

# **CORDIS** SUITE

**MachineControlServer C# client library**

Demonstration application

2020-03-17



Cordis Automation B.V.

Web: <http://www.cordis-suite.com>

E-mail: [support@cordis-suite.com](mailto:support@cordis-suite.com)

Office:

Brainport Industries Campus

BIC 1

5657 BX Eindhoven

The Netherlands

Tel: +31 (0)40 8517540

## Document change history

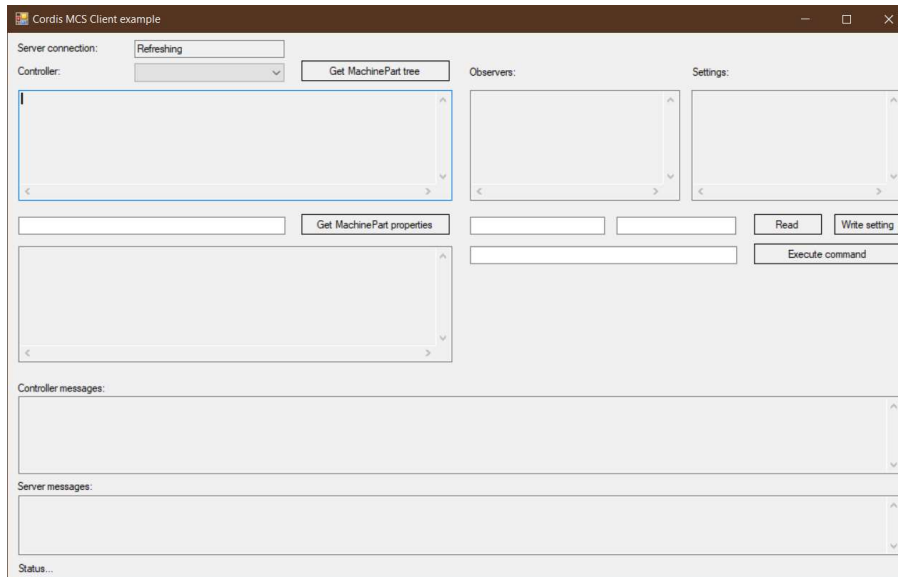
Version	Status	Date	Most significant changes	Author
0.1	Draft	2020-03-16	Initial document	RvE
0.2	Draft	2020-03-17	Reviewed / corrected	JBe
0.3	Draft	2020-03-18	Added statemachine history and plug-in commands	RvE

## Table of Contents

<b>Document change history</b>	<b>2</b>
<b>1 Demo application</b>	<b>5</b>
1.1 Get MachinePart tree	6
1.2 Get MachinePart properties	7
1.3 Read/Write Setting	8
1.4 Execute MachinePart Command	9
1.5 Read StateMachine history	10
1.6 Execute a Plug-In Command	11
1.7 Dashboard in parallel	13

## 1 Demo application

The following dialog is implemented in the Cordis.MCS.Client project, file TestForm.cs:



When the file ClientConfiguration.xml is filled in properly, and the test application restarted, the client application will connect to the Machine Control Server (MCS).

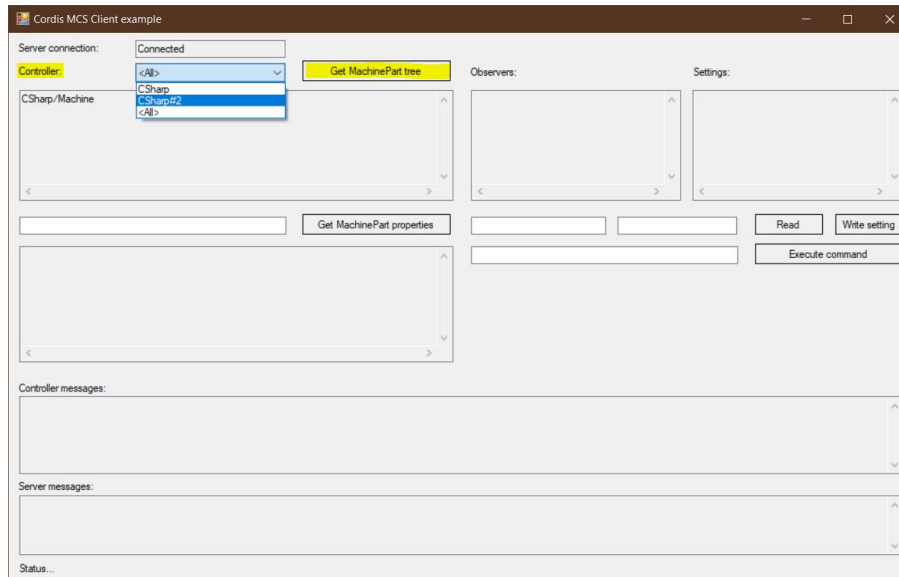
These are the default settings in ClientConfiguration.XML which correspond with the default MCS settings:

```
<?xml version="1.0" encoding="utf-8"?>
<ClientConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <CordisServerConfiguration>
    <Description>MachineControlServer</Description>
    <ServerAddress>http://127.0.0.1:80/JsonServices</ServerAddress>
    <SecurityToken>ADMIN</SecurityToken>
  </CordisServerConfiguration>
</ClientConfiguration>
```

