Programming essentials for the CPP SDK

PN149 | Posted on June 25, 2026 | Updated on June 25, 2026

Jessy ANÇAY
Sales & Project Engineer
imperix

Table of Contents

Taking a Three-phase PV inverter as a reference application, this note provides programming essentials for the CPP SDK to facilitate the implementation of full-scale converter control algorithms. While foundational C++ development is covered in the Getting started with the CPP SDK, this page focuses on providing guidelines and practical code snippets to go further with programming using the CPP SDK.

Specifically, this article addresses common power electronics control strategies, state machine management, background tasks for multirate control, communication interfaces, and control validation recommendations. The provided example encapsulates all of these concepts, serving as a comprehensive reference and a robust foundation for CPP SDK developers.

CPP SDK programming insights

When developing a new control algorithm for imperix controllers, the following two-step workflow is recommended to get started efficiently:

- **Software documentation:** Users should refer to the software documentation and/or the CPP SDK header files to understand the available hardware peripherals and corresponding software routines.
- **Peripherals and control:** It is recommended to first configure the peripherals in the initialization routine. Then, the control algorithms can be implemented within the main real-time loop.

Development Tip: To bridge the gap between the basic template and a functioning system, it is recommended to draw inspiration from existing imperix examples. Even ACG-based examples provide valuable reference for control strategies.

With this mindset, this page dissects the C++ control file from the Three-phase PV inverter for grid-tied applications example, as it covers most of the essential programming insights needed to use the CPP SDK. The complete C++ project is provided below.

Download **Central_PV_Inverter_CPP**

For additional details regarding the converter topology and the control strategy, users can refer to the aforementioned article.

Common power electronics control strategies

To facilitate the development of control algorithms, the CPP SDK includes a dedicated API folder with pre-validated power electronics functions, such as PI controllers, PLLs, and coordinate transformations. Developers are encouraged to use these standard functions, which they can modify as needed.

The following few paragraphs will showcase how to use a few of the provided functions.

PI controller

The provided PI controller should first be declared as a global variable, in this case `Ipv_reg`

```
float Vpv;                      // Solar panel voltage measurement
float Ipv = 0;                  // Solar panel current measurement
float Ipv_ref = 0;              // Solar panel current reference

//Global variables in a namespace will not show up in Cockpit
namespace{
    PIDController Ipv_reg;  // Controller for the PV current control
    float Eb;               // Boost switching voltage
};

// ...Code language: C++ (cpp)
```

It must then be initialized within the `UserInit()` routine using the appropriate parameters.

```
tUserSafe UserInit(void)
{
    // ...

    ConfigPIDController(&Ipv_reg, Kp_Ipv, Ki_Ipv, 0, 800, -800, SAMPLING_PERIOD, 10);

    // ...
}Code language: C++ (cpp)
```

Finally, the controller can be executed within the main interrupt routine.

```
tUserSafe UserInterrupt(void)
{
    // ...

    // Execute the current controllers on the MPPT strings:
    Eb = Vpv - RunPIController(&Ipv_reg, Ipv_ref - Ipv);

    // ...
}Code language: C++ (cpp)
```

It is worth mentioning that the implemented PI controller, as shown above, already includes an anti-windup strategy, and its integrator is automatically reset when the controller is not operating, using the `GetCoreState()` method that returns the operating state of the controller.

DQ PLL and Park transform

Similar to the PI controller, the DQ PLL must first be declared as a global variable alongside the necessary state variables and voltage vectors.

```
DQPLLParameters DQPLL;          // DQ PLL for grid synchronization
float Theta;                    // Phase angle of the grid voltage
float w_grid;                   // Grid angular frequency ($\omega$)
float Kp_pll, Ki_pll;           // PLL PI gains

TimeDomain Vg_abc;              // Three-phase grid voltage measurements
SpaceVector Vg_dq0;             // Voltages in the dq0 reference frame

// ...Code language: C++ (cpp)
```

The PLL must then be initialized within the `UserInit()` routine using the desired proportional/integral gains, the nominal grid frequency, and the sampling period.

```
tUserSafe UserInit(void){
    // ...

    // Initialize the DQ PLL
    ConfigDQPLL(&DQPLL, Kp_pll, Ki_pll, OMEGA, SAMPLING_PERIOD);

    // ...
}Code language: C++ (cpp)
```

