Getting started with the CPP SDK

PN148 | Posted on June 25, 2026 | Updated on June 25, 2026

Jessy ANÇAY
Sales & Project Engineer
imperix

Table of Contents

- Product description
- CPP SDK Programming workflow
- Development with the C++ IDE
  - Navigate the IDE
  - User template structure and content
- CPP SDK coding example
  - Global variables
  - UserInit() function
  - UserInterrupt() function
- Further reading

This article, targeted towards first-time imperix users, provides instructions to get started with the CPP SDK. It focuses on the typical workflow associated with the development of C++ control algorithms for imperix controllers. It notably addresses:

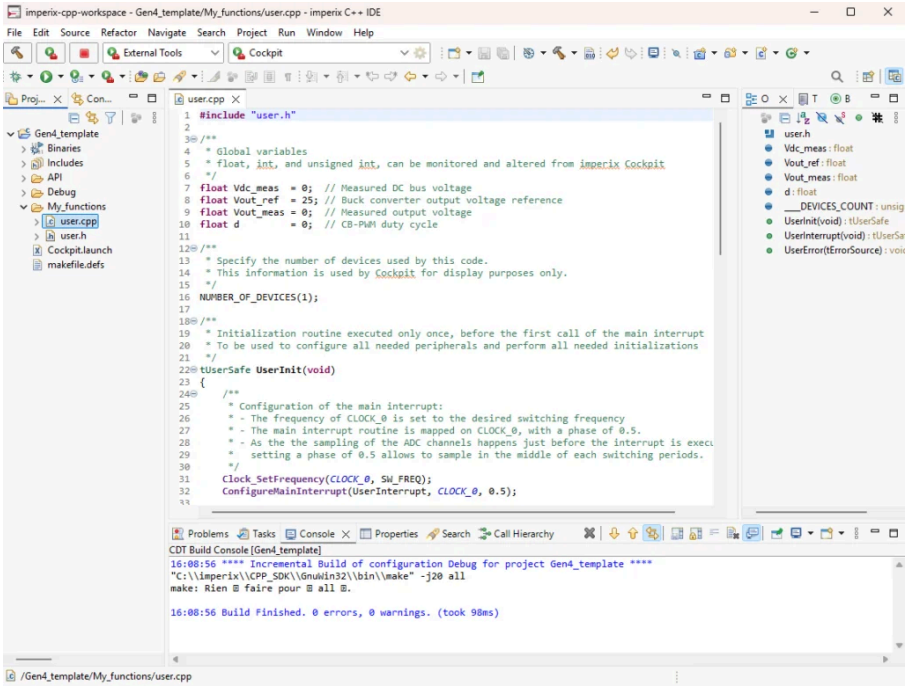
- **Product presentation:** What the CPP SDK is, and what pieces of software it contains.
- **CPP SDK Programming workflow:** A general overview of the whole workflow to give a first idea of how imperix controllers are programmed.
- **Development with the C++ IDE:** An introduction to navigating the C++ IDE and understanding the structure and content of the user code template.
- **Buck converter example:** Guidance on initializing and operating the main hardware peripherals, such as ADCs and PWM modulators, to operate a simple buck converter in open loop.

This article focuses on developing C++ code for imperix controllers. Instructions regarding the initial software installation and setup of the CPP SDK are detailed in PN146. Similarly, users interested in a block-based approach utilizing Automated Code Generation from MATLAB Simulink or PLECS should refer to the separate documentation regarding the ACG SDK.

Product description

The CPP SDK is a Software Development Kit (SDK) enabling engineers to program imperix controllers using C++ code. It includes:

- An **Eclipse-based C++ IDE** tailored to program imperix controllers

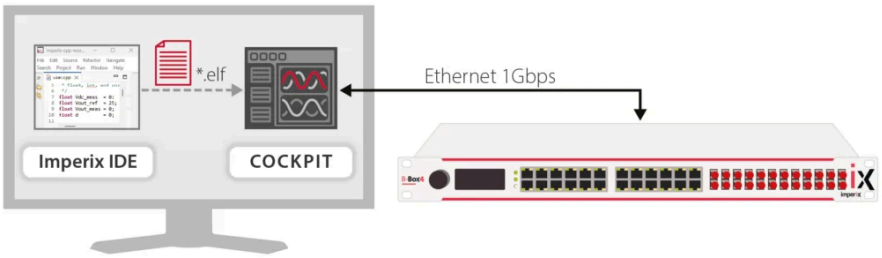


- The BBOS **operating system** for the controllers.
- The standard **FPGA firmware**.
- The **Cockpit** monitoring software. Cockpit enables real-time interaction and monitoring of the physical converter by providing access to the model variables and all the signals measured by imperix controllers.

