VCU Firmware Design Document

Written by Rafael Guevara

This document goes into detail about the design decisions for the firmware architecture of the Vehicle Control Unit (VCU) and the implementation details of important software modules.

Design Principles

During the development of this major firmware version and through many iterations, a set of design principles emerged for architectural design decisions. Future redesigns should be run against this set of principles to ensure that the firmware has a consistent architecture and core design patterns. These principles specifically tackle issues and headaches with previous iterations.

1. **Event driven:** The minimum number of tasks should be running at any given moment

   Previously, all tasks were periodic and the flow of data occurred by repeatedly checking a shared global data structure. To make the system more event driven, data travels through queues instead; tasks will only be awakened if there is data in the queue.

2. **Upstream data flow, downstream control flow:** Data should as much as possible flow up a predefined thread hierarchy (in our case, a priority hierarchy)

   A given set of data flowing upwards should unblock a higher priority task that can then execute immediately. This makes the system easier to reason about because there becomes a clear, predefined path in which data can flow.

   In an event driven system, events can wake up higher priority tasks and be handled immediately. The response to these events by higher priority tasks would determine the state of lower priority tasks.

3. **State/Control variables:** Minimize the number of variables that cache data from the OS and hardware.

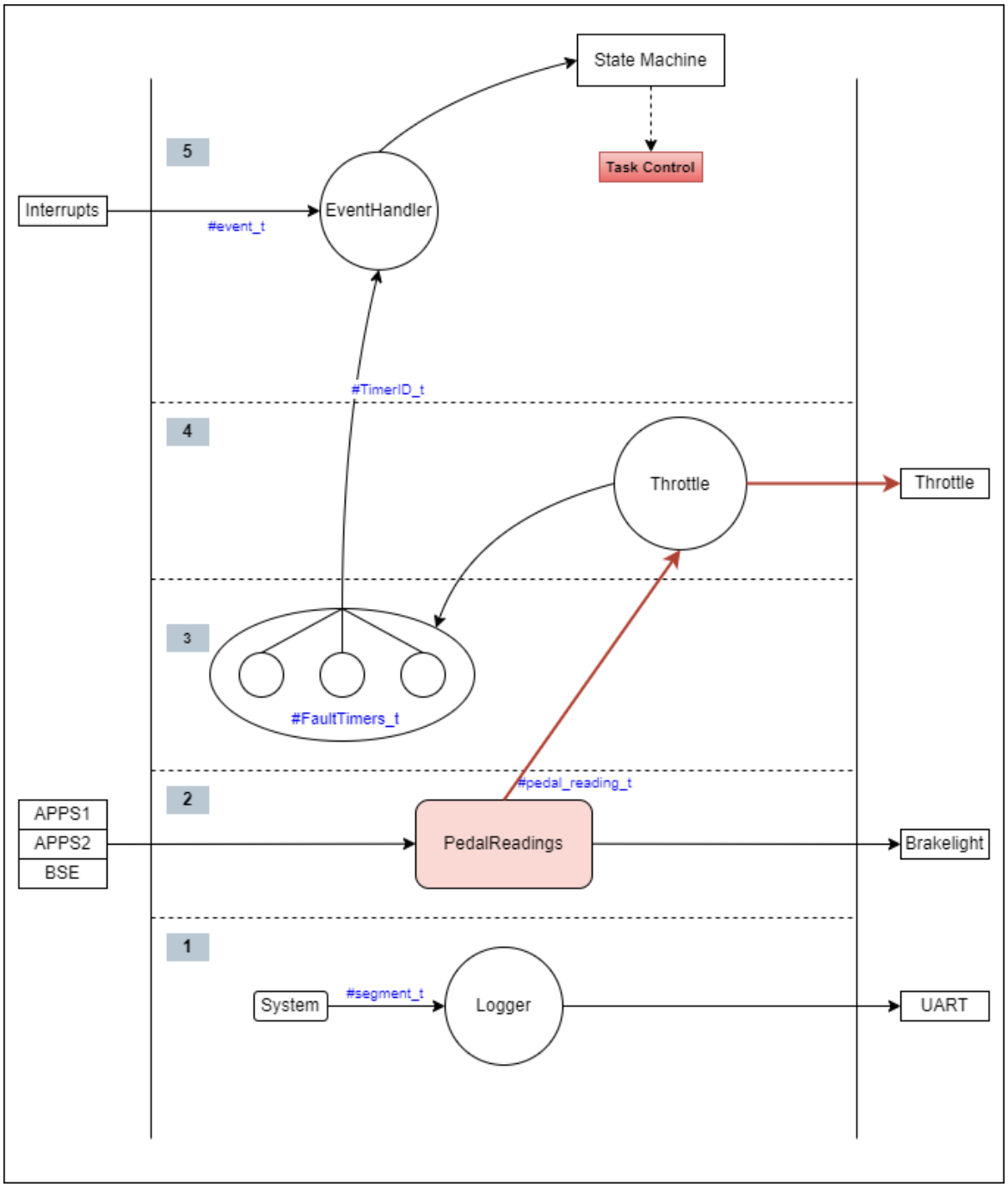
   Caching the state of the hardware or OS in variables requires constantly resynching the values and leads to the potential for stale variables to misrepresent the true state of the system. Instead we should be directly requesting the information from the hardware or OS anytime we need access to it.