In the drop-down box "Controller:" you can select (one of) the controller(s).

## 1.1 Get MachinePart tree

After selecting the proper controller, the button “Get MachinePart tree” will retrieve the complete tree of machineparts from the MCS.



For this feature the next methods are used in TestForm.cs:

```
private void btnGetMachinePartTree(object sender, EventArgs e)
private void OnGetMachinePartTree(MachinePartDto machinePart)
```

## 1.2 Get MachinePart properties

By selecting the full path of one of the machineparts and clicking “Get MachinePart properties”, the complete list of properties will be retrieved:

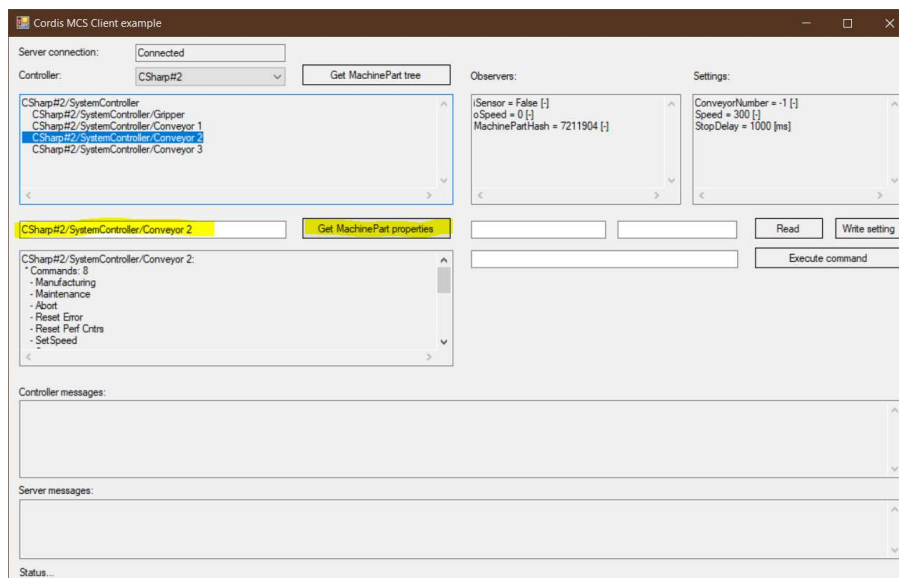
For this feature the next methods are used in TestForm.cs:

```
private void btnGetMpPropertiesClick(object sender, EventArgs e)
private void OnGetMachinePartProperties(
    MachinePartPropertiesDto machinePartProperties)
```

An overview of the content of all observers and settings is shown in the upper-right corner.

For this feature the next methods are used in TestForm.cs:

```
private void UpdateObservers(string mpFullPath)
private void OnGetObserverValues(IObservableList<ObserverDto> obj)
private void UpdateSettings(string machinePartFullPath)
private void OnGetSettingValues(
    IObservableList<SettingDto> controllerSettings)
```

