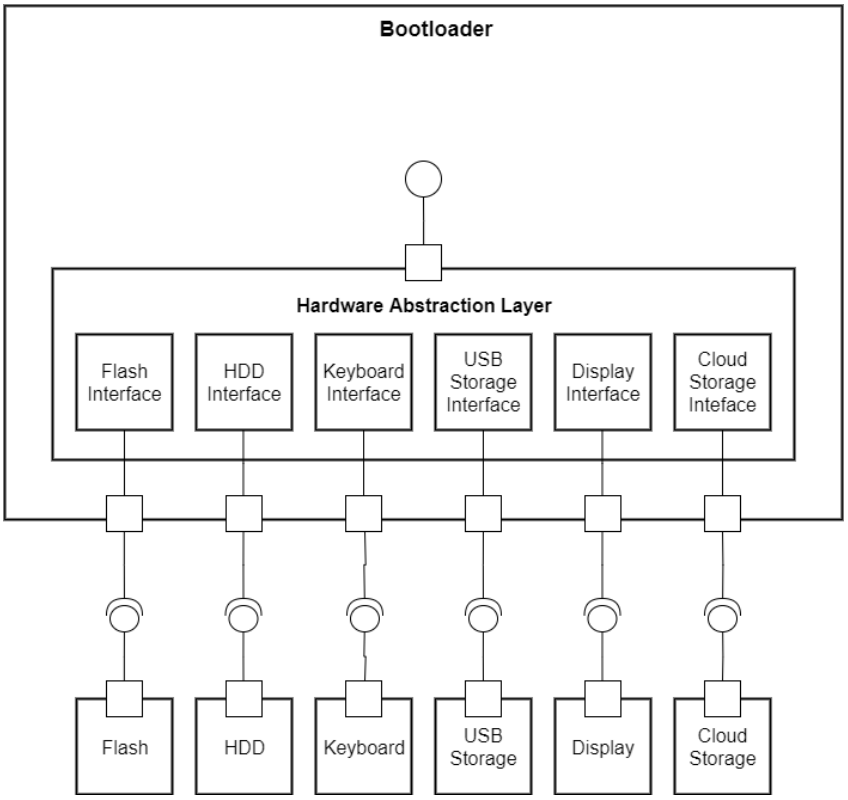


Concern: Hardware Abstraction Layer

The hardware boundary components can be grouped into a Hardware Abstraction Layer (HAL), this way the interface can be abstracted so that hardware changes do not affect other components and maintainability of hardware boundary components becomes easier.

D3.1. CA_19: Use Common HAL for all Hardware

One possible solution is to have a common HAL for all hardware that need to be connected to our system.



D3.2. CA_20: Separate HAL by Hardware Connection Type

Conflicts with CA_19

We can also split the HAL into three: On Device, Peripheral and Cloud.

So the HAL show in the above diagram would be split into:

1. On Device HAL for Flash, HDD, SSD, Built-In Display, Memory etc.
2. Peripheral HAL for hardware connected through USB, SD Card, HDMI.
3. Network HAL for Cloud Storage, network I/O devices, network displays etc.

D3.3. CA_21: Separate HAL by Function

Conflicts with CA_19 and CA_20

Another option is to separate HAL by function. For our bootloader, we can broadly classify this as:

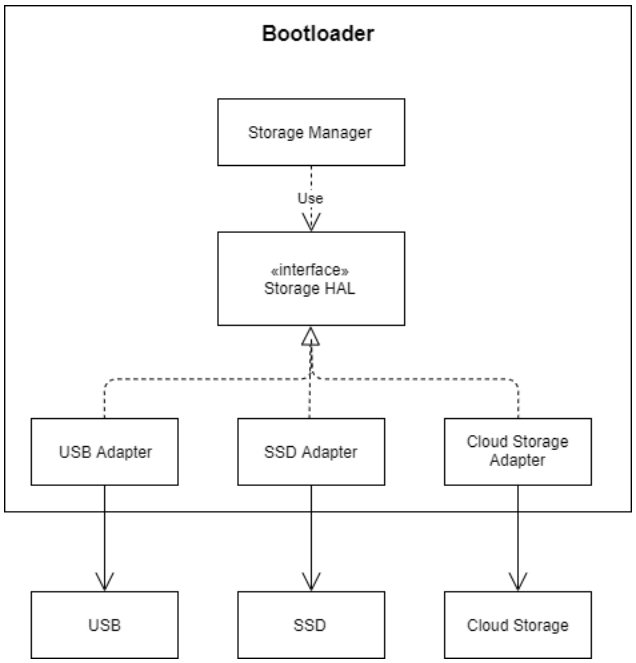
1. Storage HAL: For all storage hardware
2. I/O HAL: For I/O hardware
3. Memory Interface: For memory

Concern: HAL Adaptation

D3.4. CA_22: Implement Storage Adapter in Bootloader

Currently our Hardware Abstraction Layer depends on Storage Devices. Changes in storage devices can impact the hardware abstraction layer. To ensure changes in hardware device driver interface does not affect our Hardware abstraction layer we can use Adapter Pattern to adapt each hardware interface to our HAL.

This way changes in a hardware device driver will not affect the HAL and upper layers and will be limited to the Adapter for the hardware.

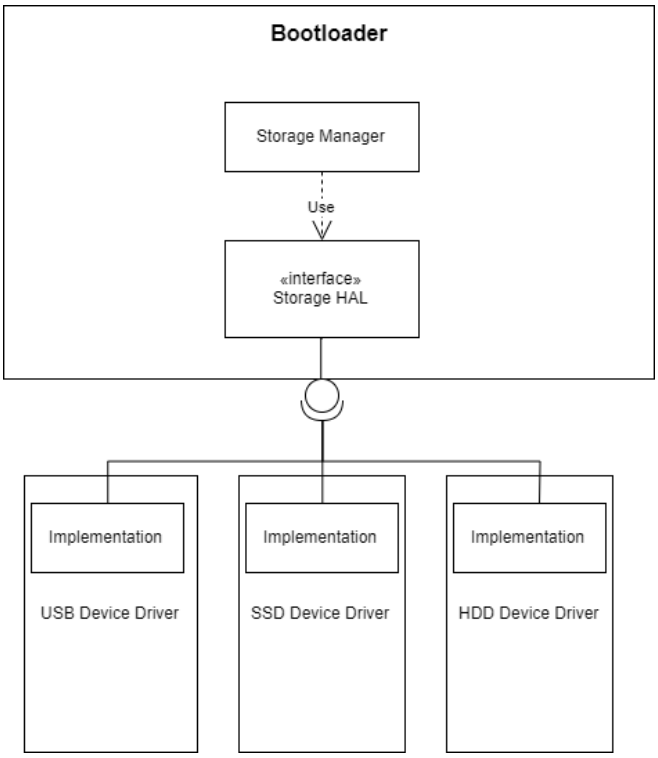


D3.5. CA_23 Storage Adapter Implementation in Device Driver

Conflicts with CA_22

We can also create an interface for storage abstraction and push the responsibility of adapting to that storage interface to Storage manufacturers in their device driver.