4. **Centralized decision making:** Software modules should detect events and immediately request to be handled at a centralized place

   We want event detection to occur immediately after the data is produced by a lower priority task. When this task notifies the state machine, the state machine will immediately execute and make decisions about the event. By giving the state machine the single responsibility of event handling, we can keep its priority high without hogging the CPU trying to poll data as well. More details in the section, Event Pipeline

High Level Architecture

As we can see in the Firmware Architecture diagram below, each task is isolated in its own priority level. Upstream data flow, according to this priority hierarchy, is used to unblock higher level tasks in an event driven manner, and downstream control flow is used to influence the system operation.



The software architecture consists of 4 user threads and one timer thread. All threads except ***PedalReadings*** are blocked from executing until data is received in their respective queues. This allows for a minimum amount of active tasks to be running at any given moment.

During normal operation, the ***PedalReadings*** task drives the rest of the system by providing data to the ***Throttle*** task. From here, the ***Throttle*** task wakes up and maps the pedal readings to a unique voltage value for the inverter. During this process, it also starts up any fault timers if a fault condition is detected. If the faults persist, the timers will eventually expire, immediately alerting the state machine. The state machine will change the state of the system and/or suspend/resume tasks as it sees fit.

Logger

The ***Logger*** task is a non-intrusive task that accepts C-style strings through its public queue interface. It outputs the data to the serial console once there are no other user tasks running.

Pedal Readings

The ***PedalReadings*** task polls APPS and BSE signals from the ADCs and sends it to a queue to be processed by the ***Throttle*** task. It will also determine when to turn on the brake light.

Throttle

The ***Throttle*** task takes APPS and BSE readings and maps them to a voltage that it will send to the inverter. It also manages a set of fault timers by starting and stopping them based on certain fault conditions. If this task lets the timers expire, they will directly notify the state machine with an error event.