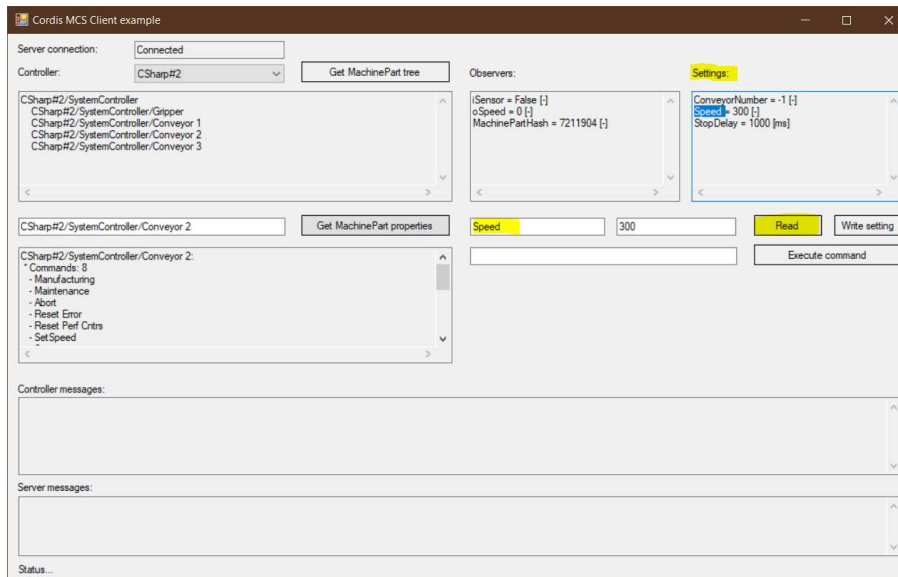


### 1.3 Read/Write Setting

By selecting the name of a setting, clicking “Read” below the “Settings:” pane, the specific setting is read from the controller

For this feature the next methods are used in TestForm.cs:

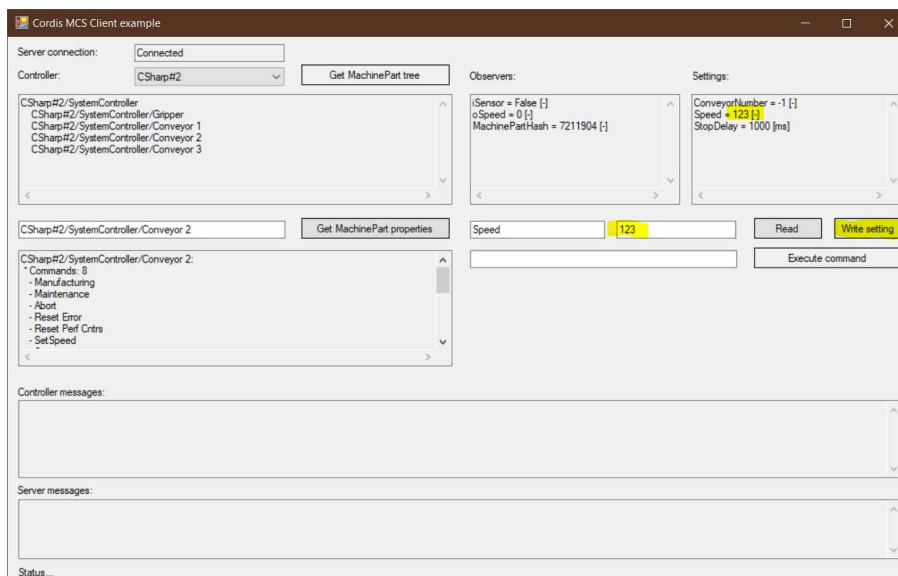
```
private void btnReadSingleSettingClick(object sender, EventArgs e)
private void OnGetSingleSetting(
    IObservableList<SettingDto> controllerSettings)
```



By modifying the read setting value, and clicking “Write setting” the setting is updated in the controller and the list of setting values is updated.

For this feature the next methods are used in TestForm.cs:

```
private void btnWriteSingleSettingClick(object sender, EventArgs e)
```