This way we invert the dependency, instead of Storage HAL depending on external storage device interface and having to adapt to a new storage device, the storage device depends on Bootloader interface.



Concern: File System Modifiability

D3.6. CA_24 Implement Virtual File System

We can use Adapter Pattern similar to **CA_22** and implement a Virtual File System which can be used to manage all supported File Systems.

This virtual file system will basically adapt each file system we need to support to our virtual file system so that our file system can be a stable module regardless of file system used by the storage device.

D3.7. CA_25: Use Custom Simplified Boot File System

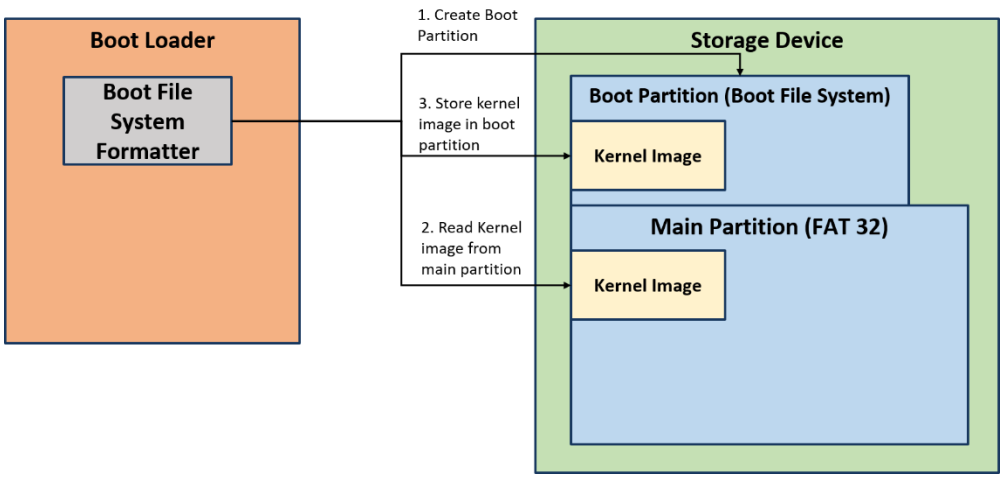
Conflicts with CA_24

We can also define a custom simplified file system specifically for our bootloader.

The file system can avoid having directories and files can be stored in contiguous blocks to improve retrieval time as we will only be storing a few files related to the kernel and bootloader configuration and do not need complex file systems used by Operating Systems.

A utility can be provided within the bootloader shell as well as be available online which would format any storage device by creating a separate partition for the Bootloader File System we have defined and storing kernel image in that partition.

Shown below is how the utility would work.



D4. QA_04: I/O Modifiability

Concern: HAL Separation

CA_19, CA_20 and CA_21 cover HAL Separation for storage and I/O Devices.

Concern: Hardware Interface Adaptation

D4.1. CA_26: Implement I/O Adapter in Bootloader

Similar to as explained in **CA_22**, we can apply Adapter Pattern for I/O device hardware abstraction.

D4.2. CA_27: I/O Adapter Implementation in Device Driver

Conflicts with CA_26

Similar to as explained in **CA_22**, we can use Dependency Inversion and have I/O device developers implement our interface in custom device drivers for our bootloader.

D5. QA_05: Shell Modifiability

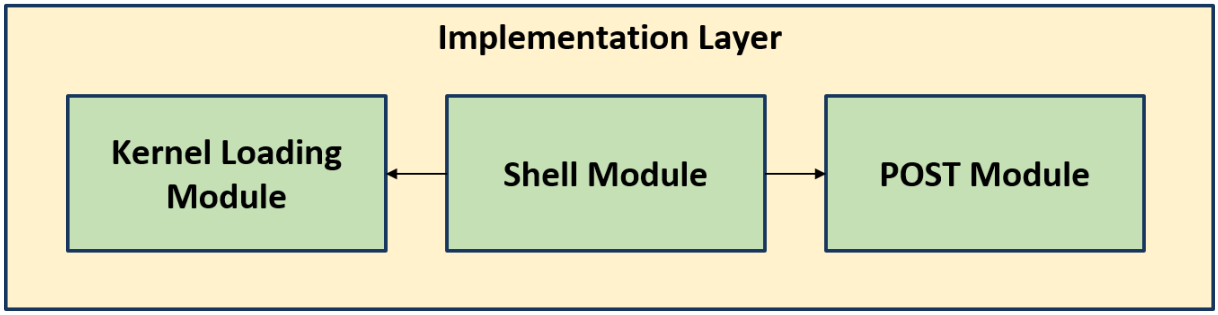
D5.1. CA_28: Segregate Modules by Concern

We can also segregate out bootloader into three major modules: Kernel Loading Module, Shell Module and POST Module.

Shell Module can have dependencies on Kernel Loading Module and POST Module since shell can be asked to change Kernel location or to perform POST on a certain device.

File System might be needed extensively by both Shell Module and Kernel Loading Module, so it can be maintained separately to avoid coupling.

Similarly Task Scheduling and Bootloader Server implementation can be maintained in a separate module.



D5.2. CA_29: Maintain each Shell Command as a separate Task

We can use the task definition from either CA_16 (Command Pattern) or CA_17 (Task Definition

Language) and apply it for Shell Commands. Each shell command can be a Task which will be executed by shell. When a new Task needs to be created

D6. General Modifiability: Layered Architecture

CA_28 Segregate Modules by Concern can be extended to create separate modules for

D6.1. CA_30: Maintain Main Bootloader, First Stage Bootloader and Bootloader Server as Applications

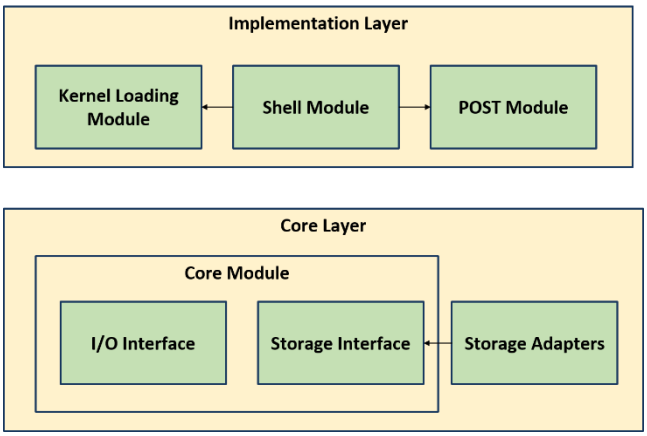
We can maintain the main Bootloader Manager of our Bootloader as an application which gets all the concrete implementations of Tasks from the implementation layer and executes them using the Task Scheduler and Task Runner.