CPP SDK Programming workflow

Before diving into control development, it is helpful to get the big picture of the CPP SDK workflow. The process bridges the gap between writing code on the PC and executing it on imperix controllers.

1. **Code development:** The control algorithm is implemented within the Imperix IDE (based on Eclipse).
2. **Compilation:** The code is built, and an executable binary file (`.elf`) is generated.
3. **Flashing the code:** This `.elf` file is retrieved by Cockpit and flashed to the controller over the Ethernet link.
4. **Execution:** The executable is loaded, and the control algorithm is run by BBOS, the controller's operating system.



Debugging note: Instead of using traditional IDE breakpoints, debugging is done by monitoring variables in real time via Cockpit.

Development with the C++ IDE

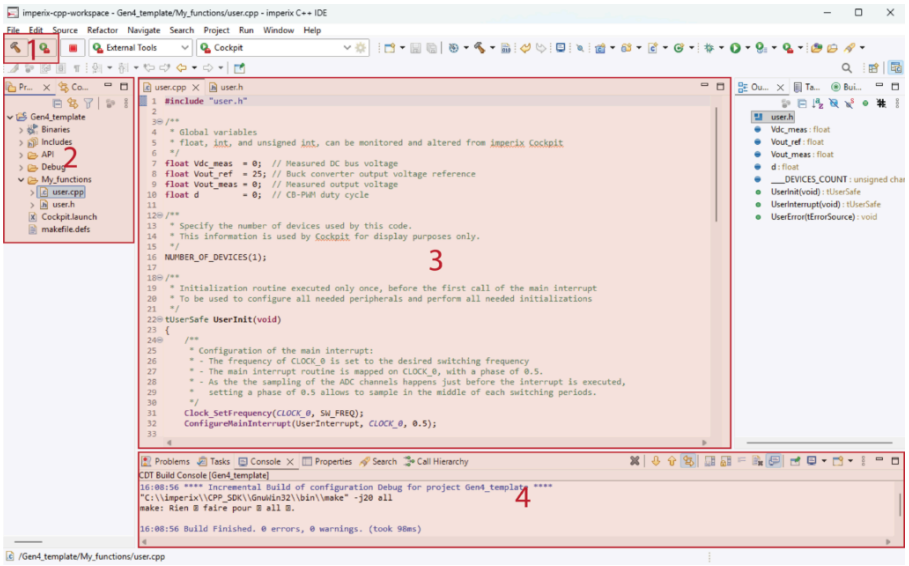
With the general workflow covered, the focus can now shift to the implementation of real-time control algorithms for the dedicated BBOS CPU core. For this, users will rely on the imperix C++ Eclipse-based IDE and the provided C++ user template.

Navigate the IDE

Assuming that the user template is already imported into the workspace, users should be presented with a window as shown in the screenshot below. Help on how to import the user template is provided in the Installation guide for imperix CPP SDK.

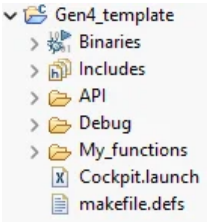
The interface is organized into several key areas as follows:

1. **Build and Launch Controls (Top Left):** Dedicated buttons located in the upper-left toolbar are used to compile the written code and deploy the resulting executable directly to the controller.
2. **Project Explorer (Left Panel):** Displays the project's directory structure and the files included in the project. The specific roles and contents of these files are detailed in the upcoming sections.
3. **Code Editor (Center Panel):** Serves as the primary workspace for viewing and modifying the C++ source code.
4. **Console (Bottom Panel):** Outputs the system and build logs, displaying any compilation warnings or errors.



User template structure and content

The user template should be imported into the IDE workspace by following the Installation guide for imperix CPP SDK. This next section will detail the content of the user template, which contains several folders (as shown in the Project Explorer in the above screenshot) that are organized as follows.



My functions

My_functions is the folder where all user files should be stored. By default, it contains only two files that the user can freely modify:

- `user.cpp`, which serves as the code root and contains the initialization routine `UserInit()` and the main interrupt service routine `UserInterrupt()`. Further details regarding these two functions are provided in the buck converter example below.
- `user.h`, which typically contains the prototypes of the user-defined routines as well as some useful readability helper definitions.

Includes – peripherals libraries

The **Includes** folder contains the header files for the user-accessible routines associated with the hardware peripherals of imperix controllers. By reviewing these headers, users can obtain a quick overview of the available routines and the necessary information regarding their implementation. This folder is structured into two main subdirectories: **Core** and **Driver**.

The **Core** folder notably includes routines managing the operational state (FAULT, BLOCKED, OPERATING) of imperix controllers.

The **driver** folder provides header files to peripheral driver routines, which are used to configure and operate the different peripherals of imperix controllers. All peripheral driver routines are further detailed in the imperix software documentation: https://imperix.com/software-documentation/

Finally, the **Includes** folder also contains a sensor header file, which provides a convenient list of definitions for the sensitivities of the various imperix current and voltage sensors.