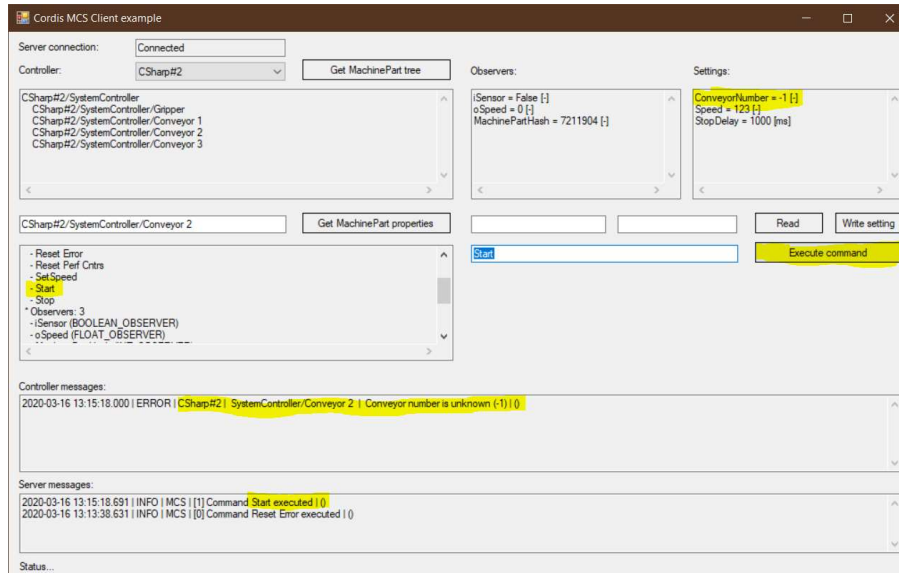


## 1.4 Execute MachinePart Command

Executing a MachinePart's command is achieved by selecting the command's name, and clicking "Execute command".

For this feature the next methods are used in TestForm.cs:

```
private void btnExeCommandClick(object sender, EventArgs e)
private void OnExecuteCommand(CommandKeyDto commandKeyDto)
```



In this case, the execution of the command is also shown as a server message in the "Server messages" pane. Because the Start command on the chosen machinepart results in an application error-message, this is also shown in the "Controller messages:" pane.

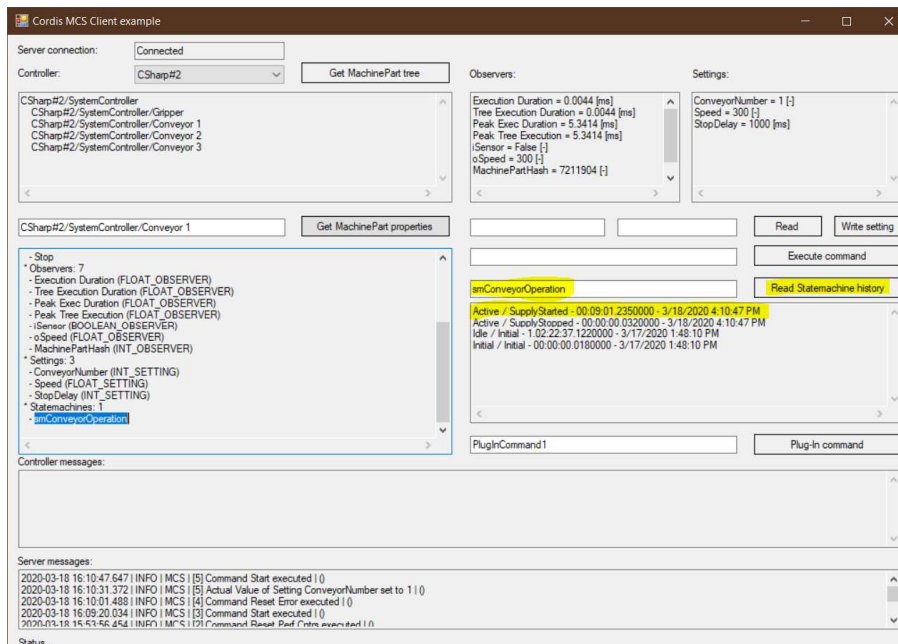
## 1.5 Read StateMachine history

Reading a statemachine's state history can be performed by selecting a statemachine's name in the machinepart properties overview, and clicking "Read StateMachine history" (provided the statemachine textbox in front of the button is empty. Otherwise edit the statemachine's name in the textbox before clicking the button.)