First stage Bootloader and Bootloader Server Application code can also be maintained as applications as they are not related to the core implementation described in the implementation layer and core layer is not dependent on them.

D6.2. CA_31 Core Layer at the Bottom for Hardware Interfaces

Hardware can change very often, and if hardware changes affect our implementation then it would become very hard because it can be used extensively throughout our implementation and this does not follow Stable Dependency Principle.

To fix this we can maintain the Hardware Interfaces as the most stable module of our Bootloader in the bottom Core Layer. All implementations can use these stable interfaces for accessing Hardware and actual implementation for each specific Hardware can be maintained in Adapters as described above.



D6.3. CA_32: Task Interfaces and File System Interface in Core Layer

Task interfaces and file system are bare bones of our implementation which will be used throughout the implementation. Since many components depend on them, they need to be stable to follow Stable Dependency Principle. We can maintain them in the Core Layer to force them to be Stable.

D6.4. CA_33: Configurator injects Adapters and Tasks into Applications

Similar to as used in **CA_14**, Configurator can also be used to inject relevant Adapters and Tasks into our Bootloader.

We can use a configuration file, which will be used by the Configurator to inject relevant Task or Adapter.

The Configurator can also be used at compile time to decide which Adapters and Tasks need to be compiled in the binary.

D7. QA_06: Bootloader Storage and Loading Efficiency

D7.1. CA_34: Separate Binary for Shell Module

We can have a separate binary for Shell Module and only load it when shell needs to be launched.

This way, when we are just booting and shell is not launched, loading would be more efficient

because shell code would not need to be loaded.

D7.2. CA_35: First Stage Bootloader to load Main Bootloader

First stage bootloader is limited in size (should not exceed 512 KB) and cannot be easily updated. Therefore, to allow for extensibility and scalability we can have a first stage bootloader which only loads our main Boot loader.

D8. NFR_02: Bootloader Configuration Time

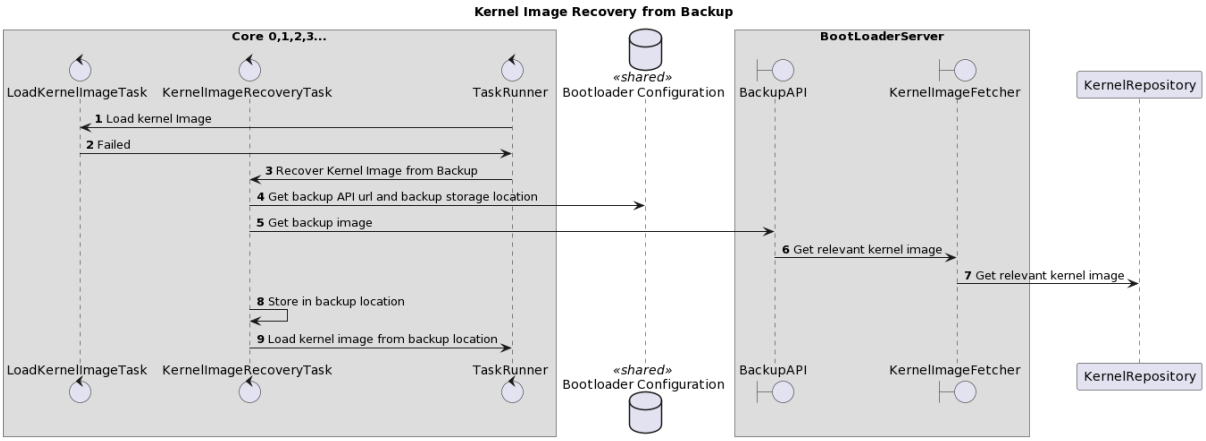
D8.1. CA_36: Persist all Boot Configuration changes at the end of Shell Session

Configuration changes made by the user can stored in CPU Cache and persisted to storage only once at the of the shell session.

D9. NFR_03: Availability of Alternate Kernel Loading Options

D9.1. CA_37: Recover from Cloud

One way to recover the OS image is by configuring backup image from being downloaded from the cloud.



D9.2. CA_38: Recover from Recovery Disk

We can also recover OS image from a recovery disk. When storage failure or OS image corruption is detected by the bootloader it will prompt the user to:

1. Create a recovery disk using online utility described in **CA_25**.
2. Connect the recovery disk to the device
3. Change the default booting device to the recovery disk from the shell.

After this the boot can proceed as normal from the recovery disk.

E. Candidate Architecture Evaluation

// A7. Architecture Design

// C7-1. Is comparison analysis of colliding candidates appropriate? (evidence)

// C7-2. Is there sufficient complement of the selected candidate?

E1. QA_01 Candidate Evaluation

Quality Requirement	Effect of Candidate Architecture			
	Conflict			
	CA_01: Compile Time Task Assignment	**CA_02: Master Slave Architecture for Cores**	**CA_03: Common Task Queue and Redundant Workers**	**CA_04: Scheduling on Server**
NFR_01, QA_01: Boot OS time should be minimized	(++) Performance can be optimized for each hardware configuration.	(+) Master can orchestrate tasks such that all independent tasks can run in parallel. However, one core is used up by master.	(++) All cores are available for parallelization and tasks can be broken down. This might be faster than master slave, but overhead of checking if dependent tasks are done is also distributed	(+) All cores are available for parallelization. However, performance can suffer due to excessive server communication.
QA_02: Multi Core Modifiability	(-) New configuration definition needed for each core count. Also, any changes in functionality will need all configurations to be re-written	(+) Only for two core setup, changes will be needed as master slave would effectively mean only one core is performing the tasks.	(++) No changes needed for different multi core setups	(++) No changes needed for different multi core setups

NFR_02: Bootloader configuration Time	(+) No effect as shell code will run on single core	(-) Master can prioritize bootloader configuration change tasks but has to wait for at least one slave to become free for it to be assigned shell tasks	(-) Shell command task might be stuck behind other tasks in the queue.	(-) Shell command task might be stuck behind other tasks.
Decision			**SELECTED because it enhances Booting Time and Multi Core Modifiability the most and those are most important**	

Risk: Shell command task might be stuck behind other tasks in the queue

Solution: Shell command priority needs to be managed using one of CA_05 or CA_06.

