

ICC-4C



Industrial 4-Channel Current Controller Manual

The ICC-4C controller's hardware and corresponding software, Optotune Cockpit, shall only be used in connection with the evaluation of the liquid lens product families. Any other use is not permitted without the prior written authorization of Optotune Switzerland AG.

IN NO EVENT SHALL OPTOTUNE BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS CURRENT CONTROLLER AND ITS DOCUMENTATION, EVEN IF OPTOTUNE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

OPTOTUNE SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE CURRENT CONTROLLER AND ACCOMPANYING DOCUMENTATION, IF ANY, PROVIDED HEREUNDER IS PROVIDED "AS IS". OPTOTUNE HAS NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Contents

1. Overview	5
2. Hardware operation	6
2.1. Package contents and description	6
2.2. Connecting to a computer or another host system	7
2.3. Connecting ICC-4C-500 to a lens	8
2.4. Connecting ICC-4C-2000 to FMR-20, BSW-20, and XPRs	11
2.5. Machine vision lighting control	13
2.6. Mounting	14
2.7. Thermal Management	15
2.8. Do's and Don'ts	15
3. Software operation	16
3.1. Controller related systems (widgets)	17
3.2. Lens related systems (widgets) for ICC-4C-500	20
3.3. FMR/BSW/XPR related systems (widgets) for ICC-4C-2000	24
4. Software development	27
4.1. C# SDK installation and run	27
4.2. C# code example	27
4.3. Python SDK installation	30
4.4. Python code example	31
5. Firmware	33
5.1. Firmware content description	33
5.2. Firmware update	34
6. UART Communication Protocol	35
6.1. Simple mode protocol for ICC-4C-500	35
6.2. Simple mode protocol for ICC-4C-2000	36
6.3. Pro mode protocol	37
6.4. Pro mode example	38
7. I2C Communication Protocol	42
7.1. I2C Configuration	42
7.2. I2C communication protocol description	42
7.3. I2C Write command structure	43
7.4. I2C Read command structure	43
7.5. Changing the I2C register count	45
8. Ethernet and TCP Communication	46
8.1. IP settings configuration via DHCP	47

8.2	Static IP address configuration	48
8.3	Ethernet communication example via Cockpit or a Telnet Client	50
8.4	UDP messaging	51
9.	Analog input mode.....	53
10.	Input/Output trigger signals	55
11.	Troubleshooting and FAQs.....	57
11.1.	Status LED	57
11.2.	Diagnostics in Optotune Cockpit.....	58
11.3.	Diagnostics with the device manager and Communication Terminal.....	58
11.4.	FAQs	61

1. Overview

The ICC-4C Industrial Controller enables seamless control of Optotune's liquid lenses and Fine Steering mirrors (FMR) and Beam-shifting windows (BSW) with a wide variety of control interfaces directly from a computer or from an electronics cabinet, offering a streamlined solution for rapid deployment in optical applications. This product is offered in two versions: ICC-4C-500 with $\pm 500\text{mA}$ of current range and ICC-4C-2000 with $\pm 2000\text{mA}$ of current range.

Key Features:

- **High-Precision Current Control:**
 - **ICC-4C-500:** Adjustable current range from -500 mA to +500 mA with fine 65 μA current step resolution
 - **ICC-4C-2000:** Adjustable current range from -2000 mA to +2000 mA with fine 250 μA current step resolution
- **Communication Interfaces:**
 - **USB**
 - **UART** or **I2C** with automatic protocol detection at device startup
 - **Ethernet** with POE+ capability
 - **Analog Voltage Input Control** in range from 0 to 10 V
 - **GPIO Trigger** for event-based control
- **Advanced Lens Calibration and Compensation:** Enables readout of stored calibration data and real-time temperature from supported lenses for dynamic adjustment in Focal Power Mode
- **Smart Step Feature** to enable faster focal power settling time of the lens by applying a preconditioned coil current waveform
- **User-Friendly Control Options:**
 - Compatible with the **Optotune Cockpit GUI** for USB based control
- **Various Output Connections and Debug Features:**
 - **Hirose Connector** for fast and robust industrial connection
 - **Auxiliary Connectors** to connect an Extension Board providing an option to connect a liquid lens via an FPC flex cable, while also providing various debugging options (external current measurement, triggering, analog voltage control)
- **Developer Support:** Software Development Kits (SDKs) are available for **Python** and **C#**
- **Compliance:** Fully certified with **RoHS**, **REACH** and **CE** standards

The detailed technical specifications can be found in [ICC-4C-500's datasheet](#) and [ICC-4C-2000's datasheet](#).

To access additional resources, like firmware updates, the Optotune Cockpit software or CAD files, please visit Optotune's [Download Center](#) (a free registration is required).

2. Hardware operation

2.1. Package contents and description

As mentioned earlier, ICC-4C is available in two versions of ICC-4C-500 and ICC-4C-2000. ICC-4C-500 is coming with 6pin Hirose connectors, shown in Figure 1(a), and is designed for operating Optotune liquid lenses. On the other hand, ICC-4C-2000 is available with a 34pin Output connector, as shown in Figure 1(b), and is designed for operating Optotune Beam shifting windows (BSW), Optotune Extended Pixel Resolution actuators (XPRs), Optotune Fine steering Mirror (FMR).



Figure 1. (a) ICC-4C-500. (b) ICC-4C-2000.

Description	Part number
ICC-4C-500	145-214-00
ICC-4C-2000	145-215-00

The part numbers mentioned above only include the controller unit. To run liquid lenses with FPC cable or Optotune OA products, it is necessary to use ICC-4C extension boards. As a result, an extension kit is available, both for ICC-4C-500 and ICC-4C-2000 which includes the required extension board, 200mm extension cables, Micro-USB cable, and DIN rail mount. Additionally, 24C power adapters are also available for different countries:

Description	Part number
ICC-4C-500 Extension KIT	148-355-00
ICC-4C-2000 Extension KIT	148-356-00
ICC-4C Power Adapter + EU mains cable	146-789-01
ICC-4C Power Adapter + CH mains cable	146-789-02
ICC-4C Power Adapter + US mains cable	146-789-03
ICC-4C Power Adapter + UK mains cable	146-789-04
ICC-4C Power Adapter + CN mains cable	146-789-05

2.2. Connecting to a computer or another host system

ICC-4C provides the following connectivity options:

A Micro **USB-B connector**

An **RJ-45 Ethernet connector** with PoE+ capability (802.3af or higher)

The **Auxiliary I/O connector** can provide additional communication and power supply options

To use the ICC-4C with **UART** communication, connect the *UART TX* (pin 7), *UART RX* (pin 9) and *GND* (pins 11 and 17) to the system that is controlling the device.

For **I2C** communication, connect with the *I2C SCL* signal being on pin 15, the *I2C SDA* on pin 13 and the *GND* (pins 11 and 17).

This connector can also provide an option of an using an **External power supply**. To use this feature, connect the *External VCC* (pin 19) and *External GND* (pin 20) to the supply rail of the external device. To enable external power supply, a dummy barrel plug needs to be inserted.

The ICC-4C-500 provides an option to map an **Analog Input Voltage** to a specific lens currents or focal power. On the other hand, ICC-4C-2000 provides such an Analog Input Voltage mapping to be used with XPR, BSW, and FMR. As a result, ICC-4C enables controlling an Optotune product by a value from an external sensor or other analog voltage. This feature is accessible by connecting the external analog signal to the *AnalogIn* pins and *AGND* (pins 6) of the auxiliary I/O connector. For each current channel, AnalogIn pins are pin 3, pin 4, pin1, and pin 2, respectively.

The controller also has two configurable **GPIO** pins (pin 6 and 8), which can be configured to generate a trigger signal synchronized with the internal signal generator or to wait for an external trigger signal before changing the current or focal power of the lens. Please note, that these pins were meant to be used with 3.3V logic and as such, the related min/max voltages and threshold levels shall be respected.

Synchronization pins can be used for vector pattern units. Sync A (pin 10) for vector pattern and Sync A and Sync B (pin 8) for the quadrant vector pattern. In this case, validity of signal level sequence is checked by a state machine with specific patterns of these signals. The logic level for all of them is 3.3V. More information about quadrant vector pattern can be found in the firmware documentation.

Driver reset signal is a link directly to the MCU. By grounding this pin, the MCU can be restarted and will not be responding until few seconds. During this period, the driver will be disappeared from the OS device manager, and all communication stops.

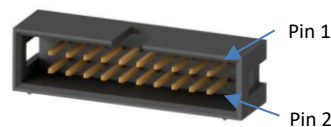


Figure 2: Auxiliary I/O connector pinout. Wurth Electronics header 100mil 90deg

Pin	Signal	Value	Function
1	AnalogIn2	0..10 V	Analog input signal #2
2	AnalogIn3	0..10 V	Analog input signal #3
3	AnalogIn0	0..10 V	Analog input signal #0
4	AnalogIn1	0..10 V	Analog input signal #1

5	nRESET	0..3.3V (CMOS level)	Driver reset signal (inverted level)
6	AGND	-	Analog signal ground
7	USART_EXT_TX	0..3.3V (CMOS level)	UART transmit line
8	SYNC_B	0..3.3V (CMOS level)	Synchronization input B
9	USART_EXT_RX	0..3.3V (CMOS level)	UART receive line
10	SYNC_A	0..3.3V (CMOS level)	Synchronization input A
11	GND	-	Digital ground
12	GPIO3_SPI_MOSI	0..3.3V (CMOS level)	General purpose digital IO #3 or SPI MOSI signal (Trigger OUT/IN)
13	I2C_HOST_SDA	0..3.3V (CMOS level)	I2C data line from host
14	GPIO2_SPI_MISO	0..3.3V (CMOS level)	General purpose digital IO #2 or SPI MISO signal (Trigger OUT/IN)
15	I2C_HOST_SCL	0..3.3V (CMOS level)	I2C clock line from host
16	GPIO1_SPI_NSS	0..3.3V (CMOS level)	General purpose digital IO #1 or SPI SlaveSelect signal (Trigger OUT/IN)
17	GND	-	Digital ground
18	GPIO0_SPI_SCK	0..3.3V (CMOS level)	General purpose digital IO #0 or SCK signal (Trigger OUT/IN)
19	VCC_EXT_IN ¹	24...48 V	External DC supply voltage positive input
20	GND_EXT_IN	-	External DC supply voltage negative input

2.3. Connecting ICC-4C-500 to a lens

The ICC-4C-500 can be connected to a lens by using one of the following two options:

By using an **industrial (Hirose) connector** the lens can be simply and safely connected by a Hirose cable to the plug on the front panel of the ICC-4C, until a “click” sound is heard. The advantage of this connector is that the positions of the pins are unambiguous, so it is safe to connect it even by hot plugging during the operation of the controller.

¹ To enable external power supply, a dummy barrel plug needs to be inserted.



Figure 3: Connecting to a lens with Hirose connector

With an **FPC flex cable and the ICC-4C-500 Extension Board**, by connecting the extension board to the Auxiliary output connector on the front panel of the ICC-4C-500 and then connecting the lens to the FPC connector on the extension board. This option provides the ability for the users to do some measurements via the extension board, but they should be more careful, as the FPC cable is not suitable for hot plugging (it can be inserted under an angle or even upside down), and an incorrect connection can cause damage to the lens and/or the controller. **To use the FPC cable connection safely, the power supply always should be removed from the controller first, before manipulating with the flex cable.** There are 4 DIP switches on the extension board corresponding to each current channel. With the default setting of DIP switches, i.e. all switches in OFF position, allows to have automatic device detection. DIP switches with ON state is not recommended.

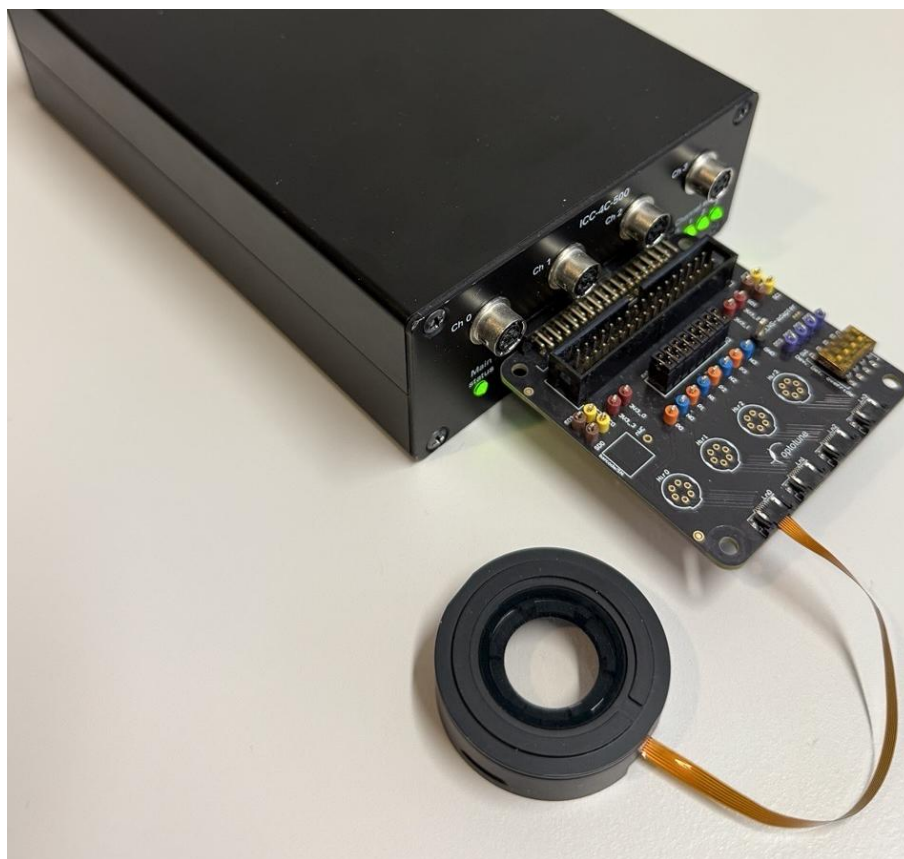


Figure 4: Connecting to a lens with FPC connector

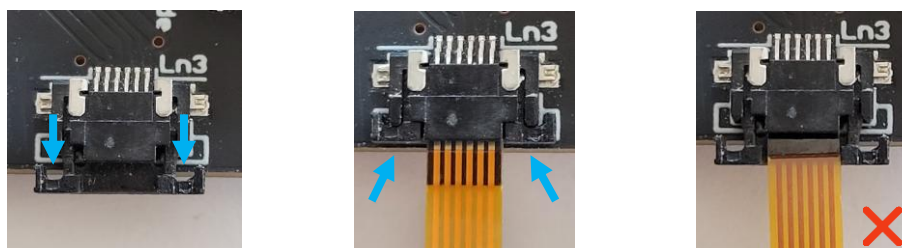


Figure 5: Proper connection to a lens with a flex cable and a wrong connection example with flipped flex cable and not enclosed connector

WARNING: Do NOT hot plug the flex cable from the adaptor board, it might damage the lens and the controller!

2.4. Connecting ICC-4C-2000 to FMR-20, BSW-20, and XPRs

The ICC-4C-2000 can be used with Optotune FMR-20, Optotune BSW-20, and Optotune XPR-20 or XPR-33, using **ICC-4C-2000 Extension board**. All of these products can be connected to ICC-4C-2000 in the same way. A picture of such a connection is presented in Figure 6. Following components are needed:

- FMR-20 or BSW-20 (incl. FFC cable)
- ICC-4C-2000 controller
- ICC-4C-2000 extension kit
 - Extension board for FMR-20 / BSW-20
 - Extension cable (15 cm, optional)
 - DIN rail clamp kit (included, not shown)
 - The FMR-20 / BSW-20 is connected via adapter board and standard FFC cable (12 pin, 0.5mm pitch). A power supply and a USB cable are also included
- ICC-4C power adapter and power cable (not shown, available plug types: EU, CH, US, UK, CN)

WARNING: Do not connect or disconnect the FMR-20 device with the FFC cable when powered. EEPROM corruption or permanent electronics damage may result.

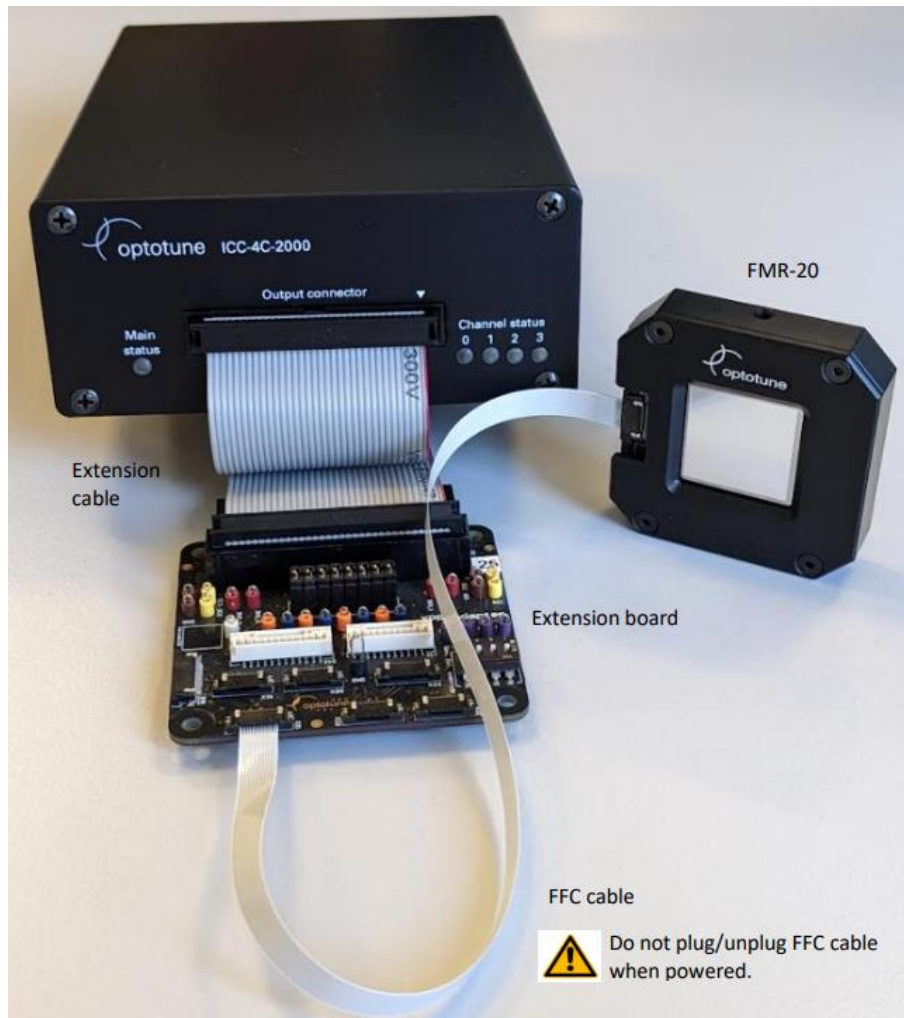


Figure 6: Picture of FMR-20 connected to the ICC-4C-2000 controller.