Event handler

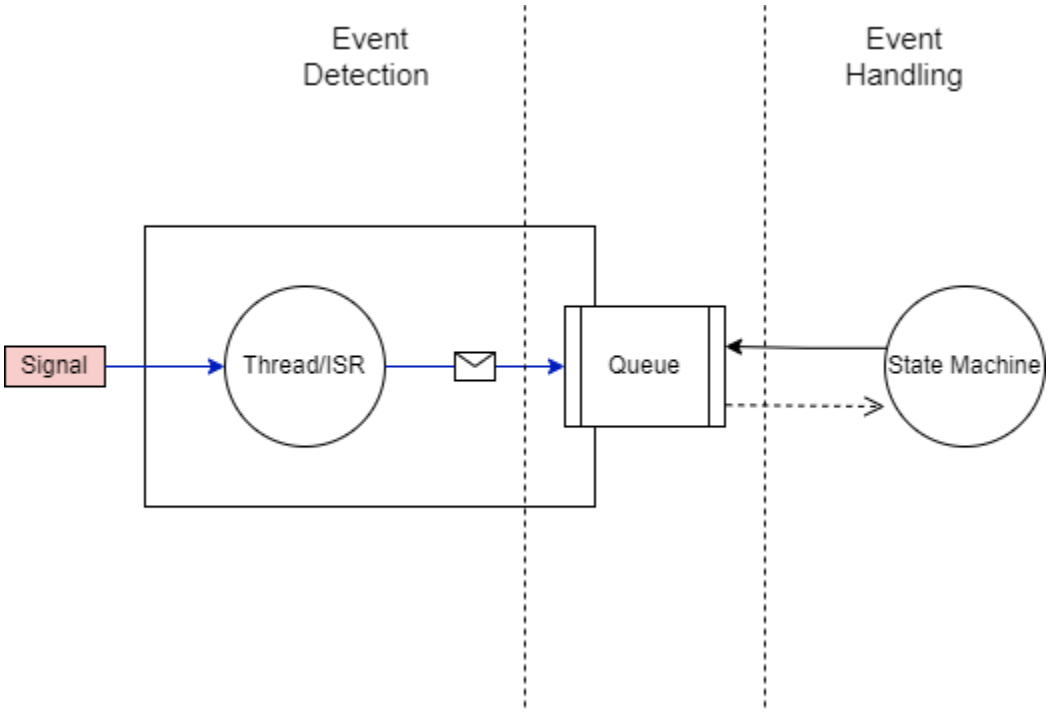
This task acts as the system's "software interrupts"; any function that is processed by the event handler will be guaranteed to execute without interruption from other tasks. This can be useful for ensuring that lower priority threads have a way to "raise the priority" of a section of their task. Hardware interrupts should also use this task's API to delay their processing.

State Machine

This is a software module that uses the event handler to receive ***eCarEvents*** from other tasks and determine the system's new state. It is responsible for handling all car events by logging event information and enabling/disabling system functionality.

Event pipeline

During normal operation, the general flow of event execution starts when a thread or ISR reads hardware signals and checks if the signal matches any user defined events. This section of the code is called **Event Detection**. If an event is detected, the task will notify the state machine to determine how the state will change. This process is referred to as **Event Handling.** There are a few benefits as to why these two processes are dealt with in separate tasks.



Operational

Separating event detection and event handling into separate tasks allows us to assign different priorities to each of them. The handling of an event can be raised to a higher level priority and will only be active if an event is detected by lower priority threads responsible for event detection.

Development

From the perspective of agile development and ever changing requirements, we isolate the hardware dependent sections so that event handling (pure software logic) can be implemented