Quality Requirement	**Effect of Candidate Architecture**	
	Conflict	
	CA_05: Use Priority Queue for Tasks	**CA_06: Interrupt and run higher priority task**
NFR_01, QA_01: Boot OS time should be minimized	(-) Queue enqueue and dequeue will occur in logarithmic time	(--) Running tasks might get pushed to the end of the queue
QA_02: Multi Core Modifiability	(-) Will need some special handling for single core	(+) Single core can also be interrupted and higher priority task can run
NFR_02: Bootloader configuration Time	(+) Shell command can be prioritized and picked up as soon as a core becomes available	(++) Shell commands can run immediately and do not have to wait for core to finish running task

Decision		**SELECTED because only shell tasks will have higher priority and booting tasks are not important while shell is active**

Risk: Running tasks might get pushed to the end of the queue

Solution: Only Shell launch and then shell command tasks will have higher priority. Booting related tasks will be enqueued with in priority order. When shell is launched, all booting related tasks can be removed from the queue.

Quality Requirement	**Effect of Candidate Architecture**	
	Conflict	
	CA_07: Add all tasks to queue at once	**CA_08: Parent tasks add dependent tasks**
NFR_01, QA_01: Boot OS time should be minimized	(-) Child tasks can keep waiting for parent tasks to be completed once dequeued. Core is occupied but idle.	(+) Child tasks are only enqueued once parent task is complete, so cores never wait. (-)Overhead of checking and enqueueing child tasks.
QA_02: Multi Core Modifiability	(+) Easier to modify dependencies as only child tasks need to know parent tasks	(-) Tougher to modify as both parent and child tasks of a task need to be maintained.
Decision		**SELECTED because booting time is more important**

Risk Tougher to modify as both parent and child tasks of a task need to be maintained

Risk: Overhead of checking and enqueueing child tasks

Solution: We can maintain the list of dependent tasks as a topologically sorted directed acyclic graph. This will make it easier to identify parent and child tasks and to modify them if needed.

Quality Requirement	Effect of Candidate Architecture
	CA_09: Cache Align all Queue Variables
NFR_01, QA_01: Boot OS time should be minimized	(+)Fewer cache evictions so performance is improved
QA_06: Bootloader Storage and Loading Efficiency	(-) Padding applied to variables so memory usage increases
Decision	**SELECTED because booting time is more important**

Quality Requirement	Effect of Candidate Architecture
	CA_10: Split and Decompress in Parallel
NFR_01, QA_01: Boot OS time should be minimized	(+) Decompressing in parallel will reduce loading time **Sensitivity Point:** Kernel image storage size may increase because of poorer compression, leading to increase in kernel loading time.
QA_02: Multi Core Modifiability	(-) Separate decompressor needed for single core and multi core decompression
Decision	**SELECTED because booting time is more important**

Risk: Kernel image storage size may increase because of poorer compression, leading to increase in kernel loading time.

Solution: We can configure a benchmark for when the kernel is re-compressed and is being replaced. If the difference in kernel image size for single compression and chunked compression is significant, we can configure the bootloader to not use chucked decompression and not replace the kernel image.

Quality Requirement	Effect of Candidate Architecture

	CA_11: Set Default Bootloader Configuration in Binary at Compile Time
NFR_01, QA_01: Boot OS time should be minimized	(+)Booting time is reduced because configuration file does not need to be loaded
QA_02: Multi Core Modifiability	(--) Some properties might need to be different for single and multi-core applications and this will be hard to manage if config is in code
QA_06: Bootloader Storage and Loading Efficiency	(-) Bootloader size will increase due to configuration data being stored in the binary
Decision	**REJECTED due to very poor modifiability.**

Quality Requirement	**Effect of Candidate Architecture** **CA_12: Validate Compressed Kernel Image**
NFR_01, QA_01: Boot OS time should be minimized	(+) Compression and validation can be done in parallel.
Compression Algorithm change	(-) Compressed image will need to re-validated
Decision	**SELECTED because booting time is more important**

Risk: Compressed image will need to re-validated

Solution: Re validation can be performed and checksum can be updated before pushing bootloader update with new algorithm to devices.

E2. QA_02 Candidate Evaluation

Quality Requirement	Effect of Candidate Architecture CA_13: Single Core Task Scheduler runs Task
NFR_01, QA_01: Boot OS time should be minimized	(-) Very slight overhead due to inheritance and extra function calls
QA_02: Multi Core Modifiability	(++) Separates out implementation of tasks from single core and multi core code such that it can be reused by both
Decision	**SELECTED because of improved multi core modifiability and not a significant negative impact on booting time**

Quality Requirement	Effect of Candidate Architecture CA_14: Use Configurator for Dependency Injection
NFR_01, QA_01: Boot OS time should be minimized	(-) Dependency injection code will have overhead in booting time
QA_02: Multi Core Modifiability	(+) Improves modifiability by providing easy way to manage switching between single core and multi core systems.
QA_06: Bootloader Storage and Loading Efficiency	(+) Bootloader size will be reduced for single core setups as multi core DLL will not need to be loaded
Decision	**SELECTED because of improved multi core modifiability and bootloader size and not a significant negative impact on booting time**

Risk: Dependency injection code will have overhead in booting time

Solution: Developers need to ensure that dependency injection is done in a way that minimizes overhead.

Quality Requirement	Effect of Candidate Architecture	
	Conflict	
	CA_15: Command Pattern for Tasks	**CA_16: Task Definition Language**
NFR_01, QA_01: Boot OS time should be minimized	(+) Very little overhead since commands objects are passed and no conversion is needed	(-) Overhead of marshalling and unmarshalling tasks
QA_02: Multi Core Modifiability	(-) Limited information can be shared in tasks and it's tougher to modify information shared	(+) Easy to modify and customize task definition to add/change information shared
Decision		**SELECTED despite impact on booting time as information shared between tasks needs to be extensive and should be easily changeable**

Risk: Overhead of marshalling and unmarshalling tasks

Solution: Marshalling and Unmarshalling needs to be implemented efficiently and without using too many expensive operations such as reflection.

Quality Requirement	Effect of Candidate Architecture	
	Conflict	
	CA_17: Fix Dependent Tasks at Compile Time	**CA_18: Get Dependent Tasks at Run Time**
NFR_01, QA_01: Boot OS time should be minimized	(+) Dependent tasks are already mentioned in task.	(-) Dependent tasks need to be found for each task.

QA_02: Multi Core Modifiability	(--) Code changes needed to change dependent tasks	(+) Easier modification since code changes are not needed.
Decision		**SELECTED despite impact on booting time as dependent tasks can change and this change should not require code change.**

Risk Dependent tasks need to be found for each task.

Solution: We can maintain the list of dependent tasks as a topologically sorted graph. This will make it easier to identify parent and child tasks.