The FFC cable can be plugged directly into the connector of the adaptor board. Open the connector clamp with tweezers. Insert the flex cable fully with the copper pads facing upwards. Close the black clamp by pushing both sides simultaneously. **IMPORTANT:** Unlock the connector before making a new connection:

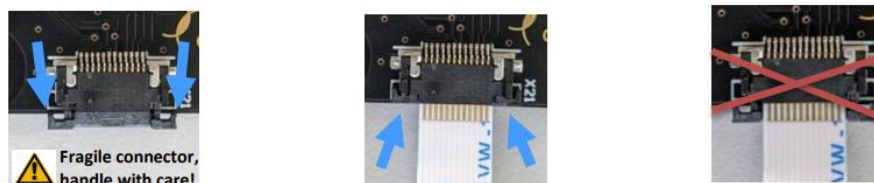


Figure 7: Proper connection to a BSW-20/FMR-20 with an FFC cable and a wrong connection with not enclosed connector

Use connector X21 on the extension board to connect the device. A second device can be connected on connector X22:

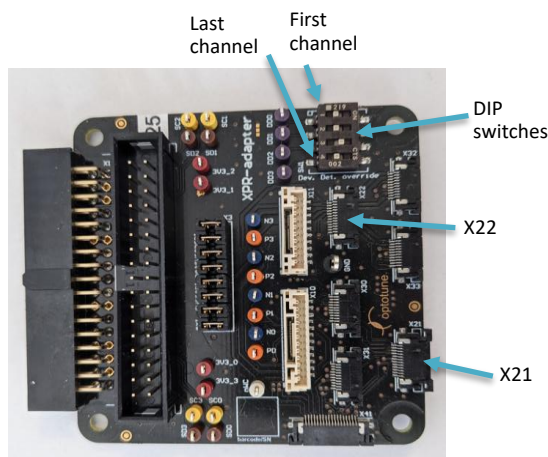


Figure 8: Connectors on ICC-4C-2000 extension board.

The default setting of DIP switches, i.e. all switches in OFF position, shall be used with the FMR-20. Using a different setting will disable automatic device detection, which is not recommended.

2.5. Machine vision lighting control

For many machine vision applications, lighting control is an essential part of the overall vision system, which typically includes the camera, liquid lens, and LEDs. The output current channels of the ICC-4C-500 can be used as a light controller within such systems. This integrated solution is particularly beneficial when both liquid lens control and lighting control are required simultaneously, as it simplifies system design and helps reduce cost and development time. The Optotune Cockpit software, along with the ICC-4C SDKs and interfaces (such as I2C), enables seamless synchronization between the lighting, liquid lens, and camera.

The ICC-4C-500 can drive multiple small LEDs with a maximum output current of 500mA and a maximum voltage of 18 V. Short current pulses are not supported. Optotune provides the CAB-6-100-M-OE cable, which is a Hirose to open-ended cable, as shown in Figure 9. This cable connects the Hirose connector on the ICC-4C-500 to the LED. The current pins (pin 1 and pin 2, black and red) should be used to connect to the LED.



Figure 9: CAB-6-100-M-OE cable and for example 12V LED

As examples, following LEDs can be supported by ICC-4C-500:

- Direct rectangular SMD illumination: LBRX-00-040-3-W-12V (12V input, white)
- Ring light: LLA-60-070-2-W-12V (12V input, white)

2.6. Mounting

The ICC-4C is available in a metal casing that can be mounted on a standard DIN rail (a suitable adapter is already included in the Extension KIT package). For more mounting details and dimensions, see the mechanical drawing below:

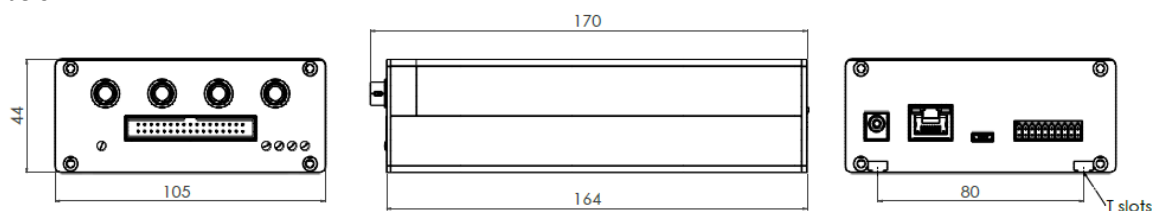


Figure 10: ICC-4C-500 casing

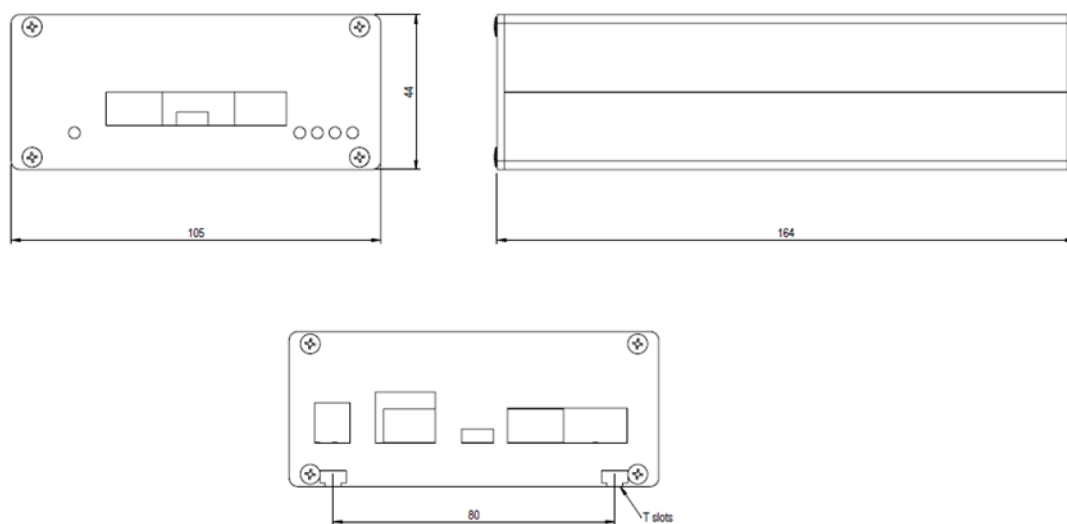


Figure 11: ICC-4C-2000 casing

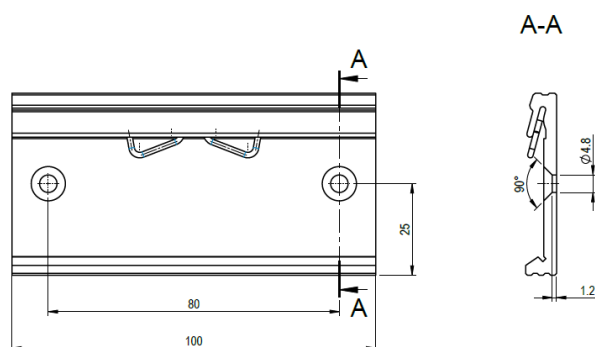


Figure 12: DIN rail adapter for ICC-4C

2.7. Thermal Management

The ICC-4CC has a built-in thermal monitoring for both the connected lens and the controller itself, to prevent overheating and potential damage. The temperature limits for both the controller and the connected lens can be programmed in the Optotune Cockpit, or by a custom software or device sending the appropriate command to the controller.

During the placement and mounting of the controller it should be ensured that it's not placed right next to a significant heat source, so that the ambient temperature around the controller will remain in the range defined in the datasheet (max. 65 °C). Temperature limits for the internal temperature sensors are defined in the firmware. For the power stage temperature sensor, the limit is 95 °C on the ICC-4C-500 and 115 °C on the ICC-4C-2000. For the output stage temperature sensor, the limit is 85 °C on the ICC-4C-500 and 100 °C on the ICC-4C-2000.

2.8. Do's and Don'ts



When using the Extension Board, do not connect or disconnect the FPC flex cable, while the device is powered on. During these manipulations the adjacent pins on the cable can be temporarily shorted and if the device is powered on at the same time, this may lead to EEPROM data corruption and/or permanent damage to the controller or the lens.



This product must be powered by a certified external SELV power supply with CE marking, compliant with Directive 2014/35/EU (Low Voltage Directive).



This controller and its extensions (for example the Extension Board) can be damaged by ESD. We recommend the device to be handled with appropriate precautions and protections. Failure to observe proper handling and installation procedures might cause damage. ESD damage can range from subtle performance degradation to complete device failure.



Do not use aggressive agents such as acetone or acids to clean the ICC-4C, the Extension Board or any of the other components included with the device.

3. Software operation

The **Optotune Cockpit GUI** allows users to establish connections with the controller and the connected lens, monitor device statuses, and perform firmware updates. The menu on the left side contains 3 main sections:

The **Electronics** section is showing the type and the serial number of the connected controllers. Clicking on the “+” button initiates a search for any compatible controller on the available interfaces.

The **Products** section is showing the type and the serial number of the lens connected to the selected controller.

The **Systems** section contains a list of available features for the given controller and lens, and each of them opens a standalone widget in the Dashboard section (right side) containing all the setting related to that feature.

The application supports multiple customizable **Dashboards**, allowing users to rearrange and resize widgets to suit their preferences and workflow.

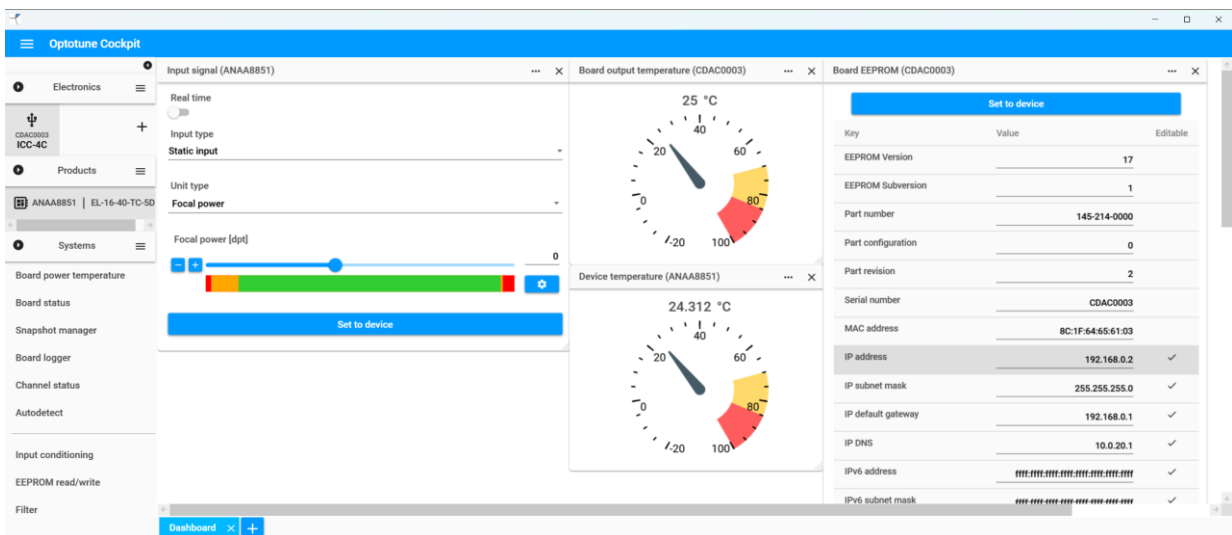


Figure 13: Optotune Cockpit main window overview

The available systems/widgets in the software can be divided into two groups based on whether they contain setting related to the controller or the connected product. We discuss all the controller related widgets in section 3.1 for both ICC-4C-500 and ICC-4C-2000, and then we have two separate sections for ICC-4C-500 and ICC-4C-2000 to discuss product related widgets.

3.1. Controller related systems (widgets)

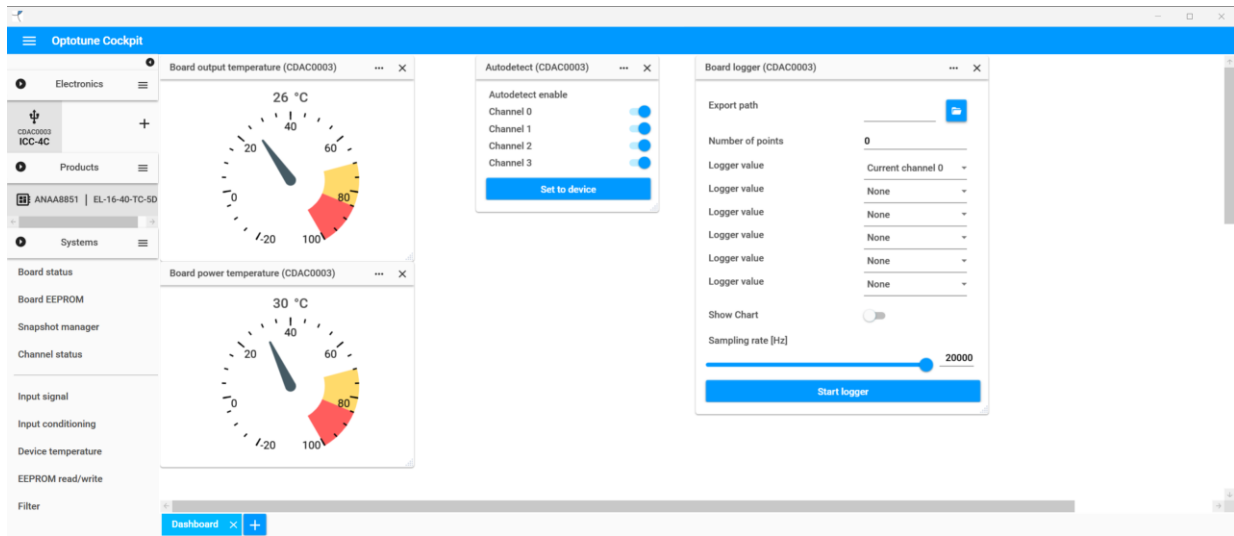


Figure 14: Examples of widgets related to the controller

The widgets related to the controller allow the user to access various settings and monitoring features of the ICC-4C, such as:

Temperature readouts for various subsystems (the input power supply, the output current stage and the MCU) and settings for the thermal shutdown temperatures for these subsystems.

Board status, Channel status and Autodetect features to monitor and control values related to the driver, such as the Frontend voltage, which can improve the accuracy of the current control for different sized lens or the Autodetect feature, which allows the user to turn on/off the automatic detection of the lens connected to the controller.

Board EEPROM widget, which contains the information stored in the ICC-4Cs memory, such as the device's serial number or the settings required for Ethernet and I2C communication.

Board logger widget, which allows monitoring and exporting values like output current, focal power and even register values into a .csv file.

Snapshot manager widget, which can store two complete sets of settings for both the controller and the lens. Slot number 1 is read-only and contains the factory defaults, while slot number 2 can be configured and saved by the user.