The pane below the button shows the history, newest (actual) state on top. The format is "State / SubState – duration of this (sub)state – timestamp of entering the (sub)state"

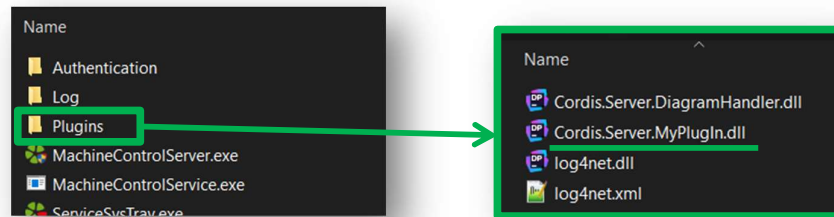
For this feature the next methods are used in TestForm.cs:

```
private void btnStateMachineHistoryClick(object sender, EventArgs e)
private void OnGetStateMachineEvents(
    IObservableList<StateMachineEventRecord>
    stateMachineEventRecords)
```



## 1.6 Execute a Plug-In Command

The MachineControlServer (MCS) supports a plug-in feature which can be used by placing a Cordis MCS Plug-In DLL in the subfolder “Plugins” of the MCS application.



During startup of the MCS, all plug-ins in this folder that do comply to the IPlugIn interface, are loaded by the server and initialized.

By developing a plug-in you can deploy functionality that runs in the context of the server, which might even run as a Windows service.

Together with the “MCS Client” example, we also provide an example called “MCS PlugIn”. Both are Visual Studio projects and can be used for customers to develop their own client and/or plug-in.

Sending commands to a MCS plug-in is also supported by the MCS C# client interface, this can be achieved by the following method:

```
void PerformPluginUseCase(
    string pluginName,
    string fullPath,
    string useCase,
    string parameters,
    string token,
    ...);
```

Despite the plug-in has no strict relation to a controller or machinepart whatsoever, the PerformPluginUseCase method does have a “fullPath” parameter which may be used in case the plug-in useCase needs this info.

Since the C# Client interface uses an asynchronous pattern, a response to a call on the interface is replied on a separate thread. This enable multiple calls to be performed in parallel, which will all be replied in separate callbacks.

However: in order to be able to determine which callback belongs to which request, the parameter “token” is provided. The token that is sent along with the request will also be replied by the accompanying callback, therefore it should be a unique token if the user wants to determine unambiguously which reply belongs to which original request.

The provided plug-in example has the following properties and commands:

```
public MyCordisServerPlugIn()
{
    Name = "MyCordisServerPlugIn";
    Description = "An example implementation of a Cordis Server Plugin.";
    Author = "Cordis Products B.V.";
    Version = "0.01";
}
```

```

FunctionDictionary = new Dictionary<string, IPluginCommand>
{
    ["PlugInCommand1"] = new PlugInCommand1(),
    ["PlugInCommand2"] = new PlugInCommand2(),
    ["PlugInCommand3"] = new PlugInCommand3()
};
}

```

Given the above definition, the Client example application demonstrates how to interact with a MCS plug-in. The text fields containing the plug-in command and the arguments are filled in corresponding to the plug-in definition above.

For this feature the next methods are used in TestForm.cs:

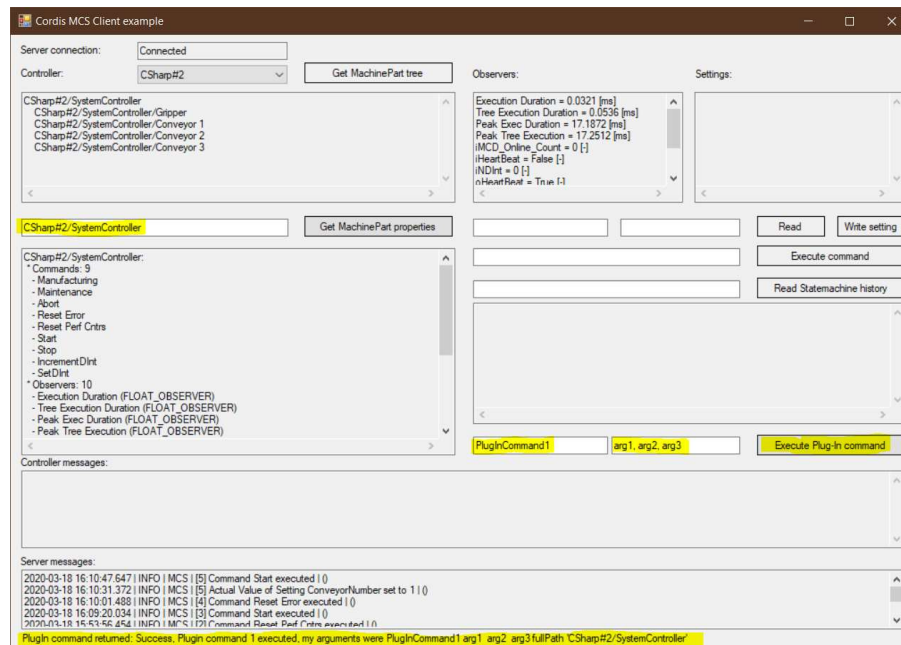
```

private void btnPlugInCommand(object sender, EventArgs e)
private void OnPerformPluginUseCase(string PlugInResponse)
private void PerformPluginUseCaseFaultHandler(IList<ExceptionInfo> faults)