E3. QA_03 Candidate Evaluation

Quality Requirement	**Effect of Candidate Architecture**		
	Conflict		
	CA_19: Use Common HAL for all Hardware	**CA_20: Separate HAL by Hardware Connection Type**	**CA_21: Separate HAL by Function**
QA_03: Storage modifiability	(+) Common modules can be shared	(+) Common modules relating to establishing and maintaining connections can be shared	(-) Common modules relating to connections will be repeated.
	(--) Very low cohesion as hardware with different functionalities and different connection types can be grouped into the same HAL	(+) Some cohesion due to common connection mode	(+) High cohesion as HAL is grouped by function and follows single responsibility principle

QA_04: I/O Modifiability	(-) I/O devices will have more specialized functionality are changes to HAL are more likely	(-) I/O devices will have more specialized functionality are changes to HAL are more likely	(+) I/O device changes will be limited to I/O HAL
QA_05: Shell Modularity	(-) Shell will be coupled to booting modules through common HAL	(-) Shell will be coupled to booting modules through common HALs	(+)Shell will be less tightly coupled to booting module as only storage HAL changes will affect shell but I/O HAL changes will not affect booting
Decision			**SELECTED as it significantly enhances modifiability and modularity**

Risk: Common modules relating to connections will be repeated.

Solution: Common modules for connection can be separated out from the module implementing HAL and maintain separately in one place.

Quality Requirement	**Effect of Candidate Architecture**	
	Conflict	
	CA_22: Implement Storage Adapter in Bootloader	**CA_23: Storage Adapter Implementation in Device Driver**

QA_03: Storage modifiability	(+) More predictable development time to add support for new storage devices as responsibility is with our developers (-) Changes to storage hardware interface will need to be adapted	(-) Tougher to predict how long it will take storage manufacturers developers to add support for our interface (+) No changes needed as interface will be updated by storage manufacturer during updates
Decision	**SELECTED because supporting all storage devices is critical to booting and changes to hardware interface can be rare**	

Risk: Changes to storage hardware interface will need to be adapted

Solution: When changes to the storage hardware interface by the manufacturer occur it will usually be communicated beforehand and can be planned and adapted.

Quality Requirement	**Effect of Candidate Architecture**	
	Conflict	
	CA_24: Implement Virtual File System	**CA_25: Use Custom Simplified Boot File System**
NFR_01, QA_01: Boot OS time should be minimized	(-) Slower because proper file system would need to be setup	(+) Faster access because we will setup a minimal file system

QA_03: Storage modifiability	(+) Any new storage device can be supported without any operation as long it is already formatted with a supported file system	(-) Each new storage device would need to be partitioned and boot partition would need to be formatted with custom file system (+) Once setup, we do not have to worry about adding support for any new file systems
QA_06: Bootloader Storage Efficiency	(-) Multiple file system support will bloat bootloader size	(+) Custom file system will be lightweight (-) Conversion tool will bloat bootloader size
Decision		**SELECTED because it enhances boot time and does not need additional file system support**

Risk: Each new storage device would need to be partitioned and boot partition would need to be formatted with custom file system

Solution: When user changes storage device in shell, we can provide a prompt that a new partition will be created and then partition the device and copy kernel image to the new partition as part of the device change without any further user input needed.

Risk: Conversion tool will bloat bootloader size

Solution: We will have to optimize conversion tool so that it does not bloat the bootloader size.

E4. QA_04 Candidates Evaluation

Quality Requirement	Effect of Candidate Architecture
	Conflict

	CA_26: Implement I/O Adapter in Bootloader	CA_27: I/O Adapter Implementation in Device Driver
QA_03: Storage modifiability	(+) More predictable development time to add support for new I/O devices as responsibility is with our developers (-) Changes to I/O hardware interface will need to be adapted	(-) Tougher to predict how long it will take storage manufacturers developers to add support for our interface (+) No changes needed as interface will be updated by I/O device manufacturer during updates
Decision		**SELECTED because I/O devices can change more regularly and adding support for each one is not critical to bootloader operation**

Risk: Tougher to predict how long it will take storage manufacturers developers to add sup port for our interface

Solution: We can keep our interface as close to common popular standard I/O hardware interfaces as possible

E5. QA_05 Candidate Evaluation

Quality Requirement	Effect of Candidate Architecture
	CA_28: Segregate Modules by Concern
QA_05: Shell Modifiability	(+) Shell modules can modified separately from kernel module and POST module
	(-) Shell module will have dependencies on Kernel and POST modules as it might need to perform tasks related to those modules

Decision	**SELECTED because shell module is likely to be the most unstable of the three modules so it does not break Stable Dependency Principle**

Risk: Shell module will have dependencies on Kernel and POST modules

Solution: As much as possible, we will have shell module depend on abstractions rather than concrete implementation in Kernel Loading and POST modules.

Quality Requirement	**Effect of Candidate Architecture**
	CA_29: Maintain each Shell Command as a Separate Task
QA_05: Shell Modifiability	(++) New shell commands can be easily added as only a new task needs to be implemented.
NFR_02: Bootloader configuration Time	(-) Some commands might be complex and would be faster if broken down and done concurrently
Decision	**SELECTED because complex command can be further broken down using dependent tasks**

Risk: Some commands might be complex and would be faster if broken down and done co ncurrently

Solution: Complex commands can be further broken down by having a parent task which either does some initial non parallelized work or just do nothing. This parent task will then have child tasks which can be performed concurrently on multiple cores.

E6. Layered Architecture Evaluation

Quality Requirement	Effect of Candidate Architecture
	CA_30: Maintain Main Bootloader, First Stage Bootloader and Bootloader Server as Applications
QA_02: Multi Core Modifiability QA_03: Storage Modifiability QA_04: I/O Modifiability QA_05: Shell Modifiability	(+) Configuration changes and other changes that do not affect core implementation will not have an effect on core implementation
	(-) Implementation cannot have any dependencies on bootloader manager
	(-) Bootloader manager cannot have any dependencies on core layer
Decision	**SELECTED because it is good to keep bootloader manager as not having many dependencies to make it easy to make no implementation related changes to its flow**

Quality Requirement	Effect of Candidate Architecture
	CA_31: Core Layer at the Bottom for Hardware Interfaces
NFR_01, QA_01: Boot OS time should be minimized	(-) Overhead of adapter
QA_03: Storage modifiability	(++) Storage and I/O device changes do not affect upper layer since upper layer depends on interface.
QA_04: I/O Modifiability	
Decision	**SELECTED because storage or I/O device related code in implementation will make maintenance and addition of new devices very hard**

Risk: Overhead of adapter

Solution: Adapter needs to be implemented such that it has minimal overhead.