Finally, the coordinate transformation and the PLL can be executed within the main interrupt. The three-phase measurements are directly transformed into the synchronous reference frame (dq0) and the PLL extracts the grid angle.

```
tUserSafe UserInterrupt(void){
// ...

    // 1. Apply the direct transformation (abc to dq0) using the previous angle
    abc2DQ0(&Vg_dq0, &Vg_abc, Theta);

    // 2. Execute the PLL to track the grid angle
    Theta = RunDQPLL(&DQPLL, &Vg_dq0);
    w_grid = (&DQPLL)->omega;
```

```
// ...
}Code language: C++ (cpp)
```

State machine implementations

Power converter control often requires managing different operational states, such as *standby*, *precharging*, *operating*, *discharging*, *fault*, and many more. State machines are powerful tools that allow to manage these transitions safely.

They can be implemented in many ways. The following snippet serves as a practical example. It details the state machine handling the different operational states of the PV inverter.

Operations state machine

```
tStateOperation State_operation = OP_INIT; // Current state of the operational state machine
tStateOperation Next_state_operation = OP_STANDBY; // Next state of the operational state machine
unsigned int    state_operation_uint; // Current state of the operation state machine for monitoring in Cockpit

void User_RunOperationFSM()
{
        switch (Next_state_operation){
        // Init
        case OP_INIT:
                State_operation = OP_INIT;
                Next_state_operation = OP_STANDBY;
                break;

        // The converter is in standby and is waiting to be turned on, everything is disabled
        case OP_STANDBY:
                if (State_operation != Next_state_operation)
                                        Log_SendMsg(5, NULL, 0);
                State_operation = Next_state_operation;

                activate_boost = 0;
                activate_inverter = 0;

                if (activate == 1 && core_state > 0)
                {
                        Next_state_operation = OP_WAITING_ON_PRECHARGE;
                }

                break;

        // The converter is started and is waiting for the precharge procedure to complete
        case OP_WAITING_ON_PRECHARGE:
                if (State_operation != Next_state_operation)
                {
                        Log_SendMsg(6, NULL, 0);
                        operation_cnt = 0;
                }
                else
                {
                        operation_cnt++;
                }
                State_operation = Next_state_operation;

                activate_boost = 0;
                activate_inverter = 0;

                if (core_state > 0 && Precharge_ready && operation_cnt > 1*SW_FREQ)
                {
                        Next_state_operation = OP_READY_TO_OPERATE;
                }
                else if (activate == 0)
                {
                        Next_state_operation = OP_STANDBY;
                }
                else if (Precharge_fault || core_state == 0 || operation_cnt > 30*SW_FREQ)
                {
                        Next_state_operation = OP_FAULT;
                }

                break;

        // The converter is fully turned on, the PV relay is closed, the boost converter, the inverter and the PWM
        case OP_READY_TO_OPERATE:
                if (State_operation != Next_state_operation)
                        Log_SendMsg(7, NULL, 0);
```

```
                State_operation = Next_state_operation;

                activate_boost = 1;
                activate_inverter = 1;

                if (activate == 0)
                {
                        Next_state_operation = OP_STANDBY;
                }
                else if (Precharge_fault || core_state == 0)
                {
                        Next_state_operation = OP_FAULT;
                }

                break;

        // A precharge or core fault was detected
        case OP_FAULT:
                if (State_operation != Next_state_operation)
                        Log_SendMsg(8, NULL, 0);
                State_operation = Next_state_operation;

                activate_boost = 0;
                activate_inverter = 0;

                if (Precharge_fault == 0 && core_state > 0)
                {
                        Next_state_operation = OP_STANDBY;
                }

                break;
        }

        state_operation_uint = (unsigned int) State_operation;
}
```
Code language: C++ (cpp)

Programmatic activation of PWM signals

Following the implementation of a state machine, developers may look to automate the activation of PWM signals. While this action is usually performed manually via the dedicated Cockpit button, it is possible to do it programmatically, using the `CoreStart()` and `CoreStop()` functions. When combined with state machine logic, these functions enable fully automated converter operation.

```
/**
* /!\ Caution /!\
* Please check that the B-Box hardware protection limits are properly configured
* to avoid causing irreversible damage to the converter.
*/
if(enable_pwm) CoreStart();
else CoreStop();
```
Code language: C++ (cpp)

Programmatically enabling the PWM outputs must be done with the utmost precaution to avoid causing serious damage to the converter. Imperix recommends using the Cockpit button when possible.