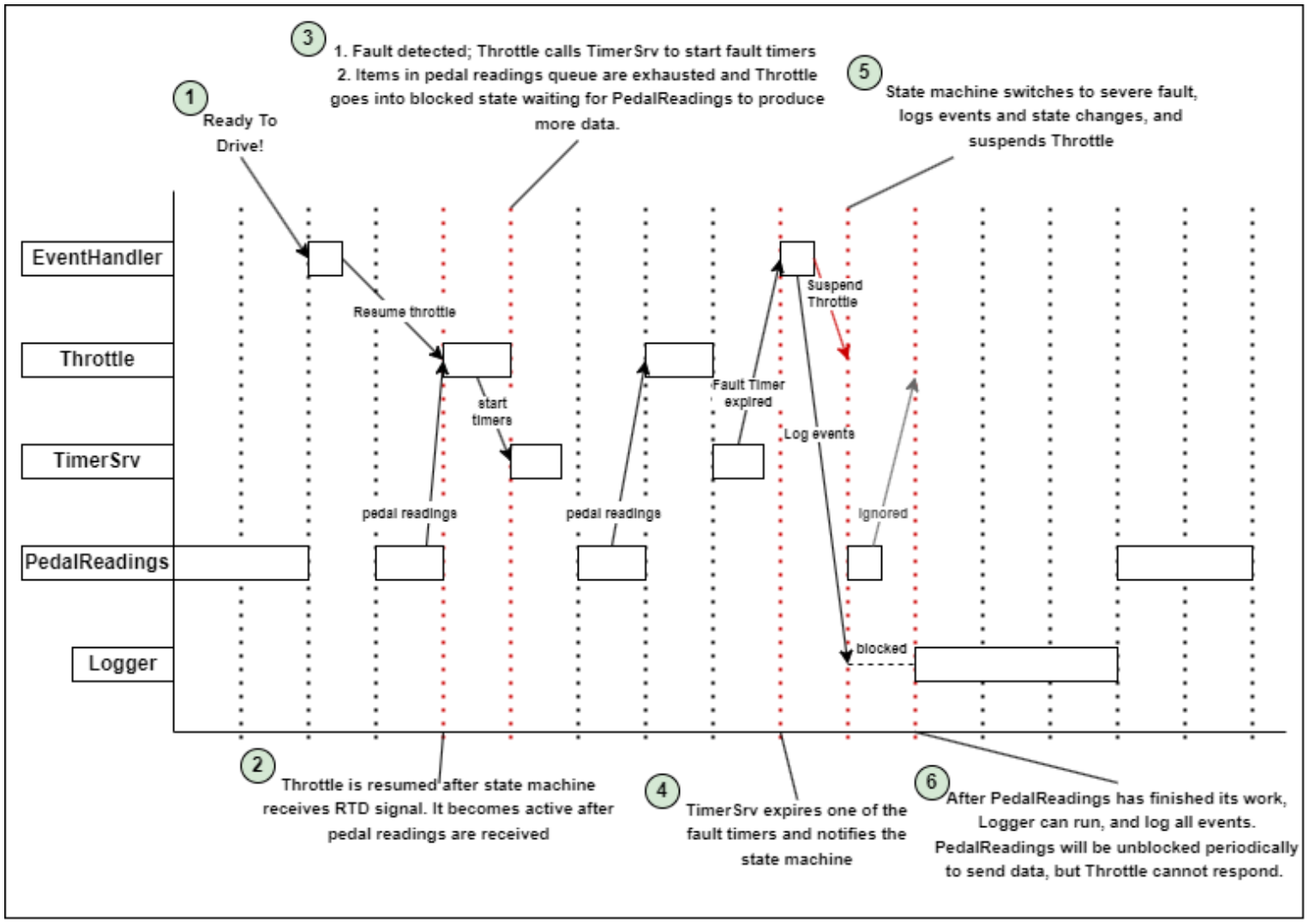
and tested before we can properly define the conditions for an event (highly dependent on hardware and rules).

Testing

Testing becomes much more robust as different build versions for stubbing functions do not need to be created to do simple tests. Instead, we can create a separate source that stubs the functionality of event detection and remains dormant unless called upon through a CLI. More details in the section, Event Injection.

Timing

In the RTOS Timing diagram below, a sample of events is used to show how the timing and priority of tasks are scheduled.



System initialization

On startup, the ***Throttle*** task is suspended and the ***PedalReadings*** task is the only task that unblocks itself periodically. Any data it queues onto the Throttle task is ignored.

Ready to Drive

A ***Ready to Drive*** event (1) is detected and notifies the state machine transitioning the system into a ready to drive state. In this state the ***Throttle*** task is resumed by the state machine. The ***Throttle*** task immediately sees data available in the queue from the ***PedalReadings*** task and becomes active again (2). During execution, it notices a fault in the hardware and starts the necessary fault timers (3).

Timer expirations

Once the ***TimerService*** is allowed to run, it notices that the fault timers have expired (they were not reset by the ***Throttle*** task because the fault was never cleared) and notifies the state machine of the event (4). The state machine determines that it should go into the ***SEVERE_FAULT state*** and suspends the throttle, setting the hardware voltage to zero and suspending the ***Throttle*** task (5).

Fault state

During these events, data is logged to the ***Logger*** task but is only run after the rest of the system has done its work. The ***PedalReadings*** task continues to unblock itself, but the system is effectively in an “idle” state because of the inactive ***Throttle*** task (6). In this state, only a reset event can put the state machine back into the idle state. From there it will need to go into the running state by receiving a ready to drive signal again.