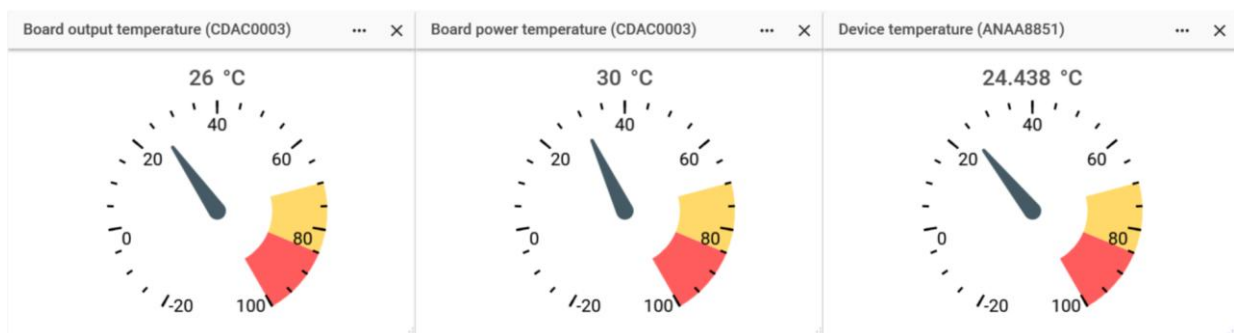


Figure 15: Temperature readout widgets

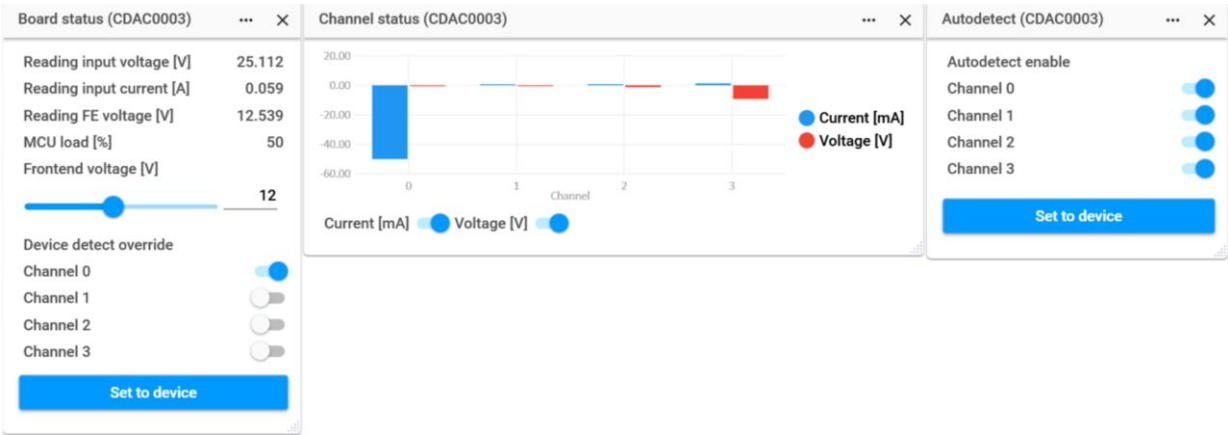


Figure 16: Board status, Channel status and Autodetect widgets

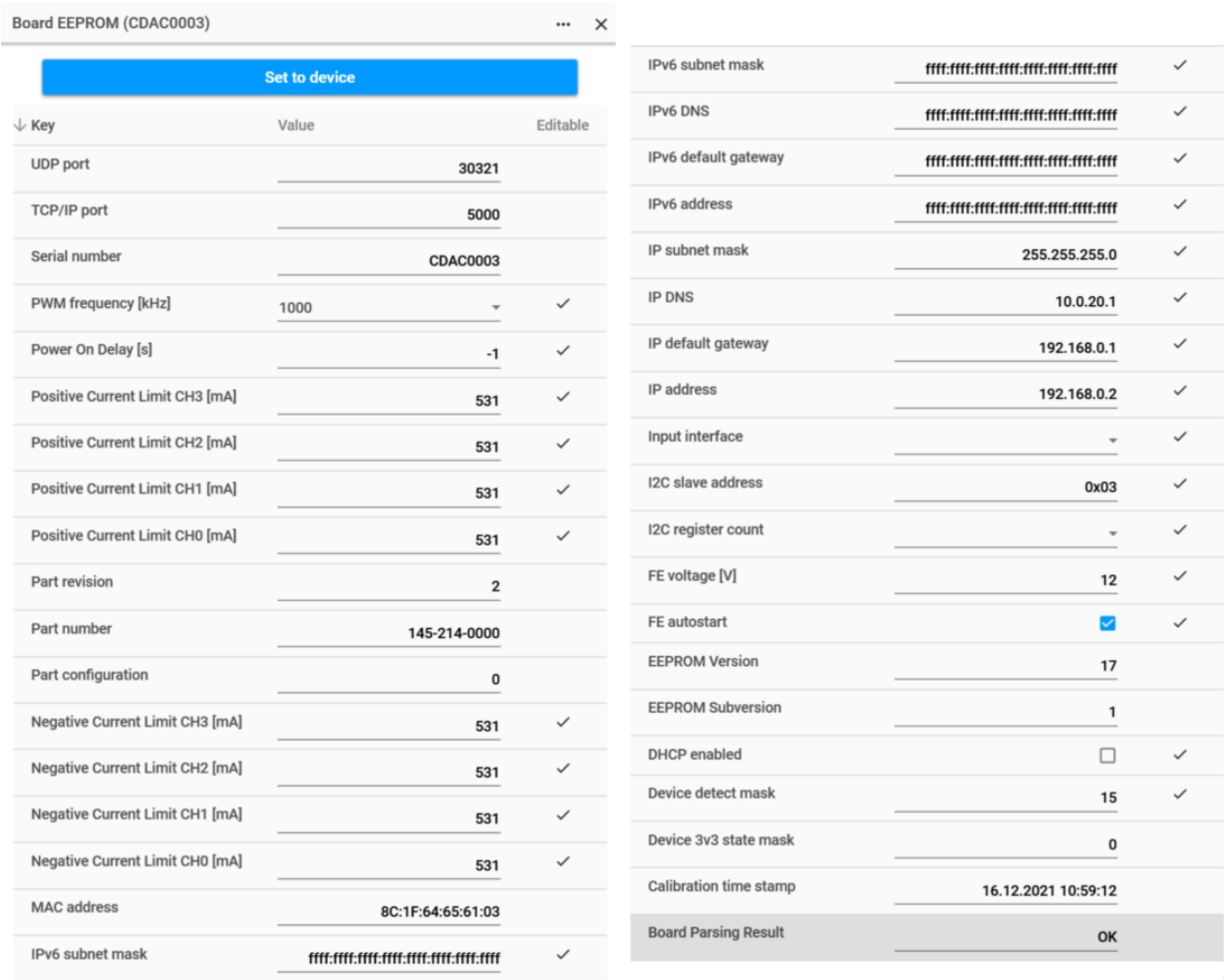


Figure 17: Board EEPROM widget

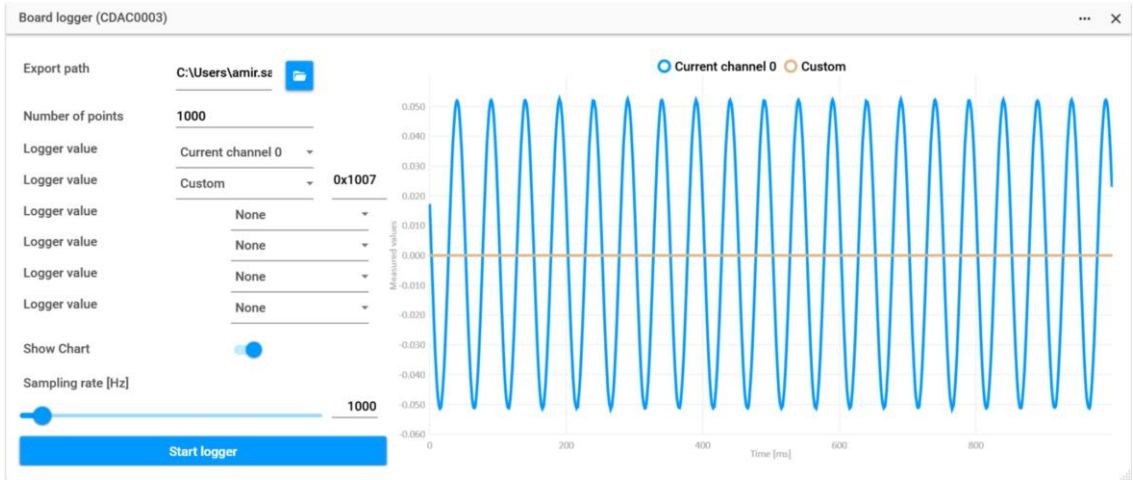


Figure 18: Board logger widget



Figure 19: Snapshot manager widget

3.2. Lens related systems (widgets) for ICC-4C-500

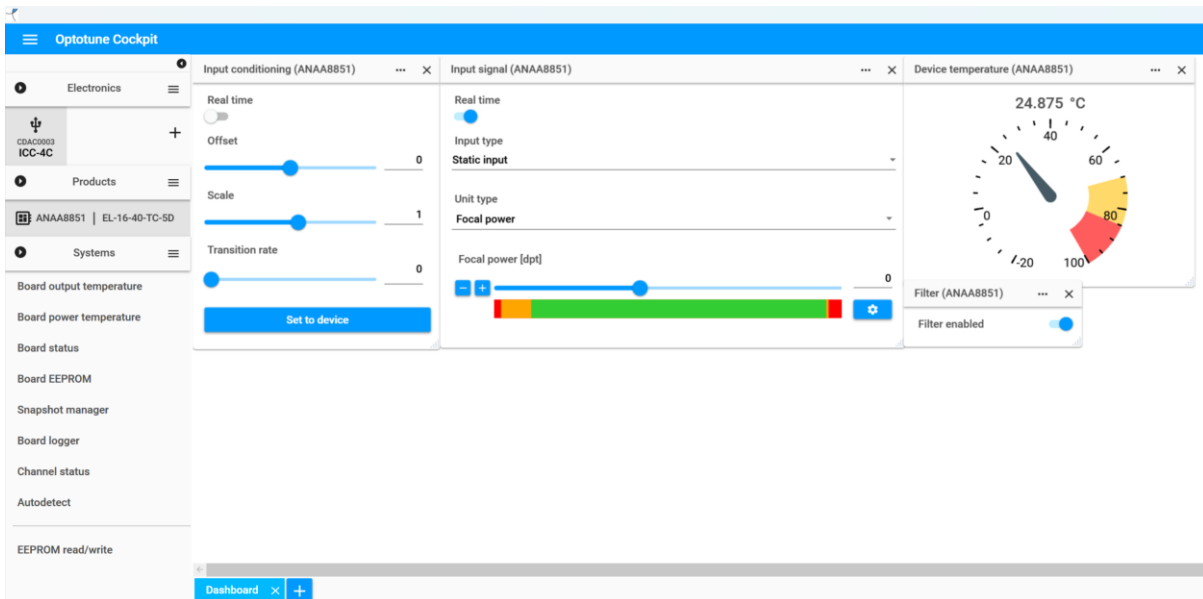


Figure 20: Examples of widgets related to the lens

The widgets related to the lens allow the user to access settings and monitoring features for the various lenses, such as:

Temperature readout and settings for the thermal shutdown temperature. This feature is only available for lenses with a built-in temperature sensor.

Smart Step Filter feature to achieve better settling times during transitions.

EEPROM read/write widget, which allows to read out the serial number, calibration data and other information from the connected lens. It can also allow restoring the EEPROM data in case of issues or data corruption. This feature is only available for lenses with a built-in EEPROM chip.

Input conditioning widget to apply an offset, scaling or transition speed limit to any input value defined in the Input signal widget.

Input signal widget, which allows the user to select the controlled value (current or focal power) as well as multiple options on how the given value should be controlled:

Static input value to maintain a certain output current or focal power value.

A **Signal generator**, where the user can set the output current or focal power to follow a predefined waveform shape from the controller's built-in signal generator (Sinusoidal, Triangular, Square, Sawtooth, Pulse, Staircase or Fast autofocus waveforms are available).

Custom vector, where the user can define multiple output current or focal power values with their respective timings for the lens to go through.

Analog input, where the user can map analog input voltage values (in the range of 0 – 10V) provided via the *Analog In* pin to a specific output currents or focal powers.

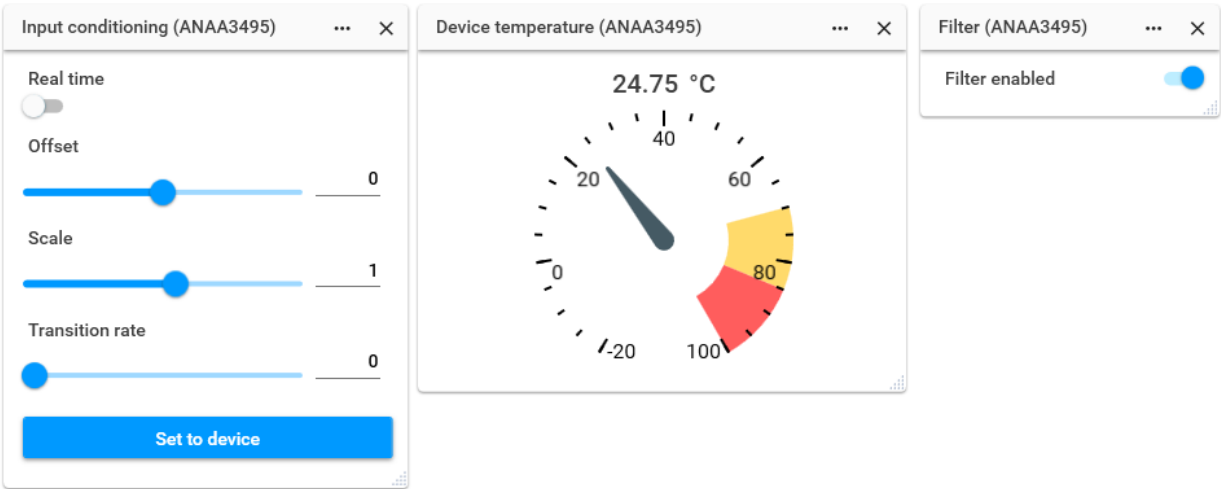


Figure 21: Input conditioning, Lens temperature readout and Smart Step Filter settings widgets

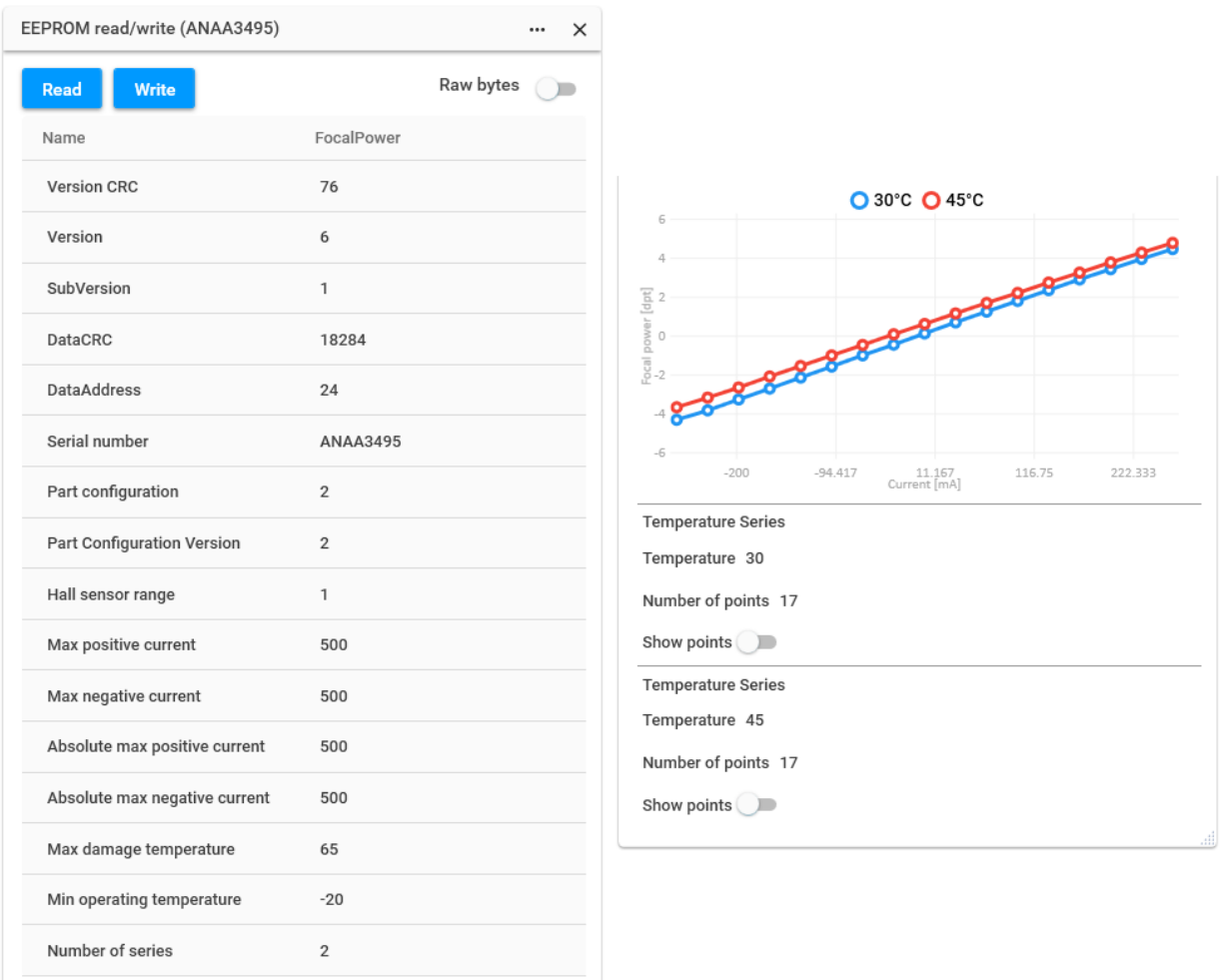


Figure 22: Lens EEPROM read/write widget

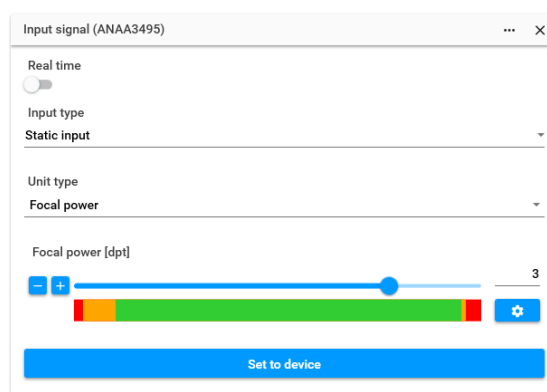


Figure 23: Input signal widget set to Static input

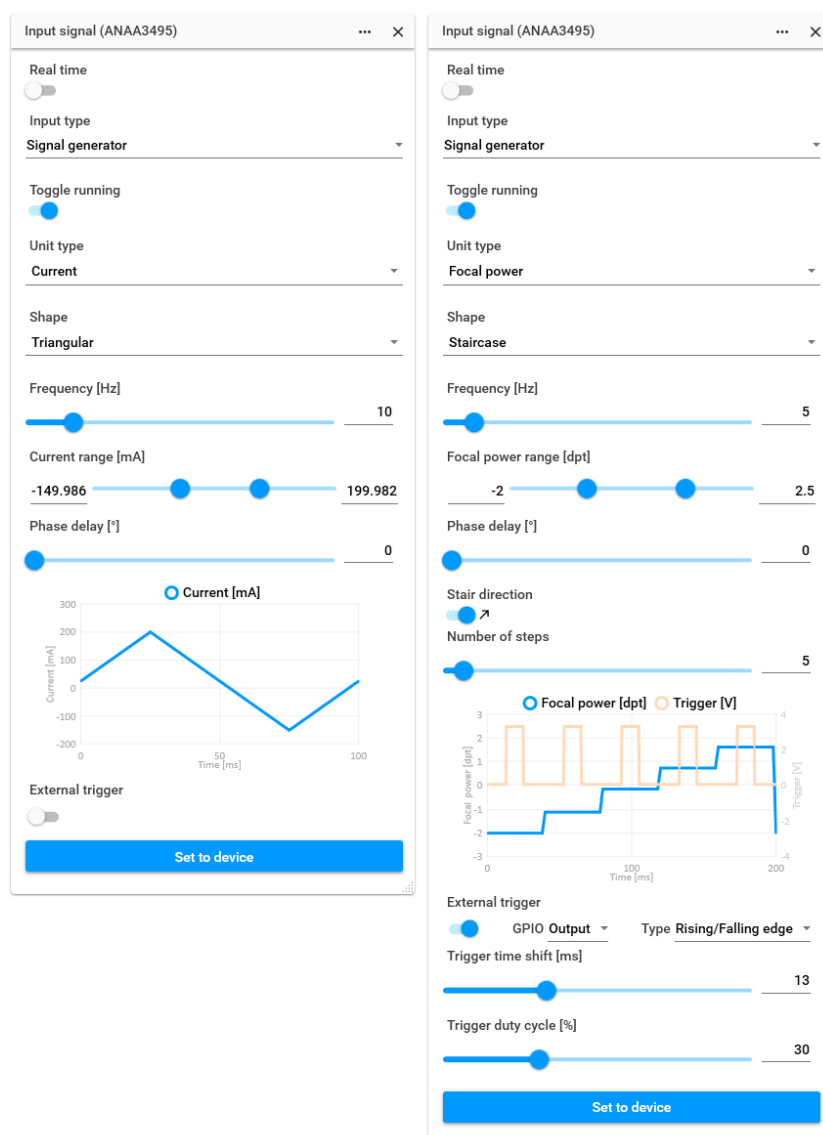


Figure 24: Input signal widget set to various waveforms from the Signal generator

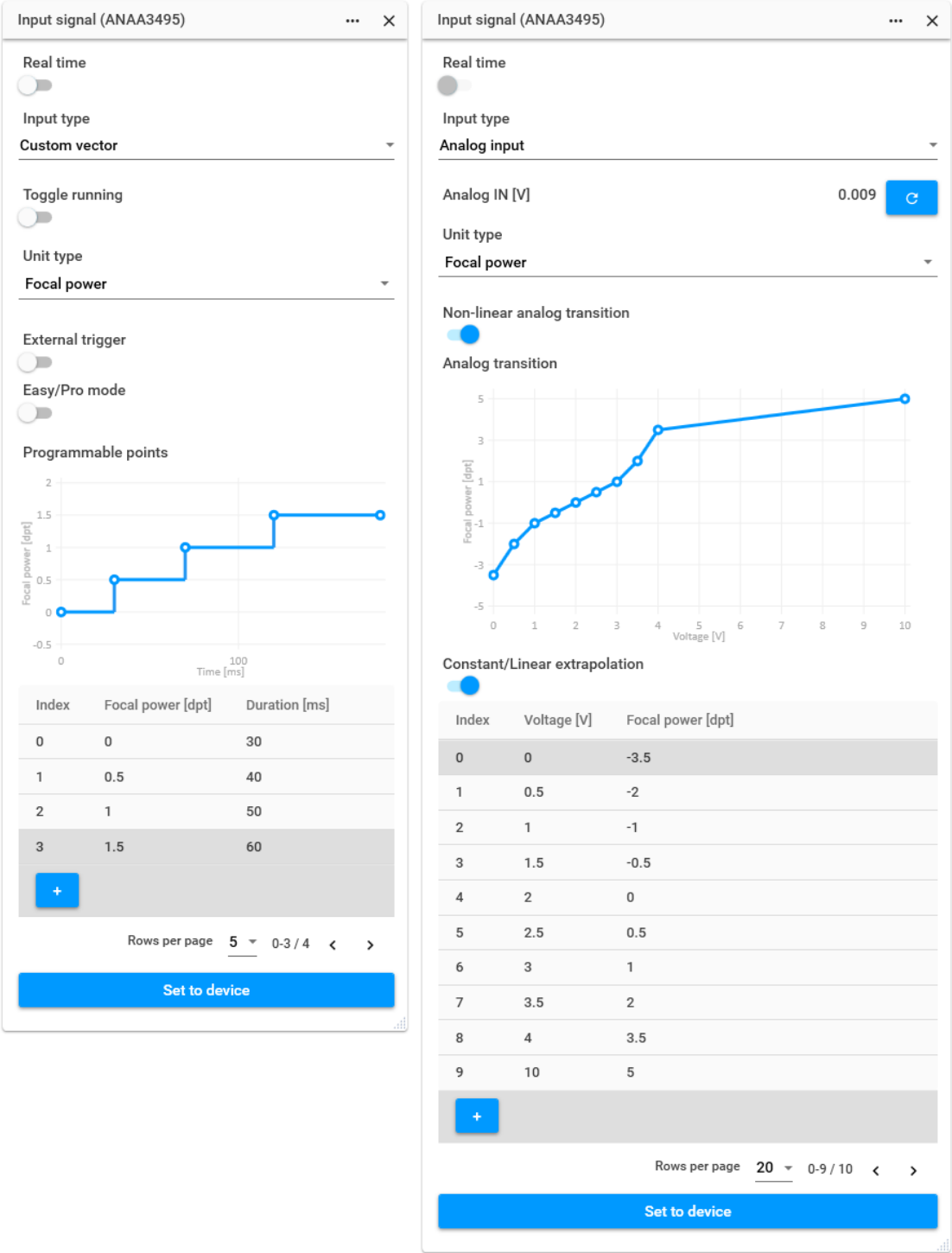
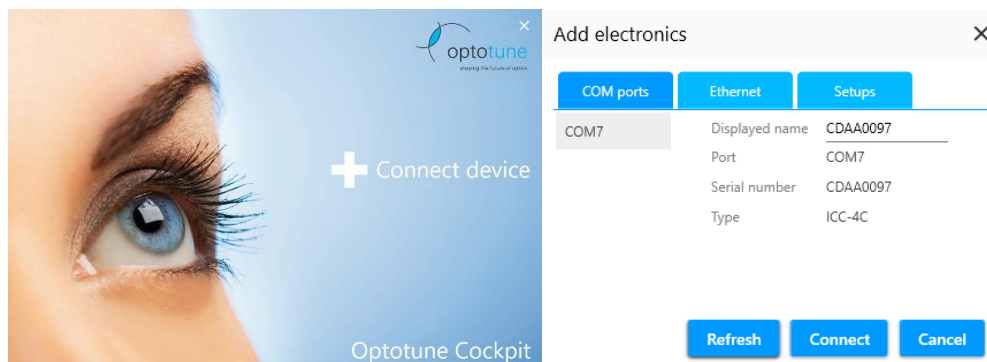


Figure 25: Input signal widget set to Custom vector and Analog input

3.3. FMR/BSW/XPR related systems (widgets) for ICC-4C-2000

Make sure that the ICC-4C is connected via USB to the computer and powered. Alternatively, connecting via Ethernet (incl. PoE+) is possible. Open Optotune Cockpit and click on *Connect device*.



