Registering a Regular Contribution as a Conformant Device

Universal Robots A/S

Version 1.15.0

Abstract

As of URCap API version 1.9, the ability to register a regular URCap as a conformant Polyscope device is introduced in PolyScope version 3.12.0/5.6.0. This document will be describe how to add a conformant gripper device to PolyScope.

Contents

1 Introduction

In PolyScope version 3.11.0/5.5.1, gripper driver concept was introduced (see the separate *Gripper Driver* document). This made it possible to easily add grippers to PolyScope with a simple, standardized UI. For gripper manufacturers with more advanced requirements for the UI, it is as of PolyScope version 3.12/5.6.0 possible to register their URCap program node contribution as a conformant PolyScope gripper device.

When PolyScope in the future introduces new device driver types, the intention is to allow for registering a regular URCap contribution as conformant with the new device type in a uniform manner. In this document, the registration as a gripper device will be used as a running example.

2 Why Register as a Device?

A template designer can choose to incorporate Gripper program nodes in his template program node. This will allow for advanced templates where it is easy for the end user to fully configure his program.

Gripper drivers are automatically registered in PolyScope, whereas regular URCaps that happen to be a gripper URCap will only be a regular URCap in the eyes of PolyScope. This means that it will not be accessible as a Gripper program node to the aforementioned template designer. To achieve this, it is now possible to register a regular URCap as being a conformant PolyScope gripper device.

3 Registering the Service

To inform PolyScope that a URCap is a conformant PolyScope gripper device, it is more specifically the implementation of the `SwingProgramNodeService` (or alternatively `ProgramNodeService`) interface that needs to be registered as a so called `GripperDevice` instance/type, together with the program node contribution to be used when inserting the program node for the gripper device.

This is done by first retrieving a `DeviceRegistrationManager` instance in the implementation of the `configureContribution(...)` method in the service. In this case, the `GripperRegistrationManager` interface is used to register the service as a PolyScope gripper device. In Listing 1 this is demonstrated for an implementation of the `SwingProgramNodeService` interface.

Listing 1: Registering a `SwingProgramNodeService` implementation as a PolyScope gripper

```
1  public class MyProgramNodeService implements SwingProgramNodeService<
       MyProgramNodeContribution, MyProgramNodeView> {
2
3    ...
4    @Override
5    public void configureContribution(ContributionConfiguration configuration) {
6      GripperRegistrationManager manager = configuration.
           getDeviceRegistrationManager(GripperRegistrationