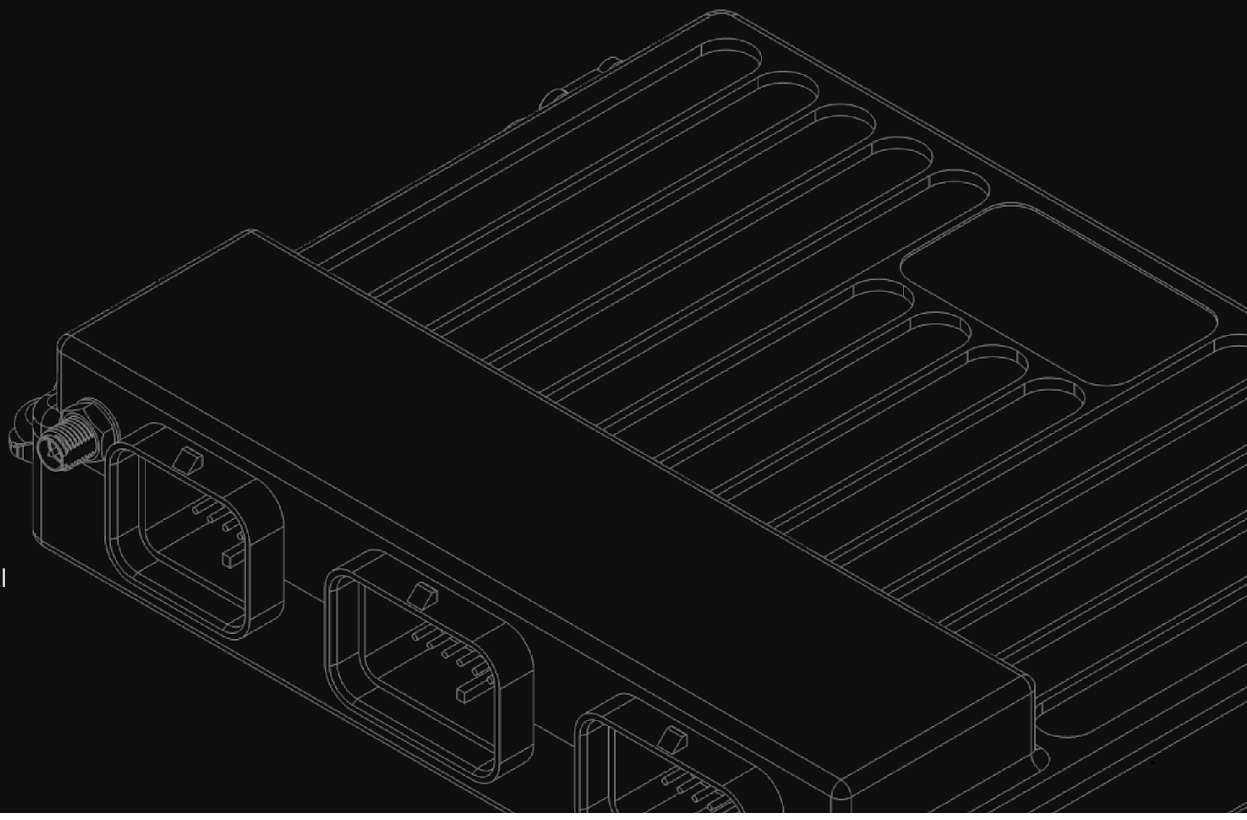
DYNAM LABS

>VCU+



Feb 5, 2025

Plugin Reference Manual

Overview

Plugins enable users to add custom functionality to the VCU+ while keeping its core firmware lightweight and efficient. By allowing customization without modifying the core, plugins ensure the system remains maintainable and adaptable to a wide range of application-specific needs.

With plugins, users can implement tailored features, such as custom control algorithms or data processing workflows, without the overhead of embedding these directly into the core firmware. This flexibility empowers users to quickly adapt the VCU to their unique requirements.

This reference manual provides a comprehensive guide to using plugins, including an overview of available functions, configuration options, and instructions for writing, testing, and deploying custom logic.

The plugin language is **Lua**, chosen for its simplicity, low learning curve, and lightweight nature. Lua is widely regarded for its ease of integration into embedded systems and its ability to execute scripts efficiently, making it ideal for the resource-constrained environment of the VCU+. Its straightforward syntax and minimal setup allow both novice and experienced developers to quickly write and test custom logic.
For more information about Lua, refer to the official documentation:

- Official Lua Documentation

This reference manual provides a comprehensive guide to using plugins, including an overview of available functions, configuration options, and instructions for writing, testing, and deploying custom Lua-based logic.

Functions

The following is a list of all currently supported functions available for use in plugins. These functions form the building blocks for creating powerful and highly customized logic within your plugins, giving you full control over VCU outputs, sensor data, communication, and control algorithms. The list is continuously growing as more functions are added.

setOut(pin)

This function sets the specified output pin to its active state:

- **High-Side Outputs**: The pin transitions to a **high state** (voltage applied).
- **Low-Side Outputs**: The pin transitions to a **low state** (connected to ground).

Example Use Case: Activating a high-side output pin to power a relay or a low-side output pin for devices requiring a ground-side switch.
Sample Code:

```
-- Activate a high-side output 1
setOut(HS1)
-- Activate a low-side output 2
setOut(LS2)
```

clearOut(pin)

This function clears the specified output pin, setting it to its inactive state:

- **High-Side Outputs**: The pin transitions to a **low state** (voltage removed).
- **Low-Side Outputs**: The pin transitions to a **high state** (disconnected from ground).

Example Use Case: Deactivating a relay or turning off a light connected to the VCU.
Sample Code:

```
-- Deactivate the high-side output 1
clearOut(HS1)
-- Deactivate the low-side output 2
clearOut(LS2)
```

getSensor(sensorID)

Retrieves the current value of the specified sensor. This allows the plugin to access real-time data, such as temperature, speed, or pressure.
Returns nil in case of an invalid Sensor state. Ensure to check for return value before using the value for arithmetic operations.
Example Use Case: Reading a temperature sensor value for dynamic control logic.
List of available Sensors:

• AcceleratorPedal • AcceleratorPedalPrimary • AcceleratorPedalSecondary • BatteryCurrent • BatteryVoltage • BatteryTemp • DI_inverterTemp • DI_statorTemp • DI_heatSinkTemp • DI_inletTemp • DIS_inverterTemp • DIS_statorTemp • DIS_heatSinkTemp • DIS_inletTemp • Temp1 • Temp2	• Temp3 • Temp4 • RPM • Din1 • Din2 • Din3 • Din4 • Din5 • Din6 • Din7 • Din8 • Din9 • Din10 • Din11 • Din12 • Din13 • Din14 • Din15	• LVBatt • CAN_KEYPAD_B1 • CAN_KEYPAD_B2 • CAN_KEYPAD_B3 • CAN_KEYPAD_B4 • CAN_KEYPAD_B5 • CAN_KEYPAD_B6 • CAN_KEYPAD_B7 • CAN_KEYPAD_B8 • Ignition • Ain1 • Ain2 • Ain3 • Ain4 • Ain5 • Ain6 • Ain7 • Ain8 • Gear

Sample Code:

```
-- Retrieve the value from Temperature sensor 1
local temperature = getSensor(Temp1)
-- Retrieve Accelerator pedal value and use it in logic
local pedal = getSensor(AcceleratorPedal)
```

sendCAN(canBus, canID, data)

Sends a CAN message with the specified ID and data payload. Enables communication with other devices on the CAN bus.

- **canBus**: The physical bus to send the message on. (ACC_CAN or AUX_CAN)
- **canID**: The identifier of the CAN message.
- **data**: The payload, typically a byte array.

Example Use Case: Sending commands to external controllers or reporting custom data.
Sample Code:

```
-- Send a CAN message on ACC_CAN with ID 0x123 and data payload {0x01, 0x02, 0x03}
sendCAN(ACC_CAN, 0x123, {0x01, 0x02, 0x03})
-- Example of sending a command to an external device on the AUX_CAN bus
sendCAN(AUX_CAN, 0x200, {0xFF, 0xAA, 0x55})
```

newPID(p, i, d)

Creates a new PID controller instance with the specified tuning parameters.

- **p**: Proportional gain.
- **i**: Integral gain.
- **d**: Derivative gain.

Example Use Case: Initializing a PID controller to maintain target speeds or stabilize temperatures.
Sample Code:

```
-- Create a PID controller with P=1.0, I=0.5, D=0.1
local pid = newPID(1.0, 0.5, 0.1)
```

pid:compute(target, actual)

Computes the control output of an existing PID controller based on a target value and the actual measured value. This function is supposed to be called in a loop.

- **target**: The desired setpoint value.
- **actual**: The current measured value.

Example Use Case: Adjusting a motor speed to match the desired target speed.
Sample Code:

```
-- Create a PID controller
local pid = newPID(1.0, 0.5, 0.1)
-- Compute the control output
local targetTemp = 100  -- Target temperature
local actualTemp = getSensor(Temp1)  -- Assume Temp1 provides actual temperature
local controlOutput = pid:compute(targetTemp, actualTemp)
```

setPWMFreq(channel, frequency)

Sets the frequency of the specified PWM (Pulse Width Modulation) channel in Hertz (Hz). This determines how many PWM cycles are generated per second.

- **channel**: The PWM channel to configure (e.g., PWM1, PWM2).
- **frequency**: The desired frequency in Hertz. Higher frequencies provide smoother outputs but may reduce resolution for duty cycle adjustments.

Example Use Case: Setting the PWM frequency for a fan or pump controller.

print(string)

Prints a message to the plugin log inside of Bolt.

- **string**: String to print

Example Use Case: Debug print statement during plugin development.
Sample Code:

```
local ain3_val = getSensor(Ain3)
print("Ain3 Value: " .. ain_val)
```