A window will appear, allowing you to select the controller. The COM port is detected automatically. After clicking connect, the device will appear on the left-hand side of the Optotune Cockpit window.

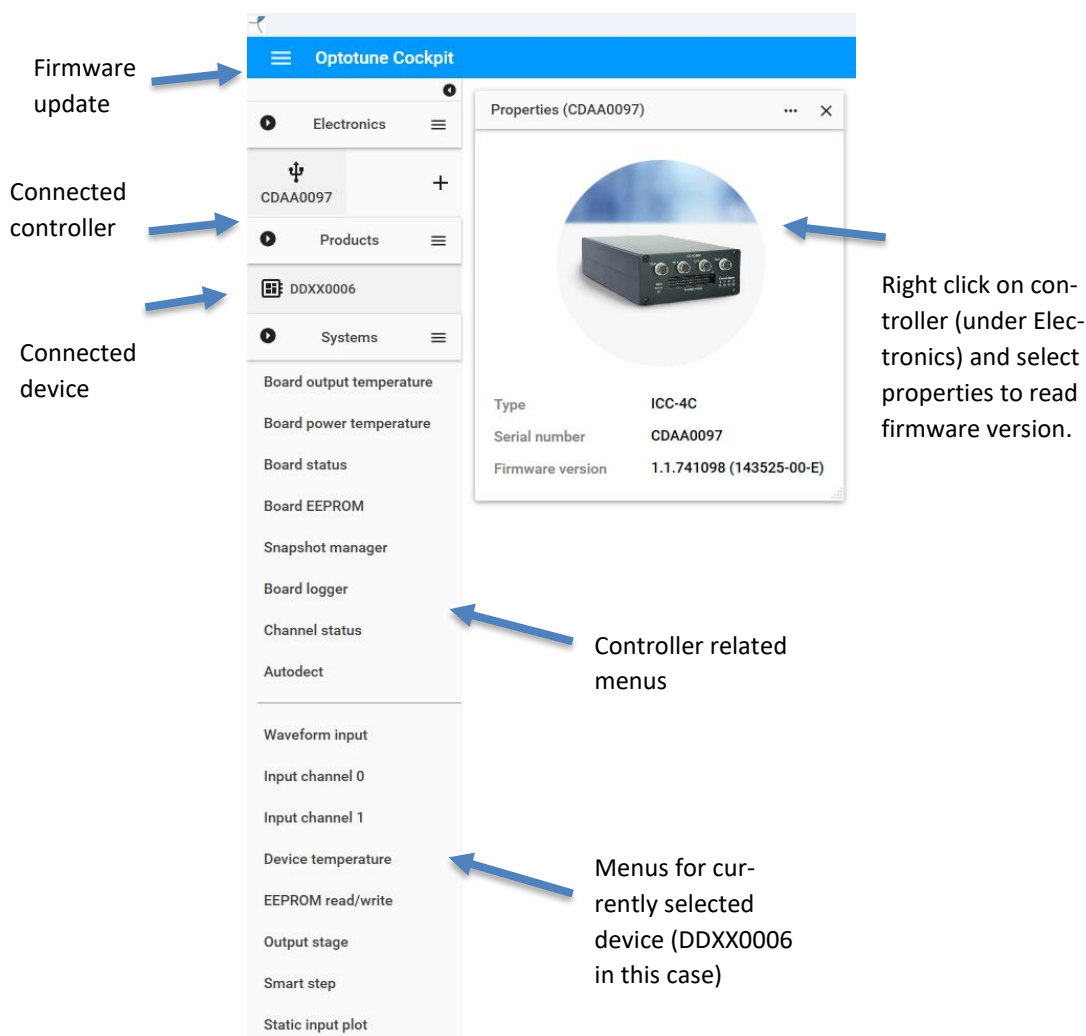


Figure 26. Widgets for ICC-4C-2000 in Optotune Cockpit

To control the device, the main panels used for operating the FMR are the *Static Input Plot* and the more general *Input Channel 0* and *Input Channel 1* menus.

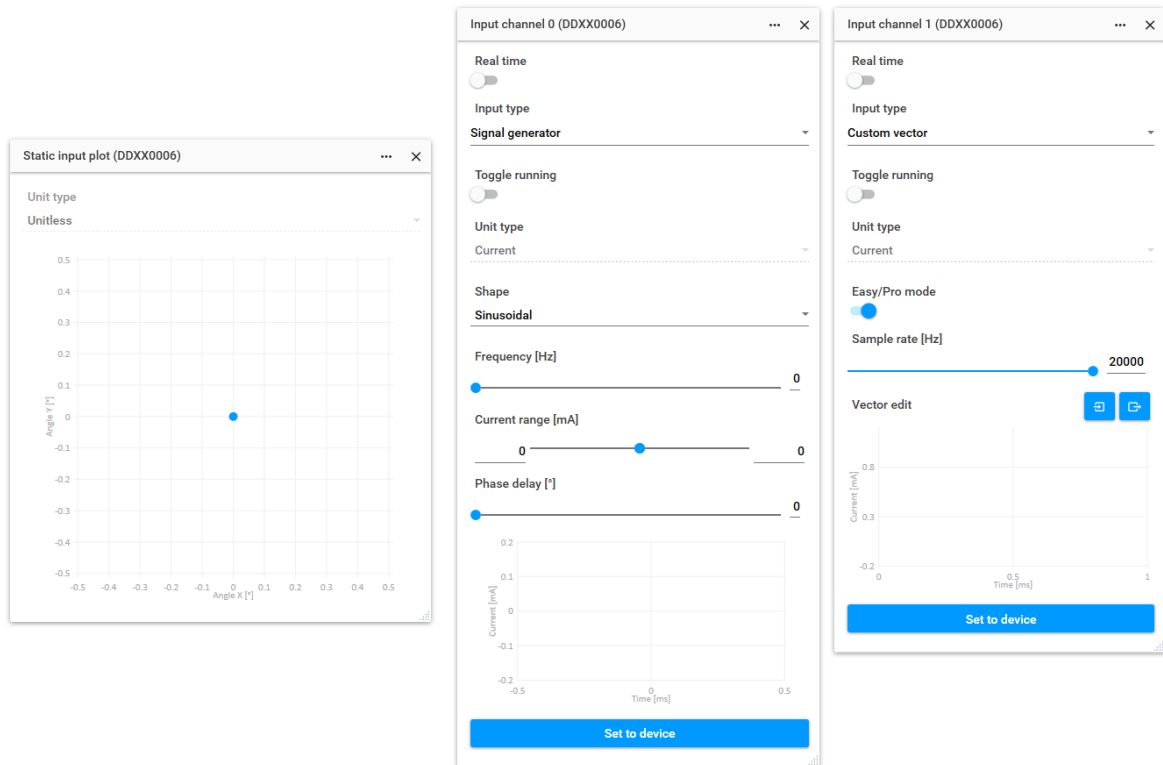


Figure 27. Input signal settings for ICC-4C-2000 for BSW/FMR/XPR

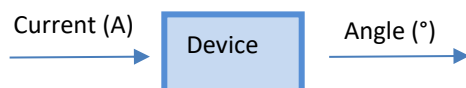
The *Static input plot* is a graphical representation of the pointing direction of the mirror. It can be clicked to make the mirror move to a desired position.

The *Input channel 0* and *Input channel 1* menus provide access to the following device control modes:

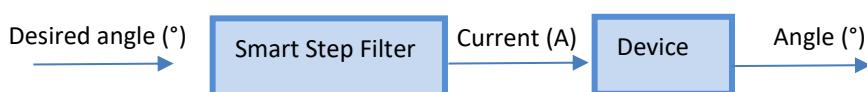
- **Static value:** Optotune Cockpit sends a static current value to the driver.
- **Signal generator:** The internal signal generator within the driver sends a time-dependent stream of position or current values until it is interrupted. Available signal generating functions are Sinusoidal, Rectangular, Sawtooth, Triangular and Staircase. These signals stay in sync over different channels, independently of the starting time, i.e. when you press *Set to device*.
- **Arbitrary vector:** The driver reads a set of user-defined positions or currents one after another and sends them to the mirror. Two modes are possible, simple mode with user defined sequence of points directly in the GUI. Import of a csv-table describing the sequence of values to be scanned.
- **Analog input:** The driver switches to Analog input mode.

There are two ways to control the mirror through this Panel, either with *Real Time Input* on or off. If *Real Time Input* is on, any changes in the panel are transferred directly to the driver and take effect immediately. If it is turned off, you need to press *Set to device* after making changes for them to be sent to the driver.

By default, the Unit type is *Current*. Due to the linear response of the device, the angular position control is straightforward for quasistatic applications.



For more dynamic waveforms, the dynamics of the device (FMR / BSW) need to be considered. The ICC-4C provides a *Smart Step* functionality for this purpose:



This filter uses a model that is based on the calibration parameters that are specific to every device:

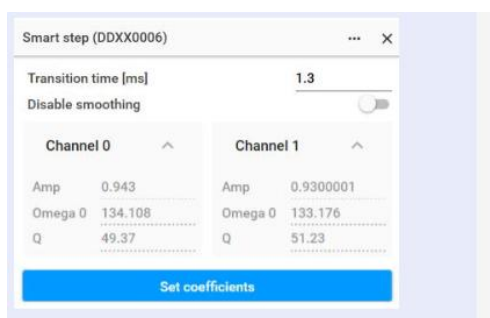


Figure 28. Smart Step settings for ICC-4C-2000 in Optotune Cockpit

The transition time of 1.3 ms should only be changed, if speed is crucial. Reducing the value can increase ringing and increase power consumption.

With *Smart Step* on, all input systems will automatically change to angle units (°) instead of current (A).

4. Software development

There are two different software development kits available for the ICC-4C – one for C# and one for Python. Both are available on the [Download Center](#) website (a free registration is required). They contain the necessary libraries to work with the ICC-4C as well as some example codes.

4.1. C# SDK installation and run

To install and run the C# SDK examples, the following steps should be performed:

1. Download the ICC-4C/ECC-1C/ICC-1C C# SDK from Optotune's website.
2. Extract the .zip file.
3. This C# SDK requires to have Visual Studio and the .NET framework 4.6.1/4.6.2 installed, so please make sure you have them on your PC.
4. Open the *Icc4cExample.sln* solution.
5. Build all and Run the appropriate project.
6. To create a new solution, in the downloaded C# SDK there are available .dll files, which could be added into any project's references and directly control Optotune's products.

Additional information about the C# SDK can be found in the detailed SDK documentation (included in the *\html* subfolder of the downloaded SDK – open the *index.html* file).

4.2. C# code example

```
2. using System;
3. using System.Threading;
4. using Icc4cExample;
5. using ICC4cPwmSdk.Device;
6. using ICC4cPwmSdk.Dtos;
7.
8. namespace Icc1cExample
9. {
10.     internal class Icc1cExample
11.     {
12.         // To run this example, rename AnaInMain to Main and in Example.cs rename Main to
13.         ExMain for example
14.         // static void Main(string[] args) -> is always an entry point
15.         static void Main(string[] args)
16.         {
17.             Console.WriteLine("Starting basic ICC-4C example...");
18.             Icc1cSdkDeviceController controller;
19.
20.             while (true)
21.             {
22.                 Console.WriteLine("Select connection: C = COM, E = ETHERNET");
23.                 var key = Console.ReadKey();
24.                 Console.WriteLine();
25.                 //COM connection
26.                 if (key.Key == ConsoleKey.C)
27.                 {
28.                     controller = new Icc1cSdkDeviceController();
29.                     Console.WriteLine("What is the COM number of the ICC-4c?");
30.                     var comport = Console.ReadLine();
31.                     if (int.TryParse(comport, out var comInt))
32.                     {
33.                         if (controller.Connect(comInt))
```

```
34.         {
35.             break;
36.         }
37.
38.         Console.WriteLine("Connection failed.");
39.     }
40. }
41.
42. // Ethernet connection
43. if (key.Key == ConsoleKey.E)
44. {
45.
46.     controller = new Icc1cSdkDeviceController(true);
47.     Console.WriteLine("Static IP or automatic search? A = auto, S = static");
48.     key = Console.ReadKey();
49.     Console.WriteLine();
50.
51.     //Automatic ethernet search (DHCP has to be enabled in the board EEPROM
52.     //default))
53.     if (key.Key == ConsoleKey.A)
54.     {
55.         var devices = EthernetSearcher.SearchEthernetIccBoards();
56.         if (devices.Count == 0)
57.         {
58.             Console.WriteLine("No ethernet boards found.");
59.         }
60.         else
61.         {
62.             Console.WriteLine($"Boards found({devices.Count}):");
63.
64.             var index = 0;
65.             foreach (var device in devices)
66.             {
67.                 Console.WriteLine($"Index: {index}, Serial number: {device.SerialNumber}, IP: {device.Ip}, Port: {device.Port}");
68.                 index++;
69.             }
70.             Console.WriteLine("Select index of the device you want to connect.");
71.             var selectedIndex = Console.ReadLine();
72.             if (int.TryParse(selectedIndex, out var indexInt) && indexInt >=
73.                 0 && indexInt < devices.Count)
74.             {
75.                 if (controller.Connect(devices[indexInt].Ip, devices[indexInt].Port))
76.                 {
77.                     break;
78.                 }
79.                 Console.WriteLine("Connection failed.");
80.             }
81.         }
82.     }
83.
84.     //Static IP connection (static IP must be configured in the board EEPROM
85.     //using Optotune Cockpit application), also DHCP has to be switched in
86.     //the board EEPROM)
87.     if (key.Key == ConsoleKey.S)
88.     {
89.         Console.WriteLine("Specify IP address of the board:");
90.         var ipAddress = Console.ReadLine();
91.
92.         if (controller.Connect(ipAddress, 5000))
93.         {
94.             break;
95.         }
96.     }
97. }
```

```
93.             Console.WriteLine("Connection failed.");
94.         }
95.     }
96. }
97.
98. //Set board to PRO mode
99. controller.ChangeModeToPro();
100.
101. //Getting board info
102. Console.WriteLine("Board info");
103. var serialNumber = controller.GetElectronicSerialNumber();
104. var fwVersion =
105.     $"{controller.Status.GetBoardFirmwareMajorVersion()}.{controller.Status.Get-
BoardFirmwareMinorVersion()}.{controller.Status.GetBoardFirmwareRevisionVersion()}";
106.
107. Console.WriteLine($"Board serial number: {serialNumber}");
108. Console.WriteLine($"Board firmware version: {fwVersion}");
109.
110. // In order to drive the connected product, device type of the product has to be
obtained (this is not needed if
111. // automatic detection is allowed)
112. var connectedDevice = controller.GetDeviceType();
113. Console.WriteLine();
114. Console.WriteLine(
115.     $"Connected device: {string.Join(Enum.GetName(typeof(EConnectedDeviceType),
connectedDevice))}");
116. Console.WriteLine();
117.
118. var lensSerialNumber = controller.GetElectronicSerialNumber();
119. Console.WriteLine(
120.     $"Lens {Enum.GetName(typeof(EConnectedDeviceType), connectedDevice)}
({lensSerialNumber}) found");
121.
122. var lensTemperature = controller.TemperatureManager.GetDeviceTemperature();
123.
124. Console.WriteLine($"Lens temperature: {lensTemperature}°C");
125.
126. var minCurrent = controller.DeviceEeprom.GetMaxNegativeCurrent();
127. var maxCurrent = controller.DeviceEeprom.GetMaxPositiveCurrent();
128.
129. Console.WriteLine($"Minimum current is {minCurrent} mA, maximum current is {max-
Current} mA");
130.
131. Console.WriteLine();
132. Console.WriteLine("Setting static current");
133. //Setting input system to static input
134. controller.InputStage.ChangeActiveSystem(EInputSignalStageSystem.StaticInput);
135.
136. for (var current = minCurrent; current <= maxCurrent; current += (maxCurrent -
minCurrent) / 5)
137. {
138.     //Value has to be converted from mA to A
139.     var currentInA = current / 1000;
140.     Console.WriteLine($"Current {currentInA} A");
141.     controller.InputStage.StaticValue.SetCurrent(currentInA);
142.     //Dioptr can be set similarly
143.     //controller.InputStage.StaticValue.SetDioptr(lensChannel, dioptrInDpt);
144.     Thread.Sleep(1000);
145. }
146.
147. Console.WriteLine("Setting static current to 0 A");
148. controller.InputStage.StaticValue.SetCurrent(0);
149.
150. Console.WriteLine();
151. Console.WriteLine("Running signal generator");
152.
```

```
153.         controller.InputStage.ChangeActiveSystem(EInputSignalStageSystem.SignalGenera-
154. tor);
155.         controller.InputStage.SignalGenerator.SetUnitType(EICC4cPwmUnitType.Current);
156.         controller.InputStage.SignalGenerator.SetShape(ESignalGeneratorShape.Sinusoidal);
157.         controller.InputStage.SignalGenerator.SetAmplitude(0.2f);
158.         controller.InputStage.SignalGenerator.SetFrequency(5);
159.         controller.InputStage.SignalGenerator.SwitchRunning(true);
160.
161.         for (var i = 0; i < 5; i++)
162.         {
163.             Console.WriteLine(".");
164.             Thread.Sleep(1000);
165.         }
166.         controller.InputStage.SignalGenerator.SwitchRunning(false);
167.
168.         Console.WriteLine();
169.         Console.WriteLine("Signal generator stopped");
170.
171.         Console.WriteLine();
172.         Console.WriteLine("Device EEPROM");
173.
174.         var eepromVersion =
175.             $"{controller.DeviceEeprom.GetEepromVersion()}.{controller.De-
176. viceEeprom.GetEepromSubVersion()}";
177.         Console.WriteLine($"EEPROM version: {eepromVersion}");
178.
179.         var eepromBytes = controller.DeviceEeprom.GetBytes(0, 10);
180.         var eepromSize = controller.DeviceEeprom.GetDeviceEepromSize();
181.
182.         Console.WriteLine(
183.             $"Printing {eepromBytes.Length}/{eepromSize} bytes saved in EEPROM:
184. {string.Join(", ", eepromBytes)}");
185.
186.         Console.WriteLine();
187.         Console.WriteLine("Example finished.");
188.         Console.ReadLine();
189.     }
190. }
```

4.3. Python SDK installation

To install the Python SDK for the ICC-4CC, the following steps should be performed:

1. Download appropriate Python SDK from Optotune's website.
2. Extract the .zip file.
3. The Python SDK is installed through *pip*, so make sure you have it installed (usually it's already preinstalled in most Python environments).
4. Open the Command-line interface on your PC or your Python environment and run the following two commands (administrator privileges might be required):
 - a. `py -m pip install Download_location\ICC-4C_PythonSDK_version\optoKummenberg-version-py3-none-any.whl`
 - b. `py -m pip install Download_location\ICC-4C_PythonSDK_version\optoICC-version-py3-none-any.whl`

5. If the installation was successful, the following folders should show up in the `\Lib\site-packages` subfolder of your Python environment:

- optoICC
- optoICC-2.0.4922.dist-info
- optoKummenberg
- optoKummenberg-1.0.4894.dist-info

* `Download_location` and `version` – replace these parts in the commands with the file location and version of the download SDK.