Quality Requirement	Effect of Candidate Architecture **CA_32: Task Interfaces and File System Interface in Core Layer**
QA_02: Multi Core Modifiability	(+) Task interface is abstracted and can be implemented for each concrete task in the implementation layer
QA_03: Storage modifiability	(+) File system can be shared between different modules of upper layers without worrying about coupling.
	Sensitivity Point: Both File system and Task Interface need to be designed well to ensure there are not many changes to them.
Decision	**SELECTED**

Quality Requirement	Effect of Candidate Architecture **CA_33: Configurator injects Adapters and Tasks into Applications**
NFR_01, QA_01: Boot OS time should be minimized	(-) Overhead of extensive dependency injection
QA_02: Multi Core Modifiability	(+) Configurator can dynamically inject Tasks that are needed into the application allowing separation of implementation from application
QA_03: Storage modifiability	(+) Configurator can dynamically add only required Adapters to the application
QA_04: I/O Modifiability	

QA_06: Bootloader Storage and Loading Efficiency	(+) Bootloader binary size will be optimized as only required modules will be injected
Decision	**SELECTED due to extensive modifiability and bootloader size benefits**

Risk: Overhead of extensive dependency injection

Solution: Compile Time Dependency Injection can be used to reduce overhead.

E7. QA_06 Candidate Evaluation

Quality Requirement	**Effect of Candidate Architecture**
	CA_34: Separate Binary for Shell Module
NFR_01, QA_01: Boot OS time should be minimized	(+) Smaller booting binary will be faster to load
QA_06: Bootloader Storage Efficiency	(+) While overall bootloader size will not be minimized, size loaded to memory at a time will be minimized
NFR_02: Bootloader Configuration Time	(-) Initial shell loading time will increase slightly
Decision	**REJECTED due to risk to NFR**

Quality Requirement	**Effect of Candidate Architecture**
	CA_35: First Stage Bootloader only loads Main Bootloader
NFR_01, QA_01: Boot OS time should be minimized	(-) Additional binary loading time will increase booting time.

QA_06: Bootloader Storage Efficiency	(+) Since we have a constraint that first stage bootloader needs to fit within 512 KB, it is better to keep a dummy bootloader as a first stage bootloader
Updating bootloader	(+) It's complicated to update first stage bootloader so main booting code is better maintained in second stage bootloader
Decision	**SELECTED due to risk of failing 512 KB constraint with future updates and difficulty in updating first stage bootloader**

Risk: Additional binary loading time will increase booting time.

Solution: We can store second stage binary in a pre-defined storage address like CA_04 so that file system does not need to be initialized to load it.

E8. NFR_02 Candidate Evaluation

Quality Requirement	**Effect of Candidate Architecture**
	CA_36: Persist all Boot Configuration changes at the end of Shell Session
NFR_02: Shell Configuration Change Time should be minimized	(++) Since bootloader configuration does not need to be updated after every command, overall configuration change time should improve a lot **Sensitivity Point**: If shell session crashes or device is restarted before shell session is closed, all changes will be lost
Decision	**SELECTED**

Risk: If shell session crashes or device is restarted before shell session is closed, all change s will be lost

Solution: We can ignore this and load the previous configuration the next time bootloader loads as it is not that much of an issue if configuration changes are not recovered. Further, we can have a "Save and Exit" option instead of just an "Exit" option to convey to the user that their changes will only be saved at the end of the session.

E9. NFR_03 Candidate Evaluation

Quality Requirement	Effect of Candidate Architecture	
	Conflict	
	CA_37: Recover from Cloud	**CA_38: Recover from Recovery Disk**
NFR_03: Availability of Alternate Kernel Loading Options	(+) Recovery time is predictable as long as network connection is available. **Sensitivity Point**: If network is not available to the bootloader it cannot recover from the cloud	(-) User will need to create a recovery disk from another system to recover unless a recovery disk is already available. So recovery time is less predictable (+) If recovery disk is already connected to the device, recovery will be faster than over cloud
Decision	**SELECTED because of predictable recovery performance**	

Risk If network is not available to the bootloader it cannot recover from the cloud.

Solution: If network is not available, we will fall back to CA_29 and ask user to recover from Recovery Disk.

E10. Overall Evaluation Result

Candidate Architecture	Result
CA_01: Compile Time Task Assignment	
CA_02: Master Slave Architecture for Cores	
CA_03: Common Task Queue and Redundant Workers	SELECTED
CA_04: Scheduling on Server	
CA_05: Use Priority Queue for Tasks	
CA_06: Interrupt and run higher priority task	SELECTED
CA_07: Add all tasks to queue at once	

CA_08: Parent tasks add dependent tasks	SELECTED
CA_09: Cache Align all Queue Variables	SELECTED
CA_10: Split and Decompress in Parallel	SELECTED
CA_11: Set Default Bootloader Configuration in Binary at Compile Time	
CA_12: Validate Compressed Kernel Image	SELECTED
CA_13: Single Core Task Scheduler runs Task	SELECTED
CA_14: Use Configurator for Dependency Injection	SELECTED
CA_15: Command Pattern for Tasks	
CA_16: Task Definition Language	SELECTED
CA_17: Fix Dependent Tasks at Compile Time	
CA_18: Get Dependent Tasks at Run Time	SELECTED
CA_19: Use Common HAL for all Hardware	
CA_20: Separate HAL by Hardware Connection Type	
CA_21: Separate HAL by Function	SELECTED
CA_22: Implement Storage Adapter in Bootloader	SELECTED
CA_23: Storage Adapter Implementation in Device Driver	
CA_24: Implement Virtual File System	
CA_25: Use Custom Simplified Boot File System	SELECTED
CA_26: Implement I/O Adapter in Bootloader	
CA_27: I/O Adapter Implementation in Device Driver	SELECTED
CA_28: Segregate Modules by Concern	SELECTED
CA_29: Maintain each Shell Command as a Separate Task	SELECTED
CA_30: Maintain Main Bootloader, First Stage Bootloader and Bootloader Server as Applications	SELECTED
CA_31: Core Layer at the Bottom for Hardware Interfaces	SELECTED
CA_32: Task Interfaces and File System Interface in Core Layer	SELECTED
CA_33: Configurator injects Adapters and Tasks into Applications	SELECTED
CA_34: Separate Binary for Shell Module	
CA_35: First Stage Bootloader only loads Main Bootloader	SELECTED
CA_36: Persist all Boot Configuration changes at the end of Shell Session	SELECTED
CA_37: Recover from Cloud	SELECTED
CA_38: Recover from Recovery Disk	

F. Final Architecture

// A7. Architecture Design

// C7-3. Is there right integration into the final architecture?

// C7-4. Is there appropriate risk management of the final architecture?

