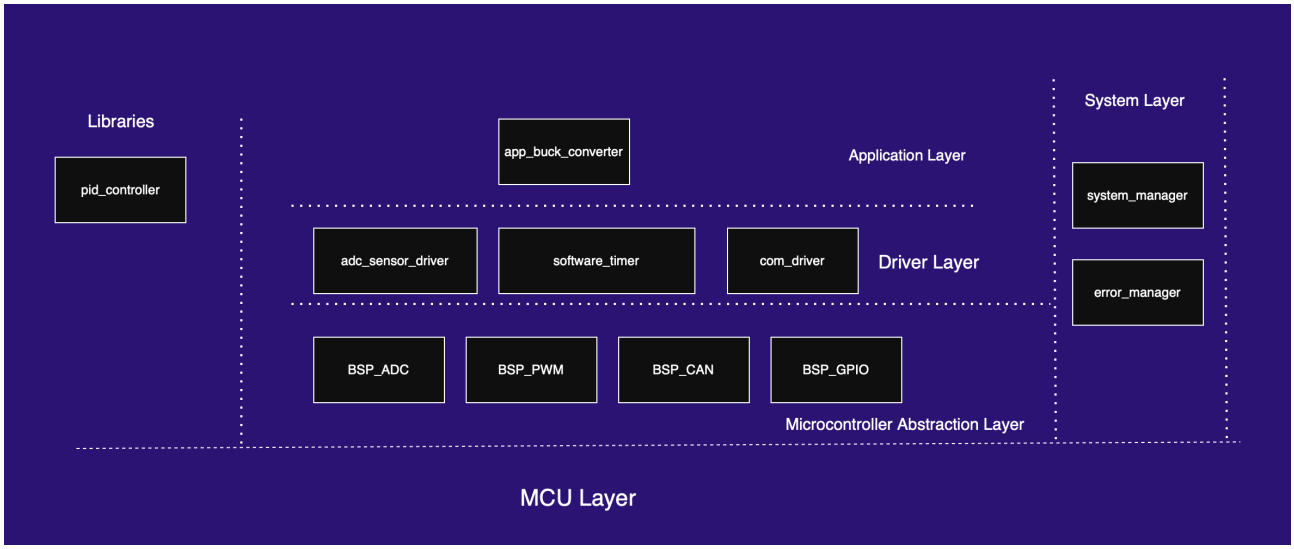
Software Architecture Layer View

1. MCU Abstraction Layer



Description:
The **MCU Abstraction Layer** acts as a thin hardware abstraction layer that isolates the upper software layers from the microcontroller's hardware-specific details. This layer provides a standardized interface to access peripherals such as ADCs, GPIOs, PWMs, and CAN communication. By using this layer, developers can modify or upgrade the underlying hardware with minimal impact on higher-level modules, which increases portability and maintainability. This layer is typically implemented using low-level register access and is tightly coupled to the hardware.

Modules:

- `bsp_can`: Provides initialization and low-level access to the CAN (Controller Area Network) peripheral of the MCU.
- `bsp_gpio`: Offers APIs for configuring and manipulating digital I/O pins, including setting pin direction, pull-up/down, and reading/writing pin states.
- `bsp_pwm`: Configures and controls Pulse Width Modulation channels, typically used for motor control or power regulation.
- `bsp_adc`: Interfaces with the MCU's ADC module to read analog values from sensors and convert them into digital values.

2. Driver Layer

Description:
The **Driver Layer** is responsible for implementing peripheral drivers and software modules that use or extend the functionality provided by the MCU abstraction layer. This layer provides structured and reusable software blocks to interact with sensors, timers, and communication interfaces. Drivers in this layer are generally independent of the specific application and aim to encapsulate hardware behavior in a modular way. This promotes code reuse and simplifies testing and validation.

Modules:

- `adc_sensors_driver`: Manages multiple analog sensors, handling ADC conversions, filtering, and data normalization.
- `software_timer`: Provides software-based timing utilities using hardware timers or counters, useful for scheduling tasks or implementing timeouts.
- `communication_driver`: Manages data transmission and reception over communication interfaces (e.g., CAN, UART), including message parsing and buffering.

3. Application Layer

Description:
The **Application Layer** contains the high-level logic that defines the behavior and goals of the system. It orchestrates the operation of the entire software stack by leveraging services from lower layers. This layer is specific to the use case or end-product and implements core functionalities such as power conversion, control loops, or user interfaces. It integrates business logic and makes direct use of drivers, libraries, and system services.

Modules:

- `app_buck_converter`: Implements the control logic of a buck (step-down) converter, including voltage regulation, switching logic, and monitoring routines.

4. Libraries Layer

Description:
The **Libraries Layer** provides reusable and general-purpose algorithms, mathematical tools, and control logic components that are not tied to a specific hardware platform or application. These libraries are designed to be modular and easily integrated into different projects. They are typically well-tested and can include digital filters, PID controllers, mathematical utilities, or protocol parsers. This layer enhances code modularity, maintainability, and development speed.

Modules:

- `pid_controller`: A generic Proportional-Integral-Derivative (PID) controller implementation used to regulate system parameters such as voltage, current, or temperature.

5. System Layer

Description:
The **System Layer** is responsible for coordinating and managing the overall operation of the embedded system. It typically includes initialization routines, runtime task management, fault handling, and system state supervision. The components in this layer ensure that all subsystems function together reliably and consistently. It often contains safety mechanisms, diagnostics, and hooks for system-wide configuration and monitoring. This layer provides the backbone of the runtime environment.

Modules:

- `system_manager`: Handles the initialization and coordination of all subsystems, manages system states, and controls power modes.

- `error_manager`: Monitors system faults, logs errors, and provides fault recovery mechanisms to maintain safe operation.