System logging and diagnostics

User log messages are highly useful for tasks such as tracking state machine transitions or reporting converter faults. These messages can be configured and displayed in Cockpit using the `Log_AddMsg` and `Log_SendMsg` functions.

However, simply placing `Log_SendMsg` inside the main interrupt will continuously spam the Cockpit Log tab at the interrupt's operating frequency, eventually overflowing the buffer. To avoid this, developers must implement logic to generate discrete trigger events. The following code demonstrates one approach to trigger these logging messages.

Log message

```
tUserSafe UserInit(void)
{
    //...
    // Configure a warning message with a unique id of 0
    Log_AddMsg(0, 20, "Operating limits exceeded (V=%.3fV / I=%.3fA)");
    //...
    return SAFE;
}

tUserSafe UserInterrupt(void)
```

```
{
    //...

    static bool warning_message_sent = false;
    if(V_meas > 850 || I_meas > 40){
        float log_values[2];
        log_values[0] = V_meas;
        log_values[1] = I_meas;
        // Display the message in Cockpit
        if(!warning_message_sent) {
            Log_SendMsg(0, log_values, 2);
            warning_message_sent = true;
        }
    } else {
        warning_message_sent= false;
    }

    //...

    return SAFE;
}Code language: C++ (cpp)
```

Background tasks and multirate control

Tasks such as MPPT, thermal monitoring, or background communication may not require execution at the strict, high-frequency rate of the primary control interrupt. To manage computationally heavy or slow-rate tasks, the CPP SDK provides a background callback routine, typically implemented as `UserBackground()` that executes during the CPU's idle time.

For this specific PV inverter application, the background routine is used to implement an MPPT algorithm executed at 200Hz and is detailed below.

To set up a background loop with the CPP SDK, it is first required to declare a background loop function prototype:

```
// Background loop prototype
tUserSafe UserBackground();Code language: C++ (cpp)
```

This function then needs to be registered during the initialization routine in `UserInit();`

```
tUserSafe UserInit(void){
  /**
   * Configuration of the main interrupt:
   * - CLOCK_0 is set to the desired frequency
   * - The main interrupt is mapped on CLOCK_0.
   * - Register the background loop function
   */
  Clock_SetFrequency(CLOCK_0, SW_FREQ);
  ConfigureMainInterrupt(UserInterrupt, CLOCK_0, 0.5);
  RegisterBackgroundCallback(UserBackground);

  //  ...
}Code language: C++ (cpp)
```

The background loop is executed as fast as possible during the CPU's idle time. It runs with a lower priority than the main interrupt and therefore does not compromise the deterministic execution of the real-time control interrupt.

While its base execution rate is unguaranteed, developers can easily implement multirate control by using a software timer inside the main interrupt. This timer would raise a flag to trigger the background task at periodic intervals, as shown below:

```
void User_BackgroundLoopTimer(){
  // Increment timer and trigger flag for background loop
  SubTaskTimer += SAMPLING_PERIOD;
  if(SubTaskTimer >= MPPT_PERIOD){
    SubTaskTimer = SubTaskTimer - BACKGROUND_LOOP_PERIOD;
    SubTaskFlag  = true;
    }
}Code language: C++ (cpp)
```

Finally, the background loop function can be defined to execute only when the periodic flag is raised:

```
// Background loop function
tUserSafe UserBackground(){
  if(SubTaskFlag){
    // Insert your code here

    SubTaskFlag = false;
  }
```

```
  return SAFE;
}Code language: C++ (cpp)
```

Software user faults

While hardware protections handle instantaneous electrical limits, additional safety conditions, such as communication timeouts or failed precharge sequences, may require software-defined faults.

For this, the CPP SDK allows users to trigger custom software faults using the `SetUserFault(const char* user_txt)` function. These faults are integrated directly into the fault manager of imperix controllers. When triggered, they immediately disable PWM signals (although not as rapidly as hardware protections) and can display a custom message in the Cockpit logs. The controller will remain locked in the `FAULT` state for as long as this function is actively called.