F1. Deployment View

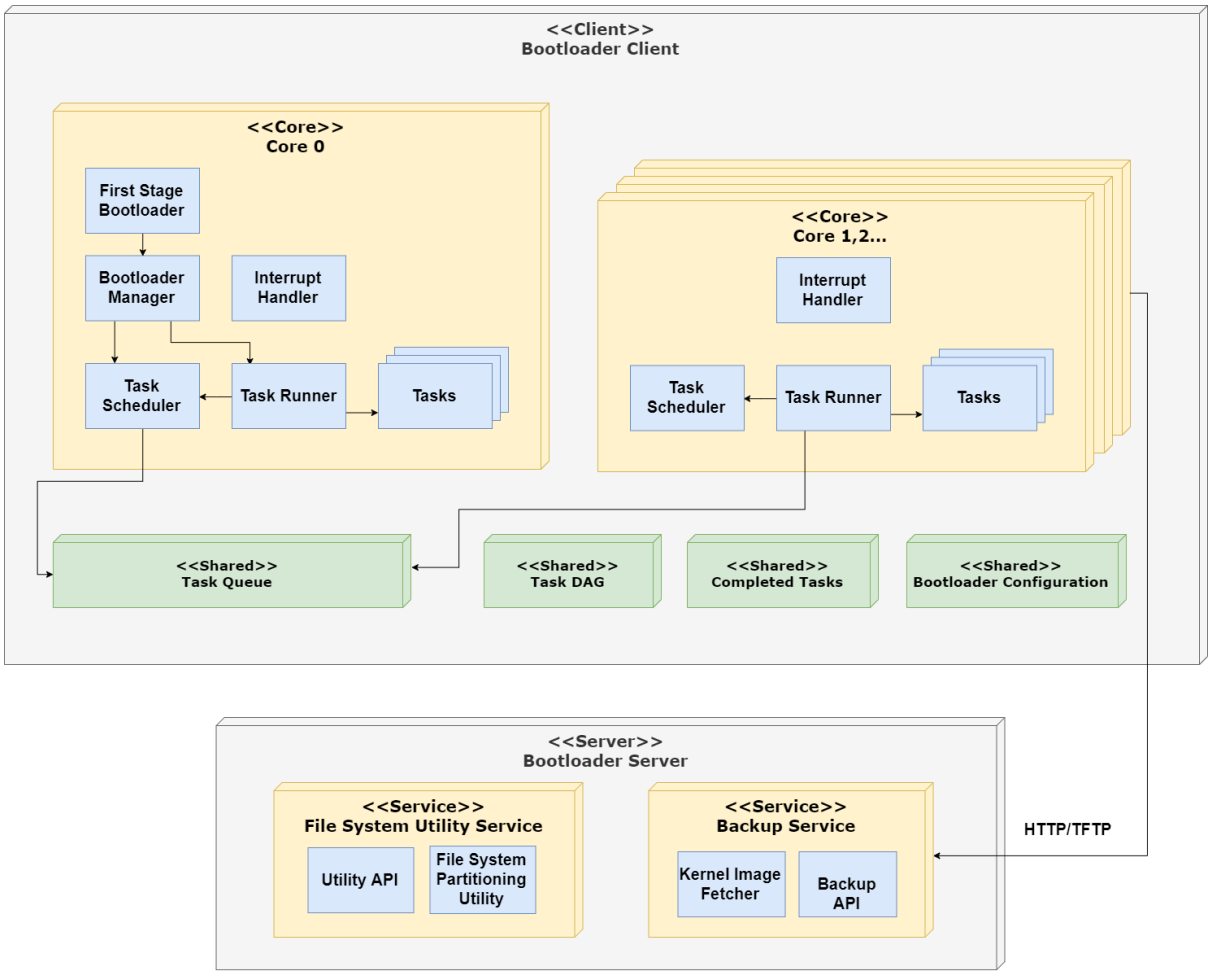


Figure 13: Multi Core Deployment View

The above Deployment View show the Multi Core Deployment of our Bootloader.

Initially, **First Stage Bootloader** will be executed by the Boot ROM to ensure Bootloader size does not become an issue as explained in **CA_35 -> First Stage Bootloader to load Main Bootloader**.

The basic idea in the main bootloader is to provide concurrent processing of tasks by enqueueing tasks to a shared **Task Queue** using **Task Scheduler** and de-queueing and running those **Tasks** on redundant cores which run **Task Runner** to constantly dequeue tasks from the queue and execute them. This way all Cores can be used for processing bootloader tasks. (**CA_03 -> Common Task Queue and Redundant Workers**).

Here, **Tasks** represent the concrete functionality such as Kernel Loader, Memory POST, and Validator etc. which the Task Runner will execute after taking them from the queue.

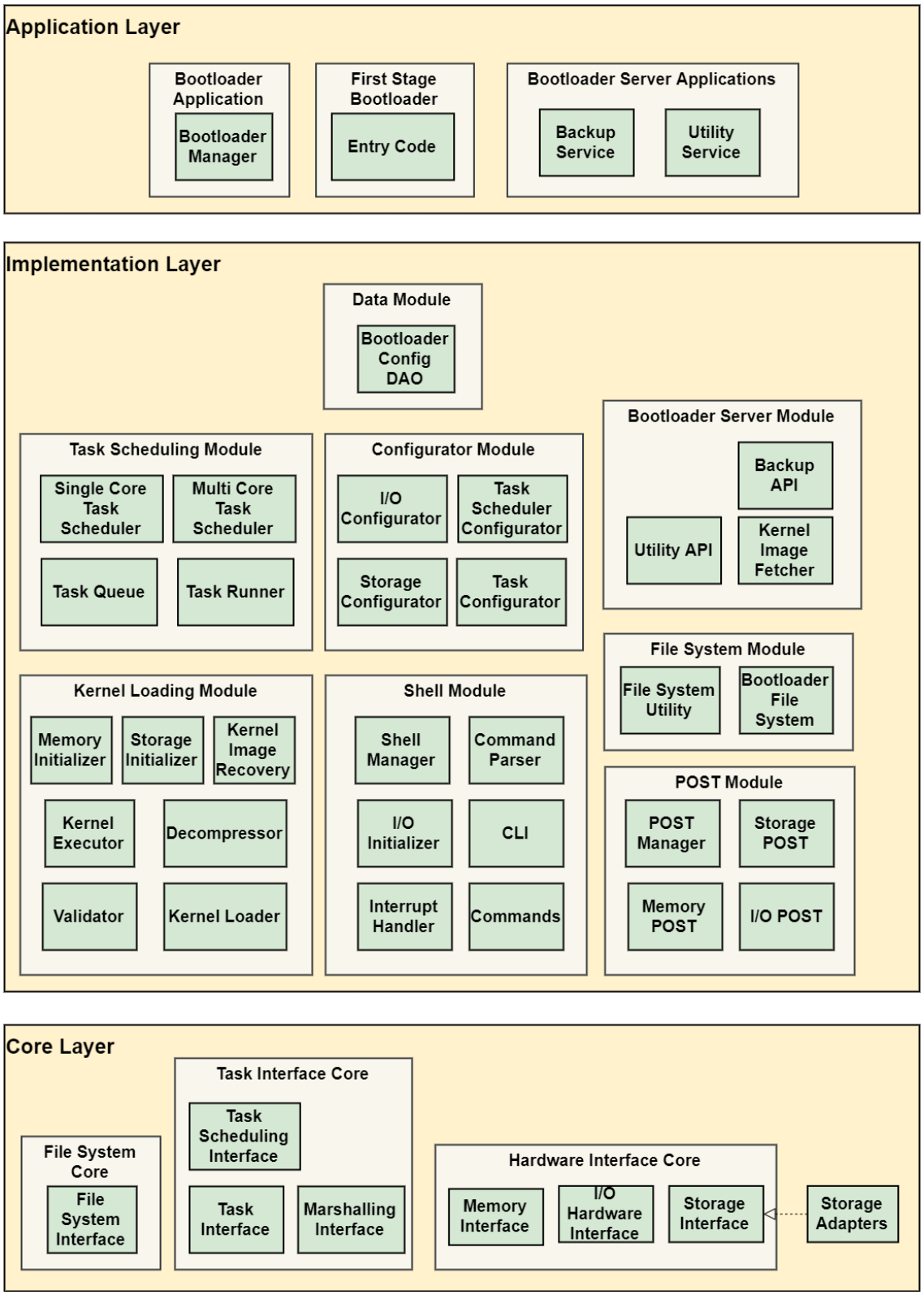
To ensure dependent tasks do not have to wait long for their parent tasks to complete each core has its own Task Scheduler, which will be used to enqueue dependent child tasks at the end of a particular task (**CA_08 -> Parent tasks add dependent tasks**). **Task DAG** in shared memory is a topologically sorted Directed Acyclic Graph which stores the task dependencies and **Completed Tasks** is a list of completed tasks which will be used to find which child tasks need to be enqueued by checking if all their parent tasks have completed.