Helper API

The **API** folder provides a library of predefined routines frequently used in power electronics control. Developers are encouraged to use these standard functions, though they can be modified as needed. Key implementations provided within this directory include:

- PI controllers,
- MPPT algorithms
- dq-frame and SOGI PLLs
- Clarke and Park transformations

CPP SDK coding example

This section will address the default content of the user files inside the **My_functions** folder of the user template. Reviewing this example will provide practical foundations for new users to get started with the CPP SDK.

These files implement a basic algorithm designed to operate a buck converter in an open-loop configuration, further detailed in the Step-down buck converter article. As previously explained, `user.h` includes the required user routine prototypes and readability helpers. Therefore, the following sections will focus on the control logic contained within `user.cpp`.

user.cpp

```
#include "user.h"
```

```
/**
 * Global variables
 * float, int, and unsigned int, can be monitored and altered from imperix Cockpit
 */
float Vdc_meas  = 0;  // Measured DC bus voltage
float Vout_ref  = 25; // Buck converter output voltage reference
float Vout_meas = 0;  // Measured output voltage
float d         = 0;  // CB-PWM duty cycle

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

/**
 * Initialization routine executed only once, before the first call of the main interrupt
 * To be used to configure all needed peripherals and perform all needed initializations
 */
tUserSafe UserInit(void)
{
        /**
         * Configuration of the main interrupt:
         * - The frequency of CLOCK_0 is set to the desired switching frequency
         * - The main interrupt routine is mapped on CLOCK_0, with a phase of 0.5.
         * - As the the sampling of the ADC channels happens just before the interrupt is executed,
         *   setting a phase of 0.5 allows to sample in the middle of each switching periods.
         */
        Clock_SetFrequency(CLOCK_0, SW_FREQ);
        ConfigureMainInterrupt(UserInterrupt, CLOCK_0, 0.5);

        /**
         * Configuration of the ADC channels, with:
         * - sensor sensitivity defined in user.h
         * - no sensor offset compensation
         */
        Adc_ConfigureSensor(ADC0, IX_PEB_800_40_V_SENSITIVITY, 0.0, 0);
        Adc_EnableSynchronousAveraging(ADC0);

        Adc_ConfigureSensor(ADC1, IX_VSR_500_HBW_SENSITIVITY, 0.0, 0);
        Adc_EnableSynchronousAveraging(ADC1);

        /**
         * Configuration of PWM channel 0 (lines 0 & 1)
         * - mapping on CLOCK_0
         * - triangle carrier
         * - 1 microsecond dead-time between complementary signals
         */
        CbPwm_ConfigureChannel(PWM_CHANNEL_0, CLOCK_0, TRIANGLE, DEADTIME);

        /**
         * Note: the function CbPwm_ConfigureChannel() is a helper defined in user.h
         * By default this routine:
         * - sets a phase of 0 between CLOCK_0 and PWM carrier
         * - sets outputs as 2 complementary signals
         * - activates the PWM channel
         */

        return SAFE;
}


/**
 * Main interrupt routine
 */
tUserSafe UserInterrupt(void)
{
        // Retrieve the measurements
        Vdc_meas  = Adc_GetValue(ADC0);
        Vout_meas = Adc_GetValue(ADC1);

        // Compute the duty cycle for the CB-PWM modulator
```

```
        d = Vout_ref / Vdc_meas;

        // Update the PWM duty-cycles
        CbPwm_SetDutyCycle(PWM_CHANNEL_0, d);

        return SAFE;
}


/**
 * Routine executed when the core state goes into FAULT mode
 */
void UserError(tErrorSource source)
{

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

Global variables

The initial segment of the user.cpp file is dedicated to the declaration of global variables. All global variables (of type int, unsigned int, or float) in `user.cpp` will be available in the monitoring software Cockpit (later introduced in the suggested further readings), as probes or tunable parameters. This enables real-time scoping and modification of these variables during converter operation.

For the buck converter in the user template, the variables below are instantiated specifically for this purpose.

```
/**
 * Global variables
 * float, int, and unsigned int, can be monitored and altered from imperix Cockpit
 */
float Vdc_meas  = 0;  // Measured DC bus voltage
float Vout_ref  = 25; // Buck converter output voltage reference
float Vout_meas = 0;  // Measured output voltage
float d         = 0;  // CB-PWM duty cycle
```

Users can prevent specific global variables from appearing in Cockpit by declaring them inside a dedicated C++ namespace, as done in the code example below.