* `py -m` – might not be required, if the command is run from certain Python environments.

Additional information about the Python SDK can be found in the detailed SDK documentation (included in the `\docs` subfolder of the downloaded SDK – open the `index.html` file).

4.4. Python code example

```
1. from time import sleep
2. from optoKummenberg import UnitType
3. from optoICC import connect, DeviceModel, WaveformType
4.
5. print("Stating basic optoICC example...")
6.
7. #Connecting to board. Port can be specified like connect(port='COM12')
8. icc4c = connect()
9.
10. print()
11. print("Board info")
12. #Getting board info
13. serial_number = icc4c.BoardEEPROM.getSerialNumber()[0].decode('UTF-8')
14. fw_version = f"{icc4c.Status.GetFirmwareVersionMajor()[0]}.{icc4c.Status.GetFirmwareVersionMinor()[0]}.{icc4c.Status.GetFirmwareVersionRevision()[0]}"
15.
16. print(f"Board serial number: {serial_number}")
17. print(f"Board firmware version: {fw_version}")
18.
19. # In order to drive the connected product, device type of the product has to be obtained (this is not needed if
20. # automatic detection is allowed)
21. connected_devices = [DeviceModel(icc4c.MiscFeatures.GetDeviceType(0)), Device-
22.                       Model(icc4c.MiscFeatures.GetDeviceType(1)),
23.                       DeviceModel(icc4c.MiscFeatures.GetDeviceType(2)), Device-
24.                       Model(icc4c.MiscFeatures.GetDeviceType(3))]
25. print(f"Connected devices: {''.join('{},'.format(x.name) for x in connected_devices)}")
26. print()
27. print("Driving lens")
28. first_connected_lens = None
29. for idx, device in enumerate(connected_devices):
30.     if device.value in [DeviceModel.EL_1030_TC, DeviceModel.EL_1030_TC, DeviceModel.EL_1230_TC,
31.                         DeviceModel.EL_1640_TC]:
32.         first_connected_lens = (device, idx)
33.         break
34.
35. if first_connected_lens is None:
36.     print("No lens connected")
37. else:
38.     # Each channel can also be accessed directly as icc4c.Channel_0 instead of icc4c.chan-
39.     nel[0]
```

```
36.     icc4c_lens_channel = icc4c.channel[first_connected_lens[1]]
37.     lens_serial_number = icc4c_lens_channel.DeviceEEPROM.getSerialNumber()[0].decode('UTF-8')
38.     print(f"Lens {first_connected_lens[0].name} ({lens_serial_number}) found on channel
{first_connected_lens[1]}")
39.
40.     lens_temperature = icc4c_lens_channel.TemperatureManager.GetDeviceTemperature()[0]
41.     print(f"Lens temperature: {lens_temperature}°C")
42.
43.     min_lens_current = icc4c_lens_channel.DeviceEEPROM.GetMaxNegCurrent()[0]
44.     max_lens_current = icc4c_lens_channel.DeviceEEPROM.GetMaxPosCurrent()[0]
45.     print(f"Minimum current is {min_lens_current} mA, maximum current is {max_lens_current}
mA")
46.
47.     print()
48.     print(f"Setting static current")
49.     #Setting input system to static input
50.     icc4c_lens_channel.StaticInput.SetAsInput()
51.
52. for current in range(int(min_lens_current), int(max_lens_current)+1, int((max_lens_current-
min_lens_current)/5)):
53.     #Value has to be converted from mA to A
54.     current_in_A = float(current)/1000
55.     print(f"Current {current_in_A} A")
56.     icc4c_lens_channel.StaticInput.SetCurrent(current_in_A)
57.     sleep(1)
58.
59.     print("Setting static current to 0 A")
60.     icc4c_lens_channel.StaticInput.SetCurrent(0.0)
61.     print(" ")
62.     print("Running signal generator")
63.     icc4c_lens_channel.SignalGenerator.SetAsInput()
64.     icc4c_lens_channel.SignalGenerator.SetUnit(UnitType.CURRENT)
65.     icc4c_lens_channel.SignalGenerator.SetShape(WaveformType.SINE.value)
66.     icc4c_lens_channel.SignalGenerator.SetAmplitude(0.2)
67.     icc4c_lens_channel.SignalGenerator.SetFrequency(5)
68.     icc4c_lens_channel.SignalGenerator.Run()
69.
70.     for index in range(5):
71.         print(".", end="")
72.         sleep(1)
73.
74.     icc4c_lens_channel.SignalGenerator.Stop()
75.
76.     print()
77.     print("Signal generator stopped")
78.
79.     print()
80.     print("Device EEPROM")
81.
82. eeeprom_version = f"{icc4c_lens_channel.DeviceEEPROM.GetEEPROMversion()[0]}.{icc4c_lens_chan-
nel.DeviceEEPROM.GetEEPROMsubversion()[0]}"
83.     print(f"EEPROM version: {eeeprom_version}")
84.
85.     eeeprom_bytes = icc4c_lens_channel.DeviceEEPROM.GetEEPROM(0, 10)
86.     eeeprom_size = icc4c_lens_channel.DeviceEEPROM.GetEEPROMSize()[0]
87.     print(f"Printing {len(eeeprom_bytes)}/{eeeprom_size} bytes saved in EEPROM:
{''.join('{:02x}'.format(x) for x in eeeprom_bytes)}")
88.
89.     print()
90.     print("Example finished.")
91.
```

5. Firmware

The ICC-4C firmware architecture is organized into multiple Systems, which are modular components of the firmware that collectively provide the device's functionality. Each system may expose a different set of registers and/or vectors, which are user-accessible variables designed for communication and control.

Available systems:

System Name	System ID
STATUS	0x10
BOARD EEPROM	0x20
DEVICE EEPROM	0x21
TEMPERATURE MANAGER	0x22
LOGGER	0x24
MISCELLANEOUS FEATURES	0x25
VECTOR PATTERN MEMORY	0x26
SNAPSHOT MANAGER	0x27
VECTOR ADAPTER	0x28
XPR CONTROL SYSTEM	0x2E
I2C	0x31
SIGNAL FLOW MANAGER	0x40
STATIC INPUT BASE	0x50
ANALOG INPUT BASE	0x58
SIGNAL GENERATOR BASE	0x60
VECTOR PATTERN UNIT BASE	0x68
QVECTOR PATTERN UNIT BASE	0x80
INPUT CONDITIONING BASE	0x98
INPUT CONDITIONING FILTER BASE	0x9C
OPEN LOOP BASE	0xB0
FEEDBACK PID BASE	0xB8
LENS COMPENSATIONS STAGE SYSTEM BASE	0xC8
OUTPUT CONDITIONING BASE	0xD0
OUTPUT CONDITIONING FILTER BASE	0xD8
LINEAR OUTPUT BASE	0xE8

Static Input

Static Input system base ID: **0x50**

The static input stage system belongs to the input stage systems of the [Signal Flow](#) and is the one which is by factory default activated at start up. With this system the user can set constant values as input to the signal flow. The register which is propagated depends on the active control stage configured by the [Signal Flow Manager](#).

Register Map:

Address	Name	Default Value	Description	Type	Access
0x5[0-7]00	Current static input	0.0	Static input in amperes. Range depends on driver type	float	read write
0x5[0-7]01	OF static input	0.0	Static input in OF. Range differs for each mirror	float	read write
0x5[0-7]02	XY static input	0.0	Static input in XY coordinates. Range is from -1.0 to 1.0	float	read write
0x5[0-7]03	Active static input type	0	Determined by active control stage system	Unit Type	read only
0x5[0-7]04	FP static input	0.0	Static input in DPT. For lenses only.	float	read write
0x5[0-7]05	Unitless value	0.0	Unitless value. Can be used for non specific value type.	float	read write

Figure 29: Available systems in the Firmware architecture of ICC-4C

5.1. Firmware content description

The ICC-4C's firmware uses the following types of registers and vectors for communication with the host PC or system:

- **Registers:**
 - **Definition:** Each register is a 32-bit value used for reading and/or writing data from/to the controller.
 - **Data Transfer:** The register data is always transmitted in big-endian format (the most significant byte is transmitted first).
 - **Data Types:** The specific meaning of the 32-bit value is system-dependent and documented in the system's firmware documentation. The common formats include:
 - IEEE 754 32-bit float.
 - Unsigned integers.
 - Boolean values.
 - **Access Types:**
 - **Read-only:** Used for monitoring or retrieving data from the controller.
 - **Write-only or Read/Write:** Used to trigger some operations or update device states.
- **Vectors:**

- **Definition:** Vectors are variable-length equivalents of registers, typically used to manage arrays of data.
- **Structure:**
 - Vectors have defined size and data type, as documented for each system.
 - Byte-addressable in most cases, though some systems may require specific alignment (e.g. 4-byte chunks).
- **Indexing:** When working with vectors, they require an index to specify the portion of the vector being accessed during the read/write operations.
- **Documentation:**
 - More comprehensive details about the supported registers and vectors, their formats, and uses can be found in the ICC-4C Firmware Documentation. The firmware (.hex file) and its related documentation can be downloaded from Optotune's [Download Center](#), after completing the required registration. The documentation must be first unzipped to show the correct formatting.

5.2. Firmware update

The ICC-4C's firmware can be updated via the Optotune Cockpit software using the USB connection. **Updating the firmware via I2C or Ethernet is not supported.** To perform a firmware update of the controller, the following steps should be performed:

1. Open the menu located in the top-left corner of the Optotune Cockpit interface.
2. Select the "Firmware Update" option. A new widget will appear on the current dashboard for managing the firmware update process.
3. Click on the folder icon within the widget to browse for the appropriate .hex file.
4. After selecting the .hex file, click on the "Update" button to begin the firmware installation process.

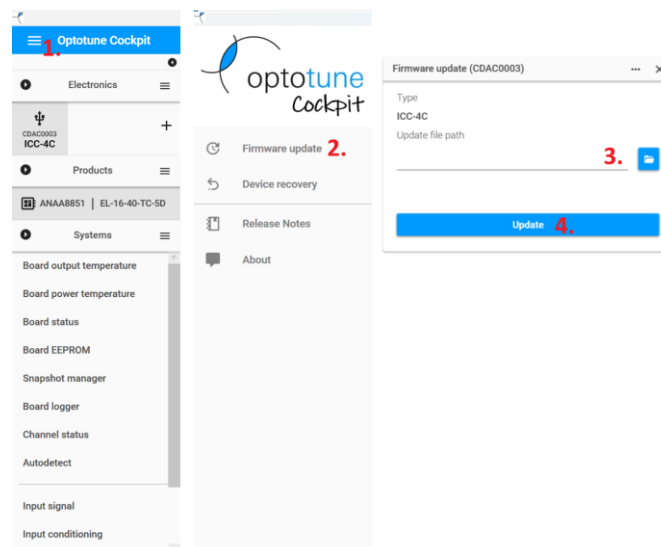


Figure 30. Firmware update for ICC-4C using Optotune Cockpit

5. Once the update is completed, please double-check the firmware version to ensure the update was successfully applied (right-click on the icon of the ICC-4C under the "Electronics" and select "Properties").

6. UART Communication Protocol

This chapter describes the protocols supported by the ICC-4Cs UART interface. While it is generally advised to use either of the SDKs presented in chapter **Error! Reference source not found.**, the following enables to program ICC-4C on platforms not covered by the Python and C# languages.

The serial communication in the system operates in one of two distinct modes: Simple Mode (ASCII Mode) and Pro Mode (Binary Mode). Simple mode protocol is easier to implement but does not cover all functionalities of the controller. We discuss the communication protocols for ICC-4C-500 and ICC-4C-2000 in separate subsections.

6.1. Simple mode protocol for ICC-4C-500

This mode is active by default upon a system startup. It provides a basic command set (**Error! Reference source not found.**) based on **RS-232** serial communication for straightforward interactions and quick setup. It's ideal for scenarios where minimal communication complexity is required. It allows interaction with the ICC-4C via commonly available serial terminals like **Termite** or **Putty**.

Interface Details:

- **Baud Rate:** 256000 (Default)
- **Data Bits:** 8.
- **Stop Bits:** 1.
- **Parity:** None (N).
- **Baud Rate Auto-Detection:** The device can adapt to baud rate changes during operation, to do so the "START" command must be issued. Regarding the UART auto baud rate feature, the limitation is that first byte which is sent must have LSB=1, otherwise the peripheral picks up wrong baud rate.

Command Structure:

- Communication is **ASCII-based**, using predefined command and response formats.
- Commands and responses are terminated with a **CR, LF** sequence (i.e., 0x0D, 0x0A in hexadecimal).
- **Case Insensitivity:** Commands are not case-sensitive, simplifying their use.
- **Whitespace Handling:** Extra white spaces in commands are ignored, ensuring flexibility in formatting.
- Commands can be issued through a serial terminal interface.
- The ICC-4C responds with ASCII-encoded messages (**Error! Reference source not found.**), making debugging and testing straightforward.

Table 1. List of simple mode commands for ICC-4C-500

Simple mode command	Description
START[CR][LF]	Driver answers "OK" if driver is ready to use and device is detected. Otherwise "ERROR" is received.
STATUS[CR][LF]	Driver answers with status encoded within 4Bytes information. Example: "0x00015000[CR][LF]" See next section for further description of the status bytes.
RESET[CR][LF]	Restarts driver's firmware. Note: no answer is sent via serial line.
GOTODFU[CR][LF]	Starts driver's loader for firmware update. Note: no answer is sent via serial line.
GOPRO[CR][LF]	Starts binary protocol-based mode of serial communication. Serial message CRC is not checked.
GOPROCRC[CR][LF]	Starts binary protocol-based mode of serial communication. Serial message CRC is checked.
GETID[CR][LF]	Answers with firmware serial number. Example: "14352500-00-A[CR][LF]"
GETVERSION[CR][LF]	Answers with firmware version number. Example: "1.0.740706[CR][LF]"
GETGITSHA1	Answers with 40 bytes hexadecimal GIT build identification.

	Example : "eb8115e6b04814f0c37146bbe3dbc35f3e8992e0[CR][LF]"
GETSN[CR][LF]	Answers with board and device serial number. Example: "Board: CDAA0057, Device: ANAA1234[CR][LF]"
GETDEVICESN[CR][LF]	Answers with serial number of a device connected depending on active channel. Example: "Device: ANAA1234[CR][LF]"
DETECTDEVICE[CR][LF]	Runs device autodetection on active channel. When detection is successful, answers with device name. Example: "XPR[CR][LF]"
SETCHANNEL=%int[CR][LF]	Set active channel. Channel index of ICC-4C is 0-3. Command is used as predecessor of other commands applying on active channel of ICC-4C driver.
GETCHANNEL[CR][LF]	Get active channel returns an integral number of active/selected channel, 0-3 for ICC-4C driver. Example: "2[CR][LF]"
SETCURRENT=%float[CR][LF]	Set current value for active channel. Command supports decimal parameter value in mA units. Current value is limited either by power capabilities of ICC driver itself or connected device.
GETCURRENT[CR][LF]	Answers with value of active channel current setpoint. Returned value is decimal number in units of milliamperes. Example: "15.6[CR][LF]"
SETFP=%float[CR][LF]	Sets focal power of active channel. Command supports decimal parameter value in units of diopters. Focal power is limited to detected lens device capability.
GETFP[CR][LF]	Answers with focal power of lens on active channel. Returned focal power is a decimal number in diopters. If no lens is detected, it returns "NO".
GETFPMIN[CR][LF]	Answers with focal power lower limit of lens device connected to active channel. Returned focal power is decimal value in diopters. If no lens is detected, it returns "NO".
GETFPMAX[CR][LF]	Answers with focal power upper limit of lens device connected to active channel. Returned focal power is a decimal value in diopters. If no lens is detected, it returns "NO".
GETTEMP[CR][LF]	Answers with actual temperature of device connected to active channel. Returned temperature is a decimal value in units of degree Celsius. Example : "27.54[CR][LF]"
SETTEMLIM=%f[CR][LF]	Set operational temperature limit for selected channel in degree Celsius.

Table 2. List of simple mode replies

Simple mode reply	Description
OK[CR][LF]	Command accepted and performed without limits.
NO[CR][LF]	Command not accepted, for any reason.
OL[CR][LF]	Command not accepted, because parameter reached lower limit.
OU[CR][LF]	Command not accepted, because parameter reached upper limit.
ERROR[CR][LF]	Command not available.

6.2. Simple mode protocol for ICC-4C-2000

In the following table, we have the list of simple mode commands and replies for ICC-4C-2000. The simple mode replies are exactly same as ICC-4C-500 in Table 2.

Table 3. List of simple mode commands for ICC-4C-2000

Simple mode command	Description
START[CR][LF]	Driver answers "OK" if driver is ready to use and device is detected. Otherwise "ERROR" is received.
STATUS[CR][LF]	Driver answers with status encoded within 4 bytes information. Example: "0x00015000[CR][LF]" See next section for further description of the status bytes.
RESET[CR][LF]	Restarts driver's firmware. Note: no answer is sent via serial line.
GOPRO[CR][LF]	Starts binary protocol-based mode of serial communication. Serial message CRC is not checked.
GOPROCRC[CR][LF]	Starts binary protocol-based mode of serial communication. Serial message CRC is checked.
GETID[CR][LF]	Answers with firmware serial number. Example: "14352500-00-A[CR][LF]"
GETVERSION[CR][LF]	Answers with firmware version number.