Software Modules

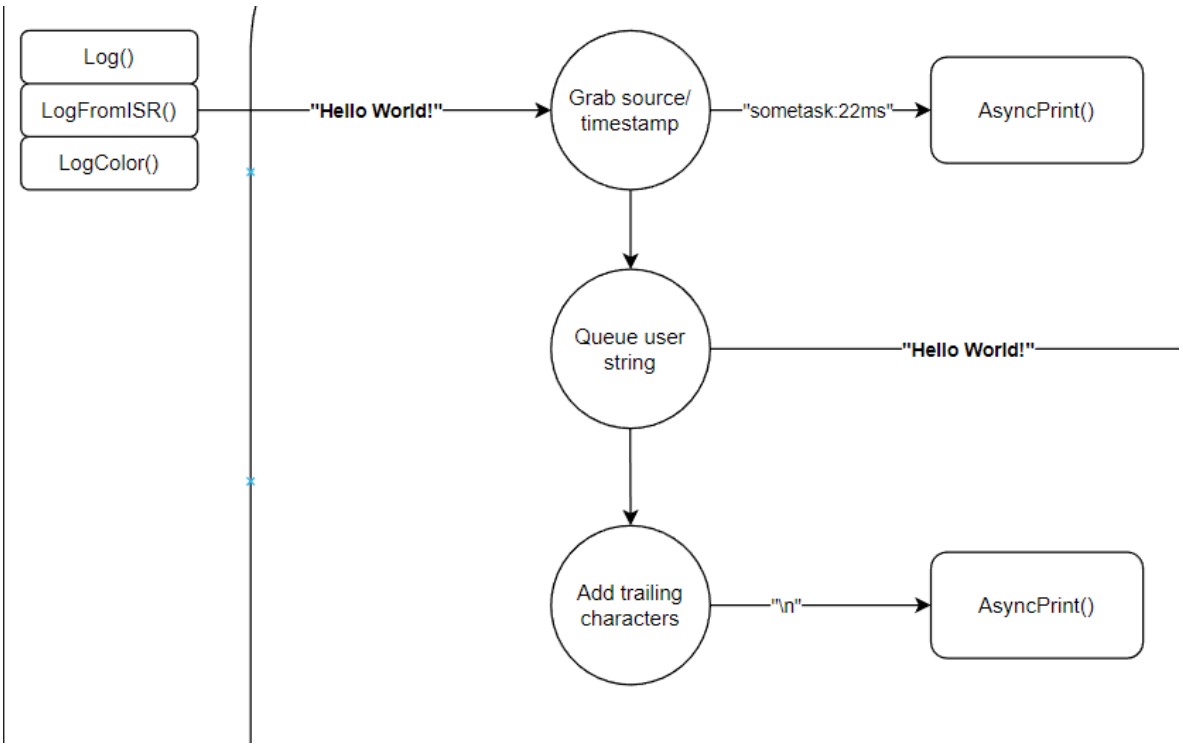
Some threads in the system live inside software modules with a well abstracted API to make it much easier to interact with. This section goes over the implementation of these modules by walking through their execution.

Logger

The Logger thread itself is quite simple; it waits for data from its queue and prints it to the serial terminal through UART. Before this happens though, the public functions must go through a procedure that helps to standardize logging and avoid dynamic memory allocation. See Logger Program Flow.