```
namespace{float my_hidden_variable;}; // Variable that does not appear in CockpitCode language: C++ (cpp)
```

UserInit() function

The next function located within `user.cpp` is `UserInit()`. This initialization routine is executed only once, prior to the initial call of the main interrupt. It is utilized to configure the main control interrupt timings and initialize all necessary hardware peripherals, such as ADCs and PWM modulators.

`UserInit()` notably defines the control period, the sampling phase, and the switching frequency, using the four clocks at the user's disposal: `CLOCK_0`, `CLOCK_1`, `CLOCK_2` and `CLOCK_3`. Comprehensive details regarding the configuration of these clocks can be found in Timing configuration on imperix controllers.

For the specific case of the user template, the `UserInit()` function configures the following:

- Sets the frequency of `CLOCK_0` to the desired frequency (e.g. 20kHz).
- Maps the main interrupt to `CLOCK_0` and sets a sampling phase of 0.5
- Configures ADC channels 0 and 1 with the proper sensor sensitivity.
  (Warning: for B-Box RCP 3.0 the configured sensitivity should include the configured input gain on the analog front end)
- Enables synchronous averaging for both ADC channels. More information on synchronous averaging is provided in Sampling techniques for power electronics.
- Initializes a carrier-based PWM peripheral on PWM_CHANNEL_0. This modulator is mapped to CLOCK_0, is based on a triangular carrier, and is configured with a dead-time of 1 us.

```c
/**
 * Initialization routine executed only once, before the first call of the main interrupt
 * To be used to configure all needed peripherals and perform all needed initializations
 */
tUserSafe UserInit(void)
{
    /**
     * Configuration of the main interrupt:
     * - The frequency of CLOCK_0 is set to the desired switching frequency
     * - The main interrupt routine is mapped on CLOCK_0, with a phase of 0.5.
     * - As the the sampling of the ADC channels happens just before the interrupt is executed,
     *   setting a phase of 0.5 allows to sample in the middle of each switching periods.
     */
    Clock_SetFrequency(CLOCK_0, SW_FREQ);
    ConfigureMainInterrupt(UserInterrupt, CLOCK_0, 0.5);

    /**
     * Configuration of the ADC channels, with:
     * - sensor sensitivity defined in user.h
     * - no sensor offset compensation
     */
    Adc_ConfigureSensor(ADC0, IX_PEB_800_40_V_SENSITIVITY, 0.0, 0);
    Adc_EnableSynchronousAveraging(ADC0);

    Adc_ConfigureSensor(ADC1, IX_VSR_500_HBW_SENSITIVITY, 0.0, 0);
    Adc_EnableSynchronousAveraging(ADC1);

    /**
     * Configuration of PWM channel 0 (lines 0 & 1)
     * - mapping on CLOCK_0
     * - triangle carrier
     * - 1 microsecond dead-time between complementary signals
     */
    CbPwm_ConfigureChannel(PWM_CHANNEL_0, CLOCK_0, TRIANGLE, DEADTIME);

    /**
     * Note: the function CbPwm_ConfigureChannel() is a helper defined in user.h
     * By default this routine:
     * - sets a phase of 0 between CLOCK_0 and PWM carrier
     * - sets outputs as 2 complementary signals
     * - activates the PWM channel
     */

    return SAFE;
}
```

UserInterrupt() function

Finally, the `UserInterrupt()` function encapsulates the main control interrupt routine. This function is executed at `CLOCK_0`'s frequency, as configured in `UserInit()`. `UserInterrupt()` implements the user-defined control algorithms required to operate power converters. This generally includes the acquisition of ADC measurements, the execution of standard power electronics control strategies (such as PI controllers and dq-frame transformations), and the update of the duty cycle for the PWM modulators.

In the case of the user template, the `UserInterrupt()` function simply computes the required duty cycle for the buck converter according to the input voltage measurement. It then updates the output of the PWM modulator.

```c
tUserSafe UserInterrupt(void)
{
    // Retrieve the measurements
    Vdc_meas  = Adc_GetValue(ADC0);
    Vout_meas = Adc_GetValue(ADC1);

    // Compute the duty cycle for the CB-PWM modulator
    d = Vout_ref / Vdc_meas;

    // Update the PWM duty-cycles
    CbPwm_SetDutyCycle(PWM_CHANNEL_0, d);

    return SAFE;
}
```

Note that both the `UserInit()` and `UserInterrupt()` functions return the value `SAFE` when they were executed without errors. Otherwise, the B-Box will go into fault, should an error occur during the execution, or if the value `UNSAFE` is returned.

Further reading

It is highly recommended to read the following pages:

- Programming essentials for the CPP SDK provides programming insights for CPP SDK users.

- Programming and operating imperix controllers addresses how to deploy the control code onto an imperix controller.
- Cockpit user guide (PN300) gives a full guide on how to use imperix Cockpit monitoring software.

Controller-specific getting-started instructions can be found in the following notes:

- B-Box 4 – Quick start Guide
- B-Box micro – Quick start Guide
- TPI 8032 – Quick start Guide