	Example: "1.0.740706[CR][LF]"
GETGITSHA1	Answers with 40 bytes hexadecimal GIT build identification. Example : "eb8115e6b04814f0c37146bbe3dbc35f3e8992e0[CR][LF]"
GETSN[CR][LF]	Answers with board and device serial number. Example: "Board: CDAA0057, Device: ANAA722[CR][LF]"
GETDEVICESN[CR][LF]	Answers with serial number of a device connected depending on active channel. Example: "Device: ANAA722[CR][LF]"
DETECTDEVICE[CR][LF]	Runs device autodetection on active channel. When detection is successful, answers with device name. Example: "XPR[CR][LF]"
SETCHANNEL=%int[CR][LF]	Set active channel. Channel index of ICC-4C is 0-3. Command is used as predecessor of other commands applying on active channel of ICC-4C driver.
GETCHANNEL[CR][LF]	Get active channel returns an integral number of active/selected channel, 0-3 for ICC-4C driver. Example: "2[CR][LF]"
SETCURRENT=%float[CR][LF]	Set current value for active channel. Command supports decimal parameter value in mA units. Current value is limited either by power capabilities of ICC driver itself or connected device.
GETCURRENT[CR][LF]	Answers with value of active channel current setpoint. Returned value is decimal number in units of milliamperes. Example: "15.6[CR][LF]"
GETTEMP[CR][LF]	Answers with actual temperature of device connected to active channel. Returned temperature is a decimal value in units of degrees of Celsius. Example : "27.54[CR][LF]"
SETTEMLIM=%f[CR][LF]	Set operational temperature limit for selected channel in degrees of Celsius.

6.3. Pro mode protocol

Pro mode is a byte-based protocol inspired by HDLC. Individual messages are delimited with the "delimiter" byte, 0x7e. To avoid the delimiter byte appearing in the message data, an "escape byte", 0x7d, is also used, and byte stuffing is performed on every message before being sent out (so the receiver should then perform byte destuffing) as follows: Every appearance of the bytes 0x7e or 0x7d in the data (so in a role other than the delimiter or the escape byte) is replaced by the sequence 0x7d 0x5e or 0x7d 0x5d, respectively. In other words, every byte that needs to be escaped has escaped by being XORed with 0x20 and prepended with 0x7d. When destuffing, every instance of 0x7d should be treated as an escape byte - it should not be interpreted as a data byte, it only marks that the following byte should be XORed with 0x20 before being interpreted as another data byte.

Above the delimiting and stuffing layer, every pro mode message has the following format:

Reserved (slave address)	Command	Data size	Data (payload)	CRC
1 byte	1 byte	1 byte	0 - 50 bytes	2 bytes

The semantics of the individual fields are as follows:

- **Slave address** – this field is currently unused
- **Command** – a 1-byte unique identifier of the selected command (see below)
- **Data size** – the number of bytes in the following data field
- **Data** – the data required by the command / response, limited to 50 bytes
- **CRC** – the CRC checksum calculated using CRC-16 CCITT algorithm (polynomial seed = 0x1021, initial value = 0xFFFF), present even if CRC is deactivated (the values can be ignored in that case)

Available pro mode commands:

Command name	Code	Command payload	Response payload	Note
Generic command	0x00	empty	empty	Used for special purposes, such as error responses to commands with unclear command code
Get firmware identification	0x01	empty	32bits firmware ID	N/A
Get status	0x02	empty	32bits status register	N/A
Start self-test	0x03	empty	empty	N/A
Load configuration	0x04	1 byte snapshot id	empty	3 snapshots are available. 0 is the factory snapshot id.
Store configuration	0x05	1 byte snapshot id	empty	3 snapshots are available. Factory snapshot is read only. 2 and 3 can be stored by the user. Snapshot saving takes approximately 200ms, please adjust timeout for communication to 300ms or more.
Set communication mode	0x06	1 byte mode	empty	0 for simple mode, unchanged otherwise
Set value	0x10	2 bytes register id, 4 bytes value	empty	see Architecture for explanation of Systems and registers
Get value	0x11	2 bytes register id	4 bytes register value	see Architecture for explanation of Systems and registers
Set multiple values	0x12	2 bytes value count = CNT, 2xCNT bytes register ids, 4xCNT bytes register values	empty	see Architecture for explanation of Systems and registers
Get multiple values	0x13	2 bytes value count = CNT, 2xCNT bytes register ids	2 bytes value count = CNT, 4xCNT bytes register values	see Architecture for explanation of Systems and registers
Set vector	0x14	2 bytes vector id, 2 bytes index, 1 byte length = SIZE, SIZE bytes data	empty	see Architecture for explanation of Systems and vectors; SIZE bytes starting at the index given are rewritten
Get vector	0x15	2 bytes vector id, 2 bytes index, 1 byte length = SIZE	SIZE bytes data	see Architecture for explanation of Systems and vectors; SIZE bytes starting at the index given are read

Responses:

Responses have the same format as the commands (possibly with different payload data, see above) with the command code of the response corresponding to the command code of the command being responded to. However, error responses are marked by the most significant bit being set in the command code (in other words, an 0x80 is added to the command code), and the payload should consist of a single 4 byte error flag, see the Error Codes section of the firmware documentation.

6.4. Pro mode example

The section below shows a detailed description of pro mode with example of setting the Signal generator and making a Temperature reading from the output stage of the ICC-4C controller. The example functions are written in Python, but they should be easy to understand and straightforward to implement on any platform.

The serial communication is first set to pro mode with disabled CRC using the "GOPRO" command. After the Pro mode communication has been established, the controller switches to a mode where it expects byte packets in the format below (different from simple commands), with the delimiter 0x7e.

When Pro mode is activated, none of the simple mode commands will work, until the specific command to change the communication mode back to simple mode is sent. After this command is sent, the simple mode communication is established, and the controller will again accept string-based input commands from the simple serial mode communication section.

Turning on the signal generator on the ICC-4C (channel 0 by default), after all its necessary parameters have been set up:

```
self.send_cmd(b'7e' + b'00' + b'10' + b'06' + b'6001' + b'00000001' + b'0000' + b'7e')
```

Where:

0x7e is the delimiter + 0x00 is an unused byte (slave address) + 0x10 is the *Set value* pro mode command + 0x06 is the data size of 6 bytes + 0x6001 is the register ID to run the signal generator for channel 0 + 0x00000001 is the

True boolean value to be written into the register + *0x0000* are the 2 unused CRC bytes + *0x7e* is the delimiter again.

Command name	Code	Command payload	Response payload	Note
Set value	0x10	2 bytes register id, 4 bytes value	empty	see Architecture for explanation of Systems and registers

Address	Name	Default Value	Description	Type	Access
0x6[0-7]01	Run	false	Turn Signal Generator on/off	boolean	read write

Reading the temperature of the output stage – register *0x2202* is done by sending the following command:

```
self.send_cmd (b'7e' + b'00' + b'11' + b'02' + b'2202' + b'0000' + b'7e')
```

Where:

0x7e is the delimiter + *0x00* is an unused byte (slave address) + *0x11* is the *Get value* pro mode command + *0x02* is the data size of 2 bytes + *0x2202* is the register ID for the output stage temperature + *0x0000* are the 2 unused CRC bytes + *0x7e* is the delimiter again.

Command name	Code	Command payload	Response payload	Note
Get value	0x11	2 bytes register id	4 bytes register value	see Architecture for explanation of Systems and registers

Address	Name	Default	Description	Type	Access
0x2202	Output stage board temperature	N/A	Degrees Celsius	float	read only

Switch command mode back to Simple mode:

```
self.send_cmd (b'7e' + b'00' + b'06' + b'01' + b'00' + b'0000' + b'7e')
```

Where:

0x7e is the delimiter + *0x00* is an unused byte (slave address) + *0x06* is the *Set communication mode* command + *0x01* is the data size of 1 byte + *0x00* is the *0* value to set the device in simple mode + *0x0000* are the 2 unused CRC bytes + *0x7e* is the delimiter again.

Command name	Code	Command payload	Response payload	Note
Set communication mode	0x06	1 byte mode	empty	0 for simple mode, unchanged otherwise

Example code (written in Python):

```
1. def float_to_ieee754_hex(x):  
2.     # Pack the float as a 32-bit float in IEEE 754 format  
3.     packed_float = struct.pack('>f', x)
```

```
4.     # Unpack the bytes as an unsigned integer and format as hex
5.     hex_representation = ''.join(f'{b:02x}' for b in packed_float)
6.     return hex_representation.encode('ascii')
7.
8.
9. def activate_signal_generator(self, channel=0):
10.    self.send_cmd(b'7e001006' + b'4' + str(channel).encode('ascii') + b'00' + b'00000060' +
11.    b'00007e')
12.
13. def set_current_unit_type_signal_generator(self, channel=0):
14.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'00' + b'00000000' +
15.    b'00007e')
16.
17. def set_fp_unit_type_signal_generator(self, channel=0):
18.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'00' + b'00000003' +
19.    b'00007e')
20.
21. def run_signal_generator(self, channel=0):
22.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'01' + b'00000001' +
23.    b'00007e')
24.
25. def stop_signal_generator(self, channel=0):
26.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'01' + b'00000000' +
27.    b'00007e')
28.
29. def set_shape_signal_generator(self, channel=0, shape=0):
30.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'02' + b'0000000' +
31.    str(shape).encode('ascii') + b'00007e')
32.
33. def set_frequency(self, channel=0, frequency=0.0):
34.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'03' +
35.    float_to_ieee754_hex(frequency) + b'00007e')
36.
37. def set_amplitude(self, channel=0, amplitude=0.0):
38.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'04' +
39.    float_to_ieee754_hex(amplitude) + b'00007e')
40.
41. def set_offset(self, channel=0, offset=0.0):
42.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'05' +
43.    float_to_ieee754_hex(offset) + b'00007e')
44.
45. def set_phase(self, channel=0, phase=0.0):
46.    self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'06' +
47.    float_to_ieee754_hex(phase) + b'00007e')
48.
49. def set_cycles_signal_generator(self, channel=0, cycles=-1):
50.     # negative value sets infinite loop
51.     if cycles == -1:
52.         self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'07' + b'ffffffff'
53.         + b'00007e')
54.     else:
55.         self.send_cmd(b'7e001006' + b'6' + str(channel).encode('ascii') + b'07' + str(cy-
56.         cles).encode('ascii') + b'00007e')
57.
```

```
58. def get_output_stage_temperature(self):  
59.     self.send_cmd(b'7e001102220200007e')  
60.  
61.  
62. def get_power_supply_temperature(self):  
63.     self.send_cmd(b'7e001102220400007e')  
64.
```

7. I2C Communication Protocol

In addition to UART and Ethernet communication, the ICC-4C also supports I2C communication, making it possible to easily integrate into MCU controlled systems. This is particularly advantageous when the user wants to control the ICC-4C as a part of a larger industrial system and align it with different cameras and actuators used by that system. Auxiliary I/O connector has specific pins for I2C (pins 13 and 15).

7.1. I2C Configuration

To set up an I2C communication with the ICC-4C, three parameters can be configured in the device's Board EEPROM settings:

1. The "I2C slave address" setting is used to store the I2C address for the ICC-4C.
The default address is set to 0x20, but can be changed by the user to any desired value.
2. The "I2C register count" setting determines, how many of the ICC-4C's registers are changed by a single I2C communication, e. g. how many data bytes are following the device address during the I2C communication.
The default value is set to be 2 registers, but it is possible to set it to 1 or 4 registers at a time as well.
3. To apply the configured settings, press the "Set to device" button and click "Yes" when asked for board reset.

7.2. I2C communication protocol description

Depending on the "I2C register count" setting in the Board EEPROM, the ICC-4C can be configured via I2C communication by setting 1, 2 or 4 registers at a time. Since the device's registers are defined with a 4-digit hexadecimal address and they are storing a 32-bit data value, changing a single register via I2C requires sending 6 bytes of data (1 byte system ID + 1 byte register address and 4 bytes of data) in a single communication. If the device is set to configure multiple registers, then the final number of bytes sent after the I2C address should be the "I2C register count" setting times 6 bytes (it should be 6, 12 or 24 bytes).

Reading data from the controller is done via 4 read pointer registers at addresses from 0x3100 to 0x3103. Writing a system ID and register address into these registers will copy the content of the given register into this pointer and then it can be read out by the following I2C read command. The device also contains 4 write error response registers (addresses from 0x3104 to 0x3107), which contain a possible error response for the previously initiated write command through I2C and it can be also read out by the following I2C read command.

Address	Name	Default Value	Description	Type	Access
0x3100	I2C read pointer 0	0x1003	The address of the register that will be used to evaluate the first register in the read request Value is updated at controller frequency.	uint32	read write
0x3101	I2C read pointer 1	0x1004	The address of the register that will be used to evaluate the second register in the read request Value is updated at controller frequency.	uint32	read write
0x3102	I2C read pointer 2	0x1003	The address of the register that will be used to evaluate the first register in the read request Value is updated at controller frequency.	uint32	read write
0x3103	I2C read pointer 3	0x1004	The address of the register that will be used to evaluate the second register in the read request Value is updated at controller frequency.	uint32	read write
0x3104	Write error response 0	0	Error response of the first register of the write request.	uint32	read only
0x3105	Write error response 1	0	Error response of the second register of the write request.	uint32	read only
0x3106	Write error response 2	0	Error response of the third register of the write request.	uint32	read only
0x3107	Write error response 3	0	Error response of the forth register of the write request.	uint32	read only

When the device has been configured to write 2 or 4 registers at the same time, but only fewer registers are required to be changed by the user, the unneeded register write commands can be replaced by configuring the read pointers to copy the value of any system and register ID.

I2C slave driver which supports only raw I2C read/write frames. SMBus frames are not supported. Speed clock capability of driver is up to 400kHz with 100ns rise time and 10ns fall time. The driver supports reception and transmission of 12 data byte frames only (13 bytes with the address byte in total).

Master has the possibility to read/write two system registers. However, all 12 data bytes have to be clocked by the master even if one system register needs to be manipulated. If a different size of data frame other than 12 is sent, data will be corrupted until receive buffer of slave is fully flushed.

Two types of commands are supported: Write command and Read command. We will discuss this in the following subsections:

7.3. I2C Write command structure

The following examples show the I2C command structure for setting 2 registers at the same time (default register count setting).

- Set 2.5 dpt focal power as static input (1st register to write):
 - System ID for "STATIC INPUT": 0x50.
 - Register address for "FP STATIC INPUT": 0x04.
 - The 32-bit float of 2.5 dpt in hexadecimal format corresponds to 0x40200000, so the data bytes are: 0x40 0x20 0x00 0x00.
- Set Read pointer 1 for the lens temperature (2nd register to write):
 - System ID for the I2C Read Pointers: 0x31.
 - Register address for I2C Read Pointer 1: 0x01.
 - The System ID for "TEMPERATURE MANAGER" is 0x22, and the register address for "DEVICE TEMPERATURE" is 0x00.

	I2C slave address	Write to the 1 st register						Write to the 2 nd register					
Byte#	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12
Meaning	I2C slave address	Sys. ID 1	Reg. address 1	Reg. data 1	Reg. data 1	Reg. data 1	Reg. data 1	Sys. ID 2	Reg. address 2	Empty byte	Empty byte	Sys. ID to read	Reg. address to read
Hex value	0x20	0x50	0x04	0x40	0x20	0x00	0x00	0x31	0x01	0x00	0x00	0x22	0x00
Operation	Select device (write)	Set 2.5 dpt focal power as static input						Set Read pointer 1 for lens temperature register					

7.4. I2C Read command structure

Reading of the ICC-4C's registers can be done in two steps. In the first step the read pointers must be configured:

- Set Read pointer 0 for the actual output current (1st register to write):
 - System ID for the I2C Read Pointers: 0x31.
 - Register address for I2C Read Pointer 0: 0x00.

- The System ID for “LINEAR OUTPUT” is *0xE8*, and the register address for “OUTPUT CURRENT” is *0x02*.
- Set Read pointer 1 for the lens temperature (2nd register to write):
 - System ID for the I2C Read Pointers: *0x31*.
 - Register address for I2C Read Pointer 1: *0x01*.
 - The System ID for “TEMPERATURE MANAGER” is *0x22*, and the register address for “DEVICE TEMPERATURE” is *0x00*.

	I2C slave address	Write to the 1 st register						Write to the 2 nd register					
Byte#	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12
Meaning	I2C slave address	Sys. ID 1	Reg. address 1	Empty byte	Empty byte	Sys. ID to read 1	Reg. address to read 1	Sys. ID 2	Reg. address 2	Empty byte	Empty byte	Sys. ID to read 2	Reg. address to read 2
Hex value	0x20	0x31	0x00	0x00	0x00	0xE8	0x02	0x31	0x01	0x00	0x00	0x22	0x00
Operation	Select device (write)	Set Read pointer 0 for output current register						Set Read pointer 1 for lens temperature register					

In the next step the host can read out the I2C data:

- Data from Read pointer 0 for the actual output current (read from the 1st register):
 - System ID from Read pointer 0: *0xE8*.
 - Register address from Read pointer 0: *0x02*.
 - Data from Read pointer 0: *0x3E0DA1B0*, which in 32-bit float format equals to 0.1383121 A.
- Data from Read pointer 1 for the actual lens temperature (read from the 2nd register):
 - System ID from Read pointer 1: *0x22*.
 - Register address from Read pointer 1: *0x00*.
 - Data from Read pointer 1: *0x41FD0000*, which in 32-bit float format equals to 31.625 °C.

	I2C slave address	Read from the 1 st register						Read from the 2 nd register					
Byte#	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12
Meaning	I2C slave address	Sys. ID 1	Reg. address 1	Reg. data 1	Reg. data 1	Reg. data 1	Reg. data 1	Sys. ID 2	Reg. address 2	Reg. data 2	Reg. data 2	Reg. data 2	Reg. data 2
Hex value	0x20	0xE8	0x02	0x3E	0x0D	0xA1	0xB0	0x22	0x00	0x41	0xFD	0x00	0x00

Operation	Select device (read)	Read pointer 0 containing Sys. ID, reg. address and data from the output current register	Read pointer 1 containing Sys. ID, reg. address and data from the lens temperature register
-----------	----------------------	---	---

7.5. Changing the I2C register count

The following examples show changing of the I2C register count command structure for setting 2 registers at the same time (default register count setting).

- Unlock the Board EEPROM (1st register to write):
 - System ID for "BOARD EEPROM": 0x20.
 - Register address for "LOCK": 0x00.
 - The 32-bit key value to unlock the Board EEPROM: 0x3F4744F6.
- Change I2C register count to 1 (2nd register to write):
 - System ID for "BOARD EEPROM": 0x20.
 - Register address for "SLAVE I2C REGISTER COUNT": 0x17.
 - The 32-bit integer value for 1 register in hexadecimal format corresponds to 0x00000001, so the data bytes are: 0x00 0x00 0x00 0x01.