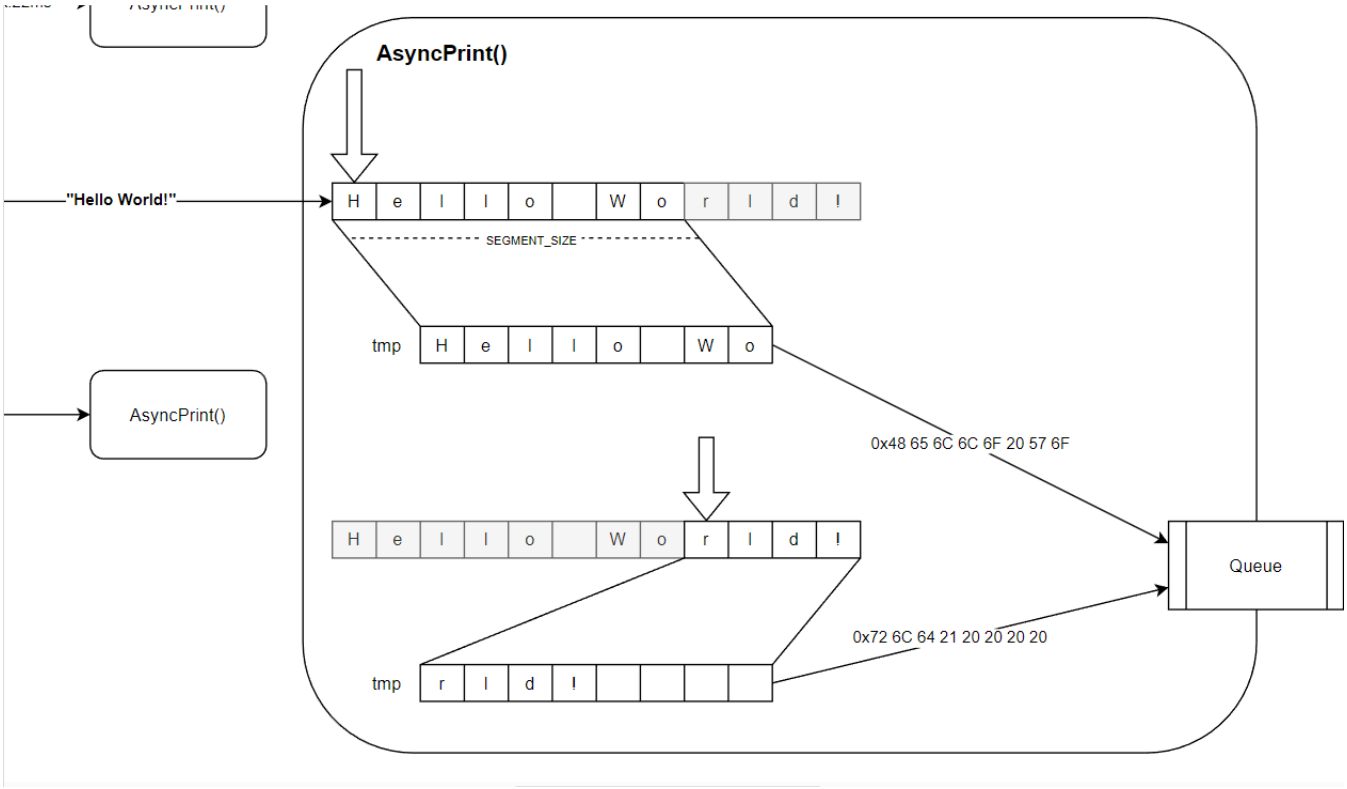
Logging context

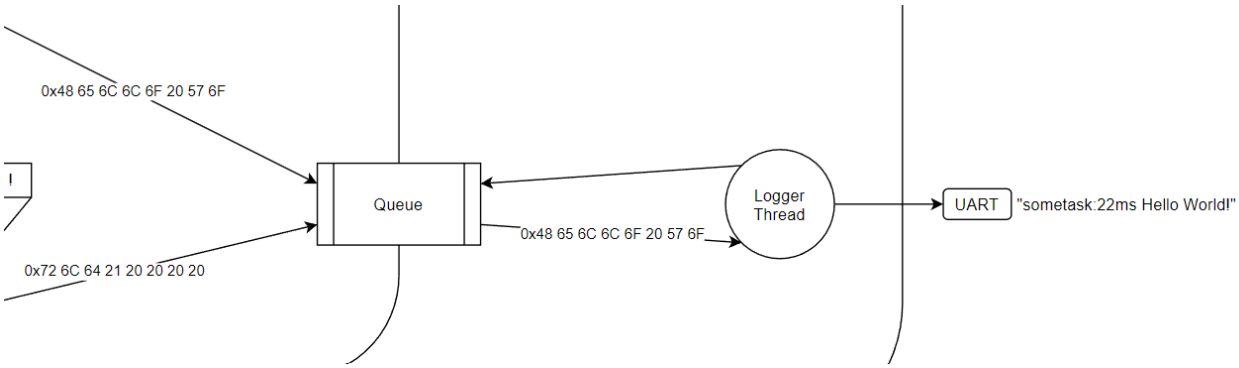
Before queueing the string passed in by the user, a log header is added first. This consists of the current tick count and the name of the current executing thread. This helps set a baseline for the information that every message logged to the console will have. Trailing characters are also added to reset any modifications made by the user's string like changing the color and styling of the terminal or adding a newline.