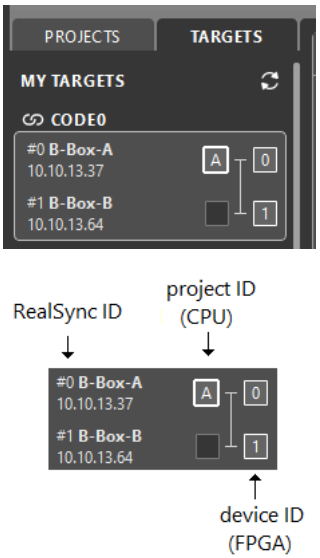
```
if(V_meas > V_max){
    SetUserFault("Maximum voltage exceeded");
}Code language: C++ (cpp)
```

As an alternative, it is also possible to trigger a user fault by returning the state `UNSAFE` within the user interrupt. Returning `SAFE` implies that no errors happened during execution.

Note that returning `UNSAFE` will generate a generic fault. It is then usually recommended to use the `SetUserFault(const char* user_txt)` method to be able to display a specific fault message.

Specifying the number of devices used

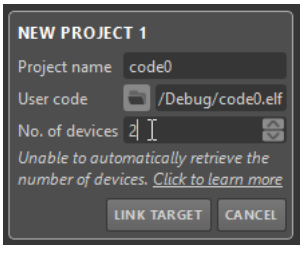
Since SDK 2025.1, as shown in the image below, Cockpit reads how many devices are used by the user code to properly display the assignment between the user code (project) and targets.



When using the ACG SDK, the number of used devices is automatically retrieved. With the CPP SDK, however, users must explicitly specify this information using the `NUMBER_OF_DEVICES` macro, as illustrated below. This is particularly needed when developing code intended for multiple targets, as the default value is set to one.

```
/**
 * Specify the number of devices used by this code.
 * This information is used by Cockpit for display purposes only.
 */
NUMBER_OF_DEVICES(1);
```

Alternatively, this information can be manually edited from Cockpit, using the *No. of devices* field.



If the Eclipse IDE underlines the NUMBER_OF_DEVICES in orange, please try the following:
Right-click on the project → Index → Rebuild. This resets the project's code cache and wipes away the false warning lines.

Control validation and debugging

Because offline simulation is not available for CPP SDK users, the developed control must be validated directly on the physical hardware. Consequently, it is strictly even more critical to configure robust hardware and software protections prior to testing (refer to Over-current and over-voltage protection documentation). Indeed, without a simulated environment, development relies heavily on incremental physical validation.

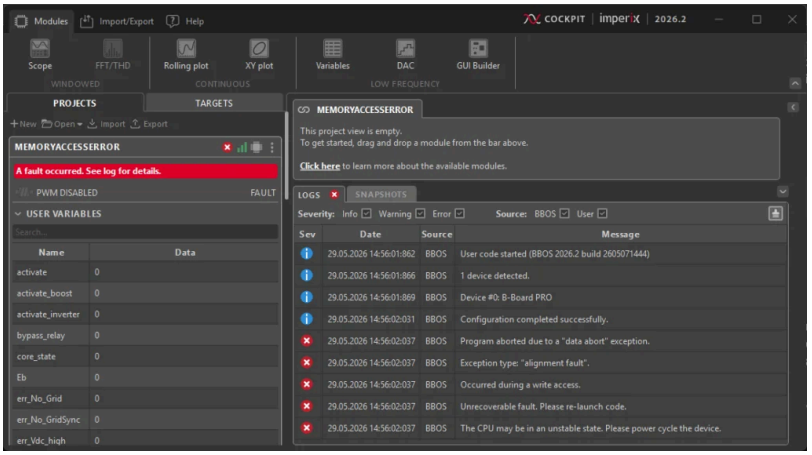
Use particular caution when passive rectification cannot be avoided (e.g., grid-tied inverters), against which hardware protections are ineffective. Further details on this topic are given in TN131.

Memory access errors

Memory access violations occur more frequently when using the CPP SDK, most often due to the misuse of pointers in C++. When an invalid memory address is accessed, a hardware exception is triggered by the processor (further details can be found in the ARM Cortex Fault Handling documentation). When such an event is triggered:

- The CPU is safely hard-locked so that erratic control behavior is prevented.
- A fatal error message detailing the fault is displayed within the Cockpit logs.
- A restart of the controller is required for system recovery.

To guarantee system safety during these CPU lockups, imperix controllers are equipped with a dedicated FPGA Watchdog. In the event that the CPU becomes unresponsive, in the case of a hardware exception for instance, this watchdog automatically intervenes to immediately disable all active PWM signals, ensuring no hardware damage.



Further readings

- Programming and operating imperix controllers (PN138) is a guide for deploying code onto imperix controllers
- Cockpit user guide (PN300) gives an overview of the tools provided by Cockpit and how to use them