```

The “PerformPluginUseCaseFaultHandler(…)” method is optional, and is used for catching exceptions in the plug-in command code.

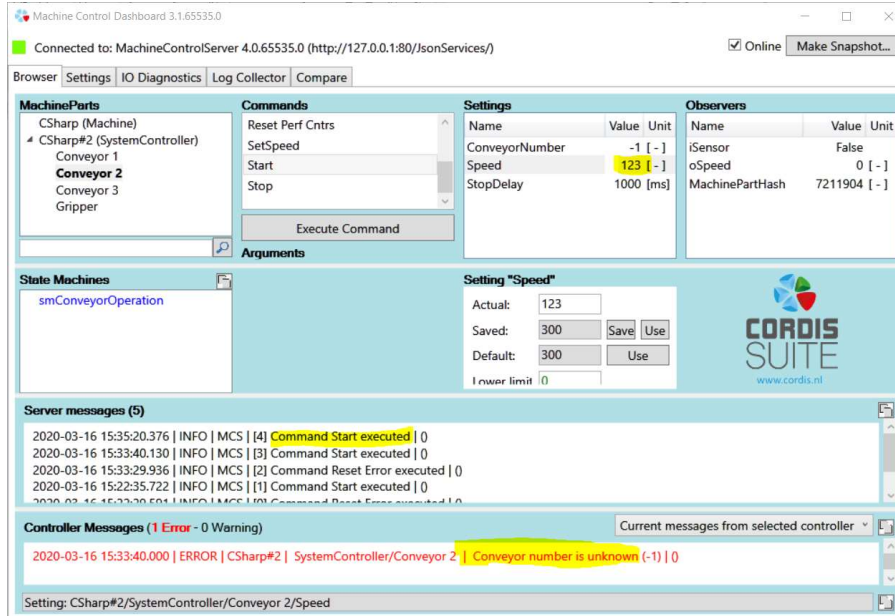
When this method is omitted, exceptions will by default be routed to the “HandleJsonFaultGlobal(…)” method of the client application. (see the [MdsiClientWrapper](#) class).



The example shows a selected machinepart because it's sent over the plug-in interface, but it's completely optional.

## 1.7 Dashboard in parallel

Changing settings, executing commands and the messages can be monitored in a simultaneously connected Cordis Machine Control Dashboard (and vice versa):



The screenshot displays the Cordis Machine Control Dashboard 3.1.65535.0 interface. It is connected to a MachineControlServer 4.0.65535.0 at http://127.0.0.1:80/JsonServices/. The interface includes several panels:

- MachineParts:** A tree view showing the hierarchy: CSharp (Machine) > CSharp#2 (SystemController) > Conveyor 1 > **Conveyor 2** > Conveyor 3 > Gripper.
- Commands:** A list of commands: Reset Perf Cntrs, SetSpeed, Start, and Stop. An "Execute Command" button is present.
- Settings:** A table with columns Name, Value, and Unit. It lists ConveyorNumber (-1 [-]), Speed (123 [-]), and StopDelay (1000 [ms]).
- Observers:** A table with columns Name, Value, and Unit. It lists iSensor (False), oSpeed (0 [-]), and MachinePartHash (7211904 [-]).
- State Machines:** A list containing smConveyorOperation.
- Setting "Speed":** A panel with input fields for Actual (123), Saved (300), and Default (300), along with Save and Use buttons. A Lower limit field is set to 0.
- Server messages (5):** A log of messages from the server, including INFO messages about command execution.
- Controller Messages (1 Error - 0 Warning):** A log of messages from the selected controller, showing an ERROR: "Conveyor number is unknown (-1) | 0".
- Setting:** A dropdown menu showing the current setting: CSharp#2/SystemController/Conveyor 2/Speed.