	I2C slave address	Write to the 1 st register						Write to the 2 nd register					
Byte#	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8	Byte 9	Byte 10	Byte 11	Byte 12
Meaning	I2C slave address	Sys. ID 1	Reg. address 1	Reg. data 1	Reg. data 1	Reg. data 1	Reg. data 1	Sys. ID 2	Reg. address 2	Empty byte	Empty byte	Empty byte	Reg. data 2
Hex value	0x20	0x20	0x00	0x3F	0x47	0x44	0xF6	0x20	0x17	0x00	0x00	0x00	0x01
Operation	Select device (write)	Unlock the Board EEPROM						Set I2C register count to 1					

After changing the I2C register count settings in the EEPROM, a **Board Reset (power cycling)** should be performed for the changes to take effect.

8. Ethernet and TCP Communication

IMPORTANT: For reliable Ethernet communication, it is recommended to update the firmware of ICC-4C to version FW 2.1.742001 and later. Ethernet communication with earlier versions might experience some bugs.

Besides UART and I2C, the ICC-4C is also capable of communicating via Ethernet, which allows seamless implementation into industrial systems or being accessible from multiple host computers. During the Ethernet communication, by default the device uses TCP/IP communication port number 5000. For IP address detection purposes, the device replies to any broadcast message with the controller's ID and serial number over the UDP port number 30321.

The IP settings of the device can be configured automatically by enabling the DHCP configuration feature, so that the device can obtain an unused IP address from the server (this configuration is active by default). Alternatively, the users can set manual IP addressing by disabling the DHCP feature and configuring the IP Address, the Subnet Mask and the Default Gateway manually. All the IP configuration settings are stored in the ICC-4Cs Board EEPROM and can be changed using Optotune Cockpit software, the available SDKs (C# or Python) or any other communication interface (UART Pro mode or I2C).

Here we define some basic concepts: Ethernet is network interface – configuration of the device needs to be matched with configuration of connected network. If device network configuration is not correct, the communication with device will not work, or will not always work (depending on actual state of routing tables).

The important network parameters are:

- IP address
- Port number
- Subnet mask
- Default Gateway

IP address: address used for identification of device in the network:

- Can be assigned statically or dynamically – DHCP or static mode
- IP address must be unique for each device in the network (IP address conflict)

Port number: each device can communicate in multiple “channels” in parallel

- Port is dedicated communication channel for specific service
- Firewalls or other network protection services may block specific ports, or need to unblock non-standard port number
- ICC-4C drivers use by default port number 5000 for TCP and 30321 for UDP
- Port numbers can be configured to any number – change it only when you need it

Subnet mask: defines subnetwork on which the device communicates

- Only the devices from the same subnet can communicate together
- Recipient IP address is checked with subnet mask – if it doesn't match with subnet, the message is not processed
- The most common subnet mask is 255.255.255.0
- Subnet of devices with 254 possible addresses that differ only in the last number
- For example, 192.168.1.1 – 192.168.1.254

Default Gateway: defines where to send message addressed to different subnet

- If a message is received from different subnet, the response will be sent here
- Can create connection between different subnets – usually address of a router is used

The driver's network parameters are stored in the Board EEPROM, and they can be edited using Optocockpit via either a USB or Ethernet connection. When changing network parameters over Ethernet, the connection may be lost, but you can always reconnect using USB to modify the network settings:

TCP/IP port	5000	
UDP port	30321	
DHCP enabled	☐	✓
I2C slave address	0x20	✓

IP address	192.168.1.4	✓
IP subnet mask	255.255.255.0	✓
IP default gateway	192.168.1.10	✓

There are two typical use cases for using ICC-4C drivers with Ethernet:

1. Static configuration between driver and PC
2. Driver connected to local network – DHCP or static variant

8.1 IP settings configuration via DHCP

The IP settings of the ICC-4C can be configured to obtain an IP address from a nearby DHCP server. This option is selected by factory defaults, but can be changed by the user under the Board EEPROM settings in Optotune Cockpit:

1. Open the Board EEPROM widget and check the “DHCP enabled” checkbox.
2. Press the “Set to device” button and click “Yes” when asked for board reset.

Set to device 2.		
Key	Value	Editable
EEPROM Version	21	
EEPROM Subversion	0	
Part number	183-428-00	
Part configuration	0	
Part revision	0	
Serial number	DIAA0045	
MAC address	8C:1F:64:65:62:4E	
IP address	192.168.1.2	✓
IP subnet mask	255.255.255.0	✓
IP default gateway	192.168.1.1	✓
IP DNS	10.0.20.1	✓
IPv6 address	ffff:ffff:ffff:ffff:ffff:ffff	✓
IPv6 subnet mask	ffff:ffff:ffff:ffff:ffff:ffff	✓
IPv6 default gateway	ffff:ffff:ffff:ffff:ffff:ffff	✓
IPv6 DNS	ffff:ffff:ffff:ffff:ffff:ffff	✓
TCP/IP port	5000	
UDP port	30321	
DHCP enabled	☒ 1.	✓
I2C slave address	0x20	✓

Figure 31: Enabling IP configuration via DHCP in Optotune Cockpit

8.2 Static IP address configuration

If the ICC-4C is connected directly to a computer via Ethernet, or if the connected network/router is configured with static IP addressing, the controller can be also configured for static IP addressing. This can be also done in the Optotune Cockpit software:

1. Open the Board EEPROM widget and uncheck the “DHCP enabled” checkbox.
 - a. Leaving this box checked will ignore the static IP settings and the device will try to obtain the IP address from a server, even if there isn’t one.
2. Configure the “IP address”, the “IP subnet mask” and the “IP default gateway” fields for the device:
 - a. For the “IP address” setting, make sure that the selected address is a usable Host IP address, and not an address for the entire subnetwork or a broadcast address.
 - b. The “IP subnet mask” should match the one configured on the host PC or the connected sub-network.
 - c. The “IP default gateway” should be configured to the IP address of the host PC, if they are connected directly, or if the controller is connected to a larger network, it usually should be the IP address of the router for the given network.
3. Press the “Set to device” button and click “Yes” when asked for board reset.
4. If a direct connection to a host PC is used, the IP address should also be configured on the PC:
 - a. Open the Control Panel and navigate to Network and Sharing Center -> Ethernet -> Properties -> Internet Protocol Version 4 (TCP/IPv4) -> Properties.
 - b. Select “Use the following IP address:” option.
 - c. The IP address of the PC should match the previously set “IP default gateway” from the ICC-4C.
 - d. The Subnet mask should be the same as the one configured on the ICC-4C.
 - e. The Default gateway of the PC should match the previously set “IP address” from the ICC-4C.
5. Click on OK in both Properties windows.

Set to device **3.**

Key	Value	Editable
EEPROM Version	21	
EEPROM Subversion	0	
Part number	183-428-00	
Part configuration	0	
Part revision	0	
Serial number	DIAA0045	
MAC address	8C:1F:64:65:62:4E	
IP address	192.168.1.2 2a.	✓
IP subnet mask	255.255.255.0 2b.	✓
IP default gateway	192.168.1.1 2c.	✓
IP DNS	10.0.20.1	✓
IPv6 address	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff	✓
IPv6 subnet mask	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff	✓
IPv6 default gateway	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff	✓
IPv6 DNS	ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff	✓
TCP/IP port	5000	
UDP port	30321	
DHCP enabled	☐ 1.	✓
IP slave address		

Figure 32: Example of a static IP address configuration in Optotune Cockpit

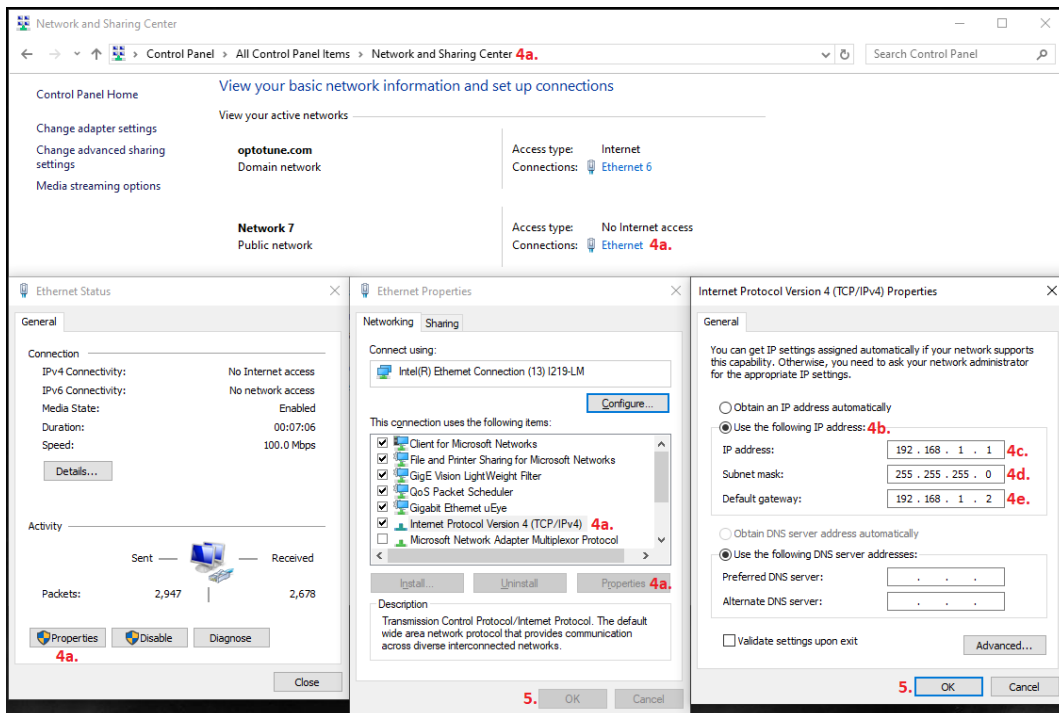


Figure 33: Example of a static IP address configuration in Windows

8.3 Ethernet communication example via Cockpit or a Telnet Client

To connect with the ICC-4C via Ethernet, the users can use the Optotune Cockpit software by selecting the Ethernet tab in the “Add electronics” window, but any Telnet Client software should work (for example Putty or Hercules).

To initiate a TCP communication via a Telnet Client, the IP address of the device should be entered to the appropriate field and TCP port number 5000 shall be selected. During the Ethernet communication, the ICC-4C uses the same set of commands and replies as for the UART communication. It is possible to use the simple mode commands terminated with a <CR> <LF> sequence, or switch to Pro mode and sending binary commands to the controller.

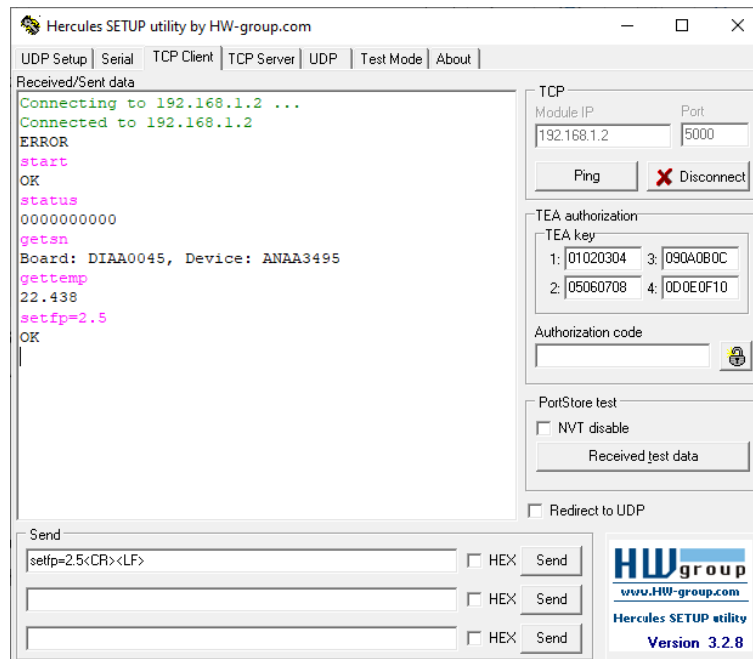


Figure 34: Ethernet communication with the ICC-4C using a Telnet Client

Upon connecting to the ICC-4C, depending on the given client software an “ERROR” message might be received – this is caused by the software sending some identification request, which is not part of the ICC-4C’s instruction set and thus the controller replies with an error. This message can be ignored, and the user could proceed by sending the “START” command to initialize the ICC-4C.

8.4 UDP messaging

UDP messaging enables multicast communication from one sender to multiple receivers, typically by using a broadcast or subnet broadcast address, where the last number of the address is 255. When the driver receives the “OptotuneSearch” message, it responds with its serial number and IP network configuration. A full broadcast message (255.255.255.255) is received by every device on the network; however, if the sender is not on the same subnet, the driver’s response is sent to the default gateway. If the gateway or network is not configured correctly, this response may not reach the sender.



Figure 35: UDP messaging

- As UDP broadcast allows to reach driver with any network configuration, it can be used for IP network parameters update
- Specific message format: SN number of driver is the identificatory:

<SerialNumber>;DHCP:<0 or 1>;IP:<IP address value>;SN:<subnet mask value>;GW:<gateway value>;

- Example: it might take up to 10 seconds until the driver will start responding to discovery

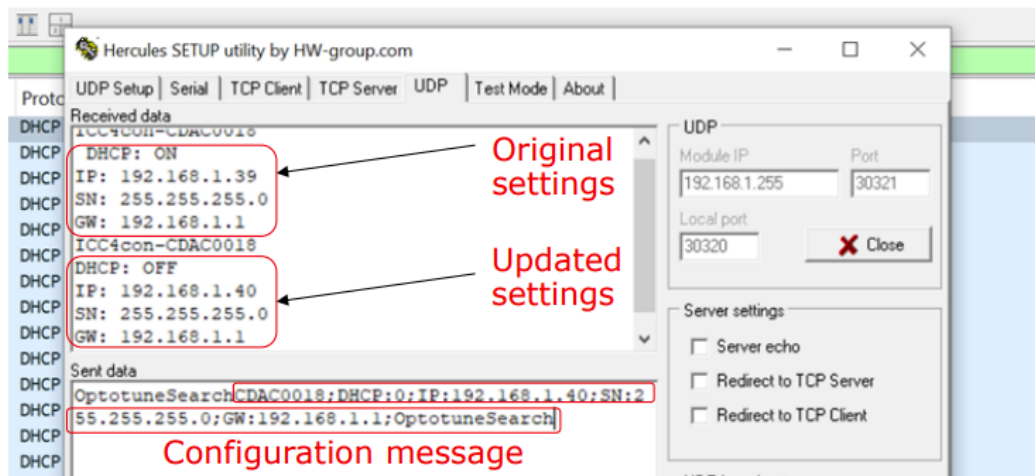


Figure 36. UDP messaging for ICC-4C-500

9. Analog input mode

For the ICC-4C it is possible to map an analog input voltage 0-10 V to the driving current or the focal power (if applicable) of the connected lens, or to the current or position of the BWS/FMR/XPR. Both linear and non-linear mapping are possible. With linear interpolation, the controller linearly interpolates the analog voltage input range (0-10 V) to the corresponding pre-set output current or focal power range. With non-linear analog voltage transition, mapping points can be added manually by the user. For each point, the user can set the input voltage and the corresponding output current or focal power. A toggle button is available to enable constant or linear extrapolation outside of the defined points.

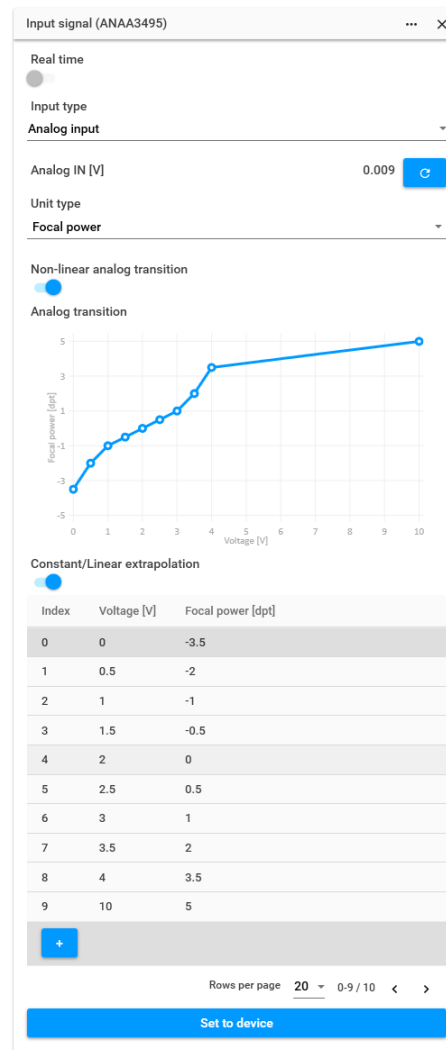


Figure 37: Analog input voltage with non-linear mapping of focal power values

Since the Analog input voltage mode may require to configure multiple points at each startup of the controller, the ICC-4C also offers a Snapshot Manager feature, which allows the user to store the settings in the device's non-volatile memory, so the snapshot with the mapping points will not be lost after powering off the device.

The ICC-4C has 4 snapshots available, which can be managed by the Snapshot manager widget in the Cockpit software. Snapshot **number one** is the "Factory settings," which allows the user to reload the factory default settings on the device (this snapshot cannot be edited or overwritten). Users can create and store a new configuration in the second available snapshot. It is also possible to choose which snapshot should be loaded during the device

start-up. For example, a user-defined snapshot can be saved as number 2 by pressing "Save" button. It is recommended to also click on the "Load" button afterwards to check for potential failures. After that the snapshot number can be selected under the "Startup snapshot" section and using the "Set" button it can be configured to be the active setting after a device startup.



Figure 38: Snapshot manager widget

The same procedure can be applied for other channels as well. After the snapshot is saved, the ICC-4C can be disconnected from the host and the analog voltage sources can be applied to the corresponding pins:



Connector¹: Würth Electronics header 100 mil 90deg

Pin	Signal	Value	Function
1	AnalogIn2	0..10 V	Analog input signal #2
2	AnalogIn3	0..10 V	Analog input signal #3
3	AnalogIn0	0..10 V	Analog input signal #0
4	AnalogIn1	0..10 V	Analog input signal #1
5	nRESET	0..3.3V (CMOS level)	Driver reset signal (inverted level)
6	AGND	-	Analog signal ground
7	USART_EXT_TX	0..3.3V (CMOS level)	UART transmit line
8	SYNC_B	0..3.3V (CMOS level)	Synchronization input B
9	USART_EXT_RX	0..3.3V (CMOS level)	UART receive line
10	SYNC_A	0..3.3V (CMOS level)	Synchronization input A
11	GND	-	Digital ground
12	GPIO3_SPI_MOSI	0..3.3V (CMOS level)	General purpose digital IO #3 or SPI MOSI signal (Trigger OUT/IN)
13	I2C_HOST_SDA	0..3.3V (CMOS level)	I2C data line from host
14	GPIO2_SPI_MISO	0..3.3V (CMOS level)	General purpose digital IO #2 or SPI MISO signal (Trigger OUT/IN)
15	I2C_HOST_SCL	0..3.3V (CMOS level)	I2C clock line from host
16	GPIO1_SPI_NSS	0..3.3V (CMOS level)	General purpose digital IO #1 or SPI SlaveSelect signal (Trigger OUT/IN)
17	GND	-	Digital ground
18	GPIO0_SPI_SCK	0..3.3V (CMOS level)	General purpose digital IO #0 or SCK signal (Trigger OUT/IN)
19	VCC_EXT_IN	24...48 V	External DC supply voltage positive input
20	GND_EXT_IN	-	External DC supply voltage negative input

Figure 39. ICC-4C Analog pins

10. Input/Output trigger signals

The ICC-4C can be configured to generate an output trigger signal when changing the focal power/current of the lens, or current of a BSW/FMR/XPR. Alternatively, it can also wait for an incoming trigger signal before executing a predefined change on the connected device. These trigger signals can be synchronized with the controller's built-in signal generator or any custom vector. For the ICC-4C, the following type of waveforms can be defined:

- Sinusoidal
- Triangular
- Square
- Sawtooth
- Pulse
- Staircase
- Fast autofocus waveform
- Any custom vector

The signal generator of the ICC-4C outputs a trigger signal from the GPIOx pins (0,1,2,3 depending on the channel, see Input/Output connector pinout for pin assignment). The trigger signal is HIGH (3.3V, max. 5 mA) at phase 0° of the selected waveform and goes LOW in the middle of the period (Figure 29). For pulse pattern, it reflects the duty cycle.