Queueing the data

To queue the data, we split the string into segments of 8 bytes and queue each segment. Each element in the queue has a static size and can be allocated on the stack as a *64 bit unsigned integer*. The Logger thread then retrieves this data and reinterprets it again as a string that can be sent to the serial console via UART.



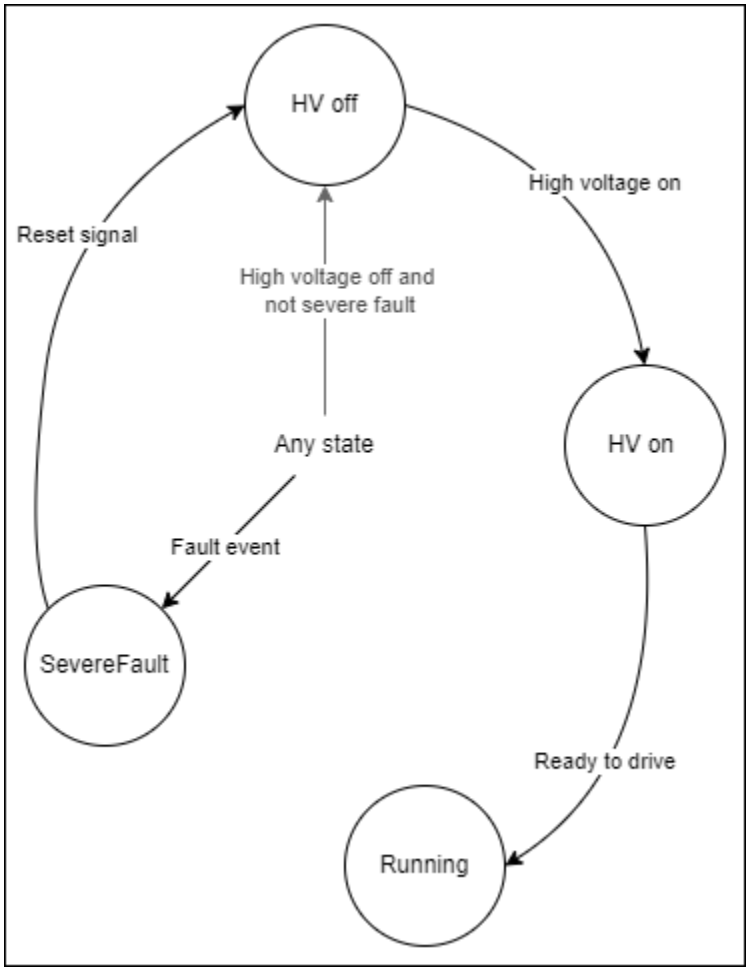


State machine

The state machine is a software representation of the state of the system used to describe ways in which the VCU can enter and exit a state. Each state may only offer a subset of the VCU's full functionality based on what it is trying to represent.

Given that the architecture is highly event driven, the state machine will be modeled as a Mealy state machine where its outputs occur on the transition between states and depends on both the state and the event.

State State Machine	**Description**
High voltage off	- Car's tractive system is off (<65V) - Throttle task is suspended
High voltage on	- Car's tractive system is on (>65V)
Running	- Ready to drive signal received by driver - Enables throttle
Severe fault	- Disables throttle task and sets inverter input to 0 - Sends shutdown signal to SDC



Developer Tools

Onboard Command Line Interface

A simple command line interface (CLI) that runs onboard the device is available for the user. For details on how to interact with it, see VCU User Manual. This section covers why the commands exist as well as their implementation.

A command is always available for the user to interact with the system via UART. The service is provided through hardware interrupts therefore no computation cycles are wasted unless a command is requested. The currently defined commands fall into roughly three categories: Variable polling, OS state information, and event injection.

Variable polling

These commands are not yet implemented as an elegant and non intrusive way of polling data that is designed to be encapsulated in software modules has not been looked into. The long term solution would be to use capture logs. It's important to implement a simple polling system first as capture logs are fairly complex for a simple debugging tool.