Bootloader Manager performs basic hardware initialization such as Memory and other basic hardware and loads the **Bootloader Configuration** into memory. In multi core setup, it also asks Task Scheduler to initialize the Task Queue and sets up the other cores to run Task Runner and listen to the queue.

Interrupt Handler is present in all cores to handle case where core execution will need to be interrupted to run high priority task (**CA_06 -> Interrupt and run higher priority task**). This will only happen for Shell tasks when shell is launched, as all other tasks can have the same priority and run according to queue order.

Backup Service in the Bootloader Server will be used when the Bootloader cannot load the kernel from Kernel Storage device (**CA_37 -> Recover from Cloud**). The Recovery Task will call the Backup API, which will use the Kernel Image Fetcher to get the relevant Kernel image from an online repository (provided by the OS provider). **File System Utility Service** is used to provide a Utility to partition a Storage Device with the custom Bootloader File System described in **CA_25 -> Use Custom Simplified Boot File System.**

F2. Module View



The above Module View shows the four layer architecture of our Bootloader.

Core Layer is the bottom layer and describes the interfaces used by the bootloader. Originally, our bootloader depended on hardware implementations. However, hardware is instable and therefore we have implemented stable interfaces which can be used by our Implementation Layer to implement functionality as described in **CA_31 -> Core Layer at the Bottom for Hardware Interfaces**. Core Layer also contains Task Interface Module and File System Interface. These two are used extensively by our implementation layer and therefore need to be stable. So we will define stable interfaces for them which can be used in the implementation. File System implementation can also be maintained in this layer as an Extension (**CA_32 -> Core Layer for Task Interfaces and File System**)

Implementation Layer contains all our implementation logic for various bootloader tasks. All functionalities such as Kernel Loading, Validation, Shell Commands, and Storage POST will implement the Task interface. Our main Bootloader functionality can be divided into 3 major modules: Kernel Loading, POST and Shell Modules to improve code cohesion and to ensure changes in one module do not affect other modules. Task scheduling code and Bootloader Server code can also be segregated into their own modules (**CA_28 -> Segregate Modules by Concern**).

A Configurator is used by the Bootloader to configure it for either Single Core or Multi Core processing by injecting the relevant implementation of Task Scheduler into the Boot Manager (**CA_14 -> Use Configurator for Dependency Injection**). Similarly, Configurators also provides Task Runner with concrete implementations of Tasks to be performed by the Bootloader and Hardware Adapter for Device Drivers. This way, mapping to concrete implementations is maintained in the Configurators and can easily be modified. (**CA_33 -> Configurator injects Adapters and Tasks into Applications**).

Application Layer contains application code for our Main Bootloader, First Stage Bootloader and Bootloader Server as this is the boilerplate code needed for our application to run and not related to any implementation (**CA_30 -> Maintain Main Bootloader, First Stage Bootloader and Bootloader Server as Applications**).

F3. Risk Management

Some of the risks in the architecture are already explained with their mitigation during Candidate Architecture Evaluation in **Appendix E** and handled in the Final Architecture. The following are some further risks associated with the current architecture:

1. There can be a case where two cores try to dequeue from the queue at the same time. In this case, atomicity of dequeue operation is required.

   Mitigation: Mutexes or Locks are costly so instead we can implement atomicity in our queue to use atomic Read-Modify-Write operations to provide non-blocking synchronization.

2. Shell Entry and Shell Exit need to be managed carefully to ensure shell launches quickly and after shell is closed, bootloader continues execution.

   Mitigation: At shell entry an interrupt will be sent to execute Shell Entry task immediately. The queue can be emptied at this point and booting tasks can be stopped. Some cleanup and garbage collection of running booting related tasks can also be performed. At shell exit, the initial tasks from the Bootloader Task DAG can once again be enqueued to the Task Queue to restart the boot.

3. Current Architecture relies on I/O device manufacturers to support our I/O interface in their Device Drivers.

   Mitigation: I/O interface must be defined such that it can easily be adapted in I/O device drivers. Even though I/O devices are not needed in the core Boot use case, if there is a need to support an I/O device urgently and manufacturer cannot support it at their end, an Adapter can be implemented for that device.

4. Using Custom File System means that the storage device used for booting must allow partitioning and must have enough space for a Boot Partition.

   Mitigation: We need to ensure in communication with storage device manufacturers that all devices can support this and also ensure partitioning code is well tested so that there are no hardware specific issues which would corrupt the device.

5. Current Architecture requires the user to have some knowledge of how to operate shell as a GUI has not been provided.

   Mitigation: This can be mitigated somewhat by launching a help section with list of commands and their function when shell is launched.

G. Architecture Evaluation(ATAM)

// A10. Architecture Evaluation

// C10-1. Are there sufficient quality scenarios evaluating architecture?

// C10-2. Are there sufficient architectural decisions identified?

// C10-3. Is the analysis of design decisions appropriate? (evidence)

// C10-4. Are the mitigation plans to the risk factors appropriate?