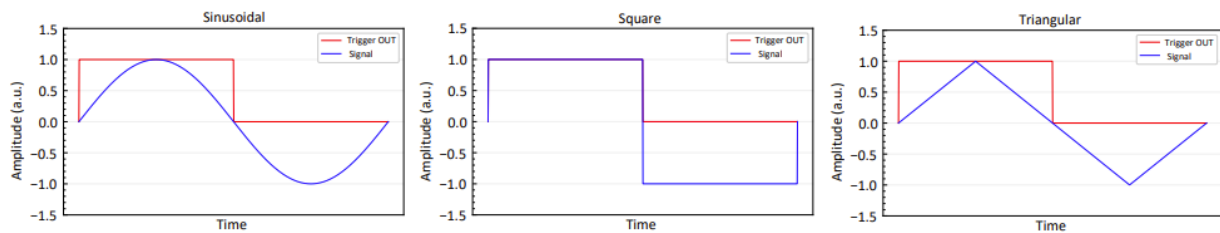


Figure 40: Different waveforms overlapped with the corresponding Trigger OUT signal

When configured as inputs, the trigger signals can be used to start the built-in signal generator or a preconfigured custom vector. When the trigger input signal goes HIGH (max 3.3V), the selected waveform starts off at phase 0°.

The signal generator of the ICC-4C can be synchronized with an external input trigger. When the trigger input signal goes HIGH (max 3.3V), the selected waveform starts off at phase 0° (Figure 30). There are two options:

- SYNC_A pin: an input signal to this pin synchronizes all channels simultaneously
- GPIOx: by default, GPIOx pins are programmed to be used as trigger output (see previous paragraph). They can be reprogrammed to be used as trigger input for single channels (0, 1, 2, 3). In this case, trigger input function is overridden by GPIOx and hence any signal on SYNC_A is ignored.

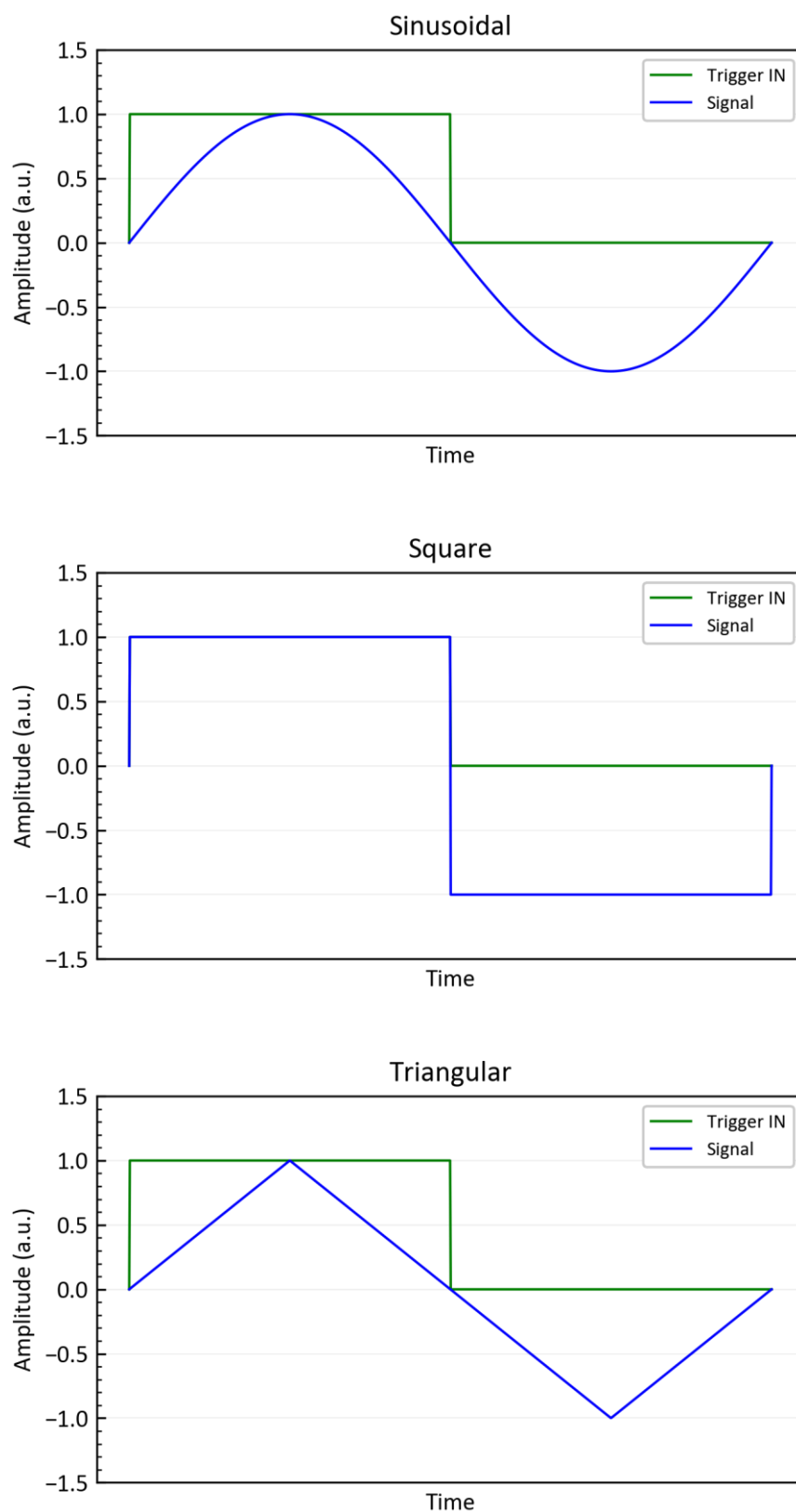


Figure 41: Different waveforms overlapped with an exemplary Trigger IN signal

11. Troubleshooting and FAQs

11.1. Status LED

The ICC-4C has 5 status LEDs on its front panel to indicate potential errors/warnings – one is showing the general status of the whole controller (Main Status) and the other one is showing issues related to the output channel (Out Status).

LED	Color	Legend
Main Status	Red	Power on, no connection
	Orange ¹	Operation OK (possible error)
	Green	Operation OK
Channel Status	Red	Device error
	Green	Device detected; operation OK

¹ mixed from red and green LEDs. Depending on the viewing angle, one color might be more dominant.

Both indicators are two-color LEDs (red/green) mounted in the same package, and the firmware is turning on either one of them (Red or Green) or mixing both at the same time (Orange). For the Orange color, one of the consisting colors might be more dominant depending on the view angle.

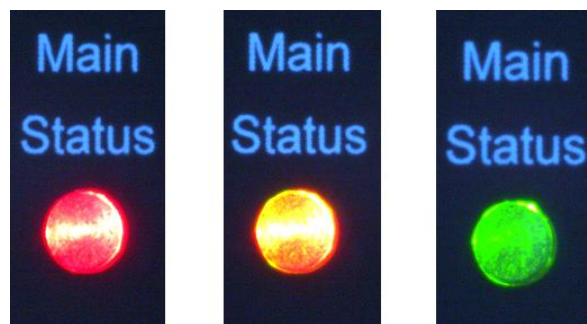


Figure 42: Pictures of the Main Status LED during various operation modes (Red, Orange, Green)

In normal operation, **both LEDs should be glowing Green**, when the device is powered, but below is a list of the most common error cases:

- **All LEDs OFF** – firmware issue. Potentially can be solved by the “Device recovery” feature in Cockpit or in some cases by updating the firmware.
- **Main status LED is Red** – the device is unable to initiate communication. Check the communication cables (USB, Ethernet).
- **Main status LED is Orange; Channel Status is OFF** – device not connected/not detected. Check the Hirose or the flex cable².
- **Out Status is Red** – Device error (for example overcurrent detected)

Further information can be obtained about the errors given/warnings by the Optotune Cockpit software or by reading the device’s status registers.

²Always power the device off before handling the flex cable, otherwise EEPROM corruption and/or permanent damage to lens or controller may occur.

11.2. Diagnostics in Optotune Cockpit

If the device can be connected to a host computer via USB or Ethernet, the Optotune Cockpit software can provide multiple diagnostic features. If some error/warning is present on the controller, a red or orange triangle can show up next to the device's name in the menu under the Electronics section. Clicking on this triangle opens a description of this error/warning and allows the user to clear the given message.

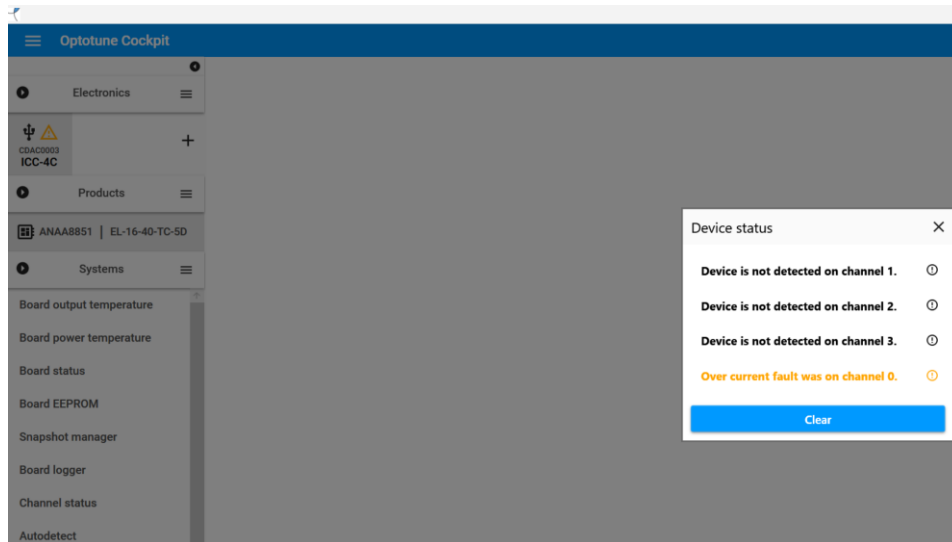


Figure 43: Example of a warning shown in Cockpit

Additional information can be collected by inspecting the data provided by the various **Temperature readouts**, the **Board status** and **Channel status** widgets.

The **Board EEPROM** widget can provide useful information about the settings stored in the controller, while the **EEPROM read/write** widget allows to check the data related to the connected lens.

The **Board Logger Widget** allows us to monitor some of the most important values in the controller and any custom value (register address) over time and export the data into a .csv file.

11.3. Diagnostics with the device manager and Communication Terminal

If the controller cannot be recognized by Cockpit, it might be still possible to run some diagnostics using the **Windows Device Manager** and some Serial terminal software, like [Termite](#) or [Putty](#). Normally the ICC-4C should show up in the Device Manager as a **USB Serial Device** under the COM Ports section, but in case of firmware issues it can be also present itself as a DFU in FS Mode device or some unrecognized USB device.

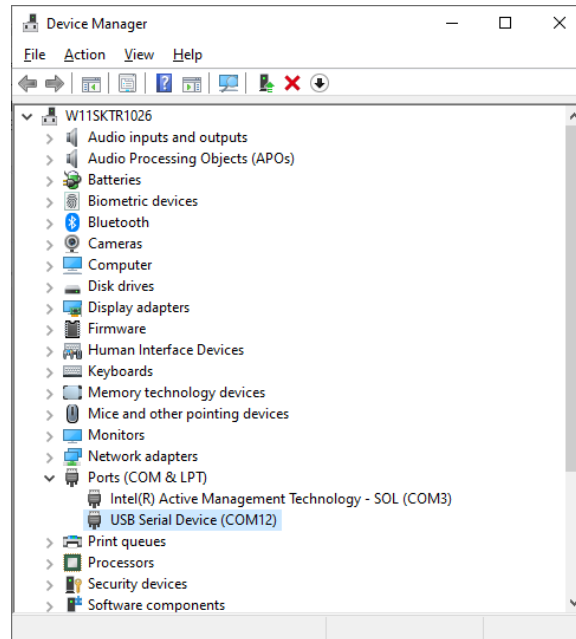


Figure 44. ICC-4C showing up in the Device Manager

If the device can be detected as a USB Serial Device, the user can connect to it by using terminal software and do some basic debugging. The settings for the software should be the following:

- **COM Port:** The same COM port as the one seen in the Device Manager.
- **Baud Rate:** Any (should be auto-detected by the device, but for debugging purposes Baud rates like 57600 or lower is recommended).
- **Data Bits:** 8
- **Stop Bits:** 1
- **Parity:** None (N)

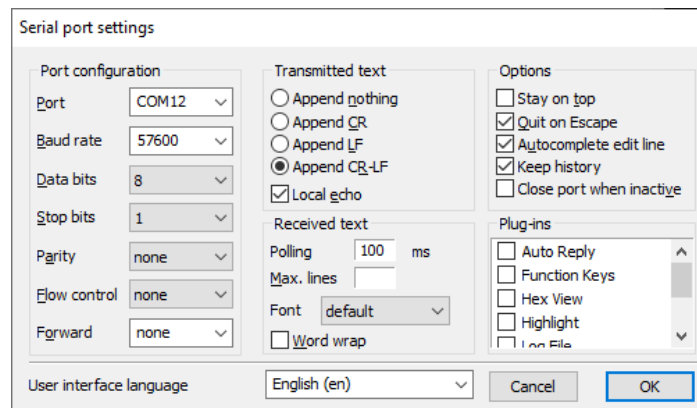


Figure 45: Connection settings for the ICC-4C in Termite

When connected to the COM port, the ICC-4C should accept any simple mode command described in the UART Communication Protocol section of this document. The most useful commands for diagnostics are the following:

- **START[CR][LF]** – Answers “OK” if controller is ready to use and a device is detected. Otherwise “ERROR” is received.

- **STATUS[CR][LF]** – The controller answers with a status encoded within 4 bytes of information (for example “0x00015000[CR][LF]”), which directly represents the data in the ICC-4C status register in the firmware (register address 0x1007).
- **RESET[CR][LF]** – Restarts the controller’s firmware. No answer is received after this command.
- **GOTODFU[CR][LF]** – Starts the controller’s bootloader for firmware update. No answer is received after this command, but if successful the device should change into a “DFU in FS Mode” device in the Device Manager and can be firmware can be recovered or updated in Cockpit.

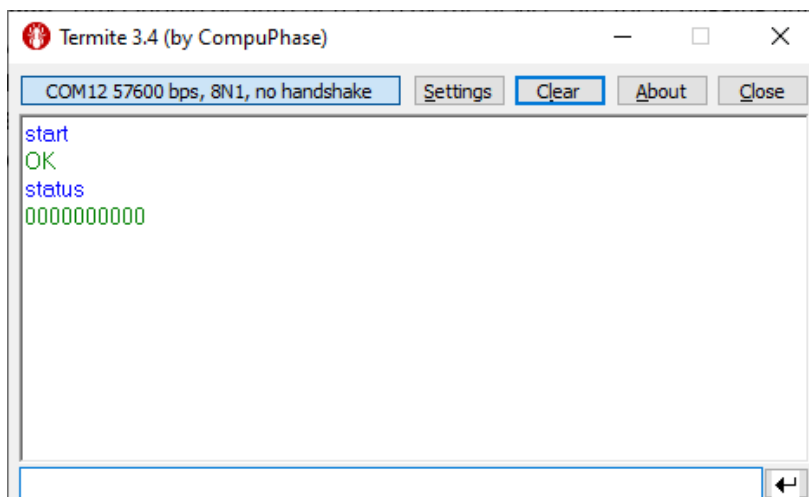


Figure 46: Communicating with the ICC-4C in Termit

0x1007	System status register	0 - No error	Register encodes system status and errors	Read: Bit# 0 - Channel 0 output fault condition 1 - Channel 0 output had fault condition 2 - Channel 1 output fault condition 3 - Channel 1 output had fault condition 4 - Channel 2 output fault condition 5 - Channel 2 output had fault condition 6 - Channel 3 output fault condition 7 - Channel 3 output had fault condition 8 - Driver over-heat fault 9 - Driver had over-heat fault 10 - Device on channel 0 not detected 11 - Device on channel 0 was not detected 12 - Device on channel 1 not detected 13 - Device on channel 1 was not detected 14 - Device on channel 2 not detected 15 - Device on channel 2 was not detected 16 - Device on channel 3 not detected 17 - Device on channel 3 was not detected 18 - Channel 0 has over-current fault on 3V3 supply line 19 - Channel 0 had over-current fault on 3V3 supply line 20 - Channel 1 has over-current fault on 3V3 supply line 21 - Channel 1 had over-current fault on 3V3 supply line 22 - Channel 2 has over-current fault on 3V3 supply line 23 - Channel 2 had over-current fault on 3V3 supply line 24 - Channel 3 has over-current fault on 3V3 supply line 25 - Channel 3 had over-current fault on 3V3 supply line 26-31 - Reserved Write: Writing to this register resets all history error flags (odd bits 1, 3, ... 25)	read write
--------	------------------------	--------------	---	---	------------

Figure 47: The ICC-4C’s firmware status register (reg. address 0x1007)

11.4. FAQs

Question 1:

The ICC-4C is unable to detect the connected lens or it does not recognize the lens type. What can I do?

Answer 1:

Check the device status register via the Cockpit software or terminal. Also check if the “Autodetect” feature is enabled on the controller. Some lenses do not have EEPROM, so they show up as Unknown devices and can be controlled only in current mode.

Question 2:

My lens has EEPROM, but I still can’t use Focal power mode. What can I do?

Answer 2:

If the lens has EEPROM (check the datasheet), but it’s not accessible, it may indicate data corruption. To restore the data via the Cockpit software, please contact Optotune’s support team for the original lens EEPROM data and further instructions.

Question 3:

The ICC-4C stopped working after an unsuccessful (for example unintentionally interrupted) firmware update and is not recognized by Optotune Cockpit any longer. What can I do?

Answer 3:

In some cases, it may be possible to restore the device’s firmware by opening the menu in the upper left corner of our Cockpit software and select “Device recovery”. Select the “Electronic type” and the device you want to restore and click “Recover”. This solution usually works if the device is stuck in Device Firmware Upgrade (DFU) mode.

If there is no selectable device, then a full hardware reset might still be possible – please contact Optotune’s support team further instructions about this.

Question 4:

I’m unable to change the Board EEPROM settings (for example Ethernet settings or I2C address, etc.). What is the issue?

Answer 4:

This may indicate that the controller’s Board EEPROM data got corrupted. It can be easily verified by opening the Board EEPROM widget and scrolling down to the “Board Parsing Result” field – if it says “NOK”, please contact Optotune’s support team for the original EEPROM data for the given controller and further instructions.