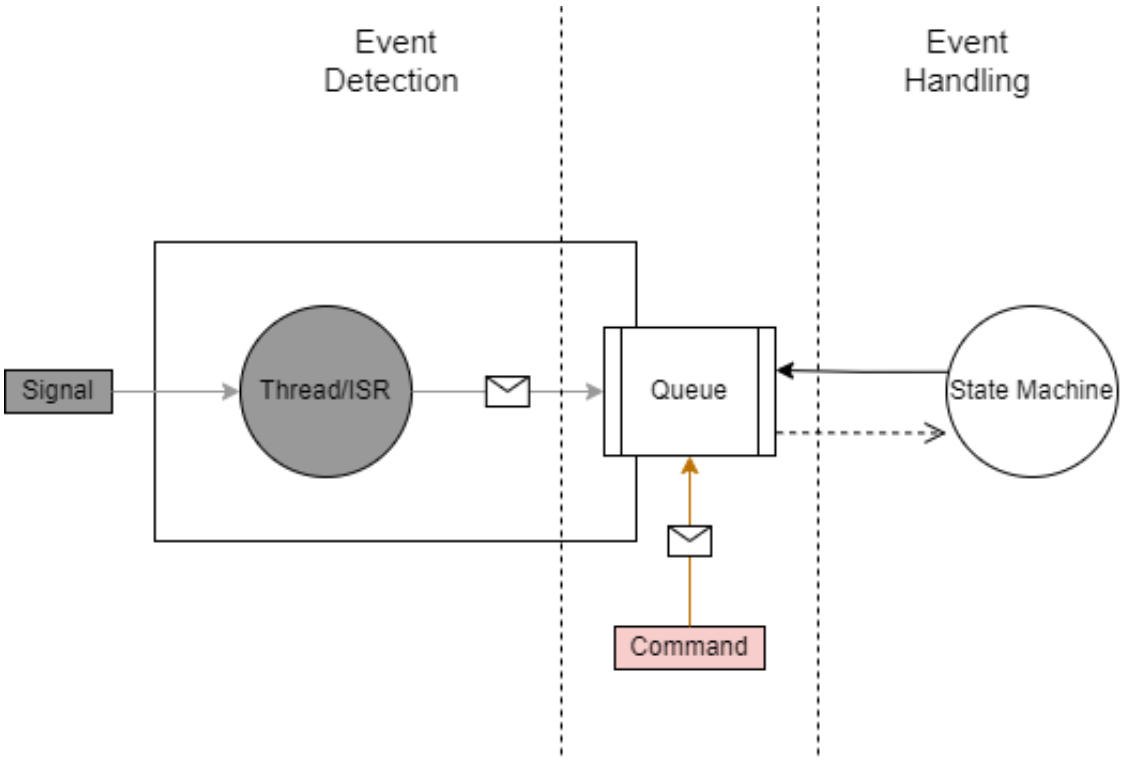
OS State

These commands directly call FreeRTOS helper functions (RTOS task (thread) utilities) to show state information at the OS level.

Event injection

These commands inject user defined events directly into the state machine to be processed and handled.

In the diagram Event Handling below, a visual representation of the separation of event processing is shown. The greyed out section on the left shows the normal flow of events in the car starting from hardware signals up to the point of notifying the state machine through its queue. The orange pathway shows a command executed via the VCU's onboard CLI. It directly injects an event into the queue for the state machine to read and bypasses this normal execution flow. In this way, the state machine and its effects on the rest of the system can be analyzed and validated immediately after defining a new event but before implementing the logic to detect it. Once it is implemented, we can validate the full event pipeline by ensuring that the state machine receives a notification from threads detecting real hardware signals.



Simulation Mode

The concept of a simulation mode for the VCU has been explored many times before. Please see the original proposal, Queue-Prototype.

A custom encoding scheme for communicating between the PC and the VCU efficiently was also created. VCU Encoding Scheme (Simulation Mode)

A snapshot of the table representative of Phantom Release 1 is provided below.

Value	Description	Bits
APPS 1	range(1500, 4500 + 1) = 3000 unique values log2(3000) ~= 12 bits Offset = 1500	0 : 11
APPS 2	range(500, 1500 + 1) = 1000 unique values	12 : 21

	log2(1000) ~= 10 bits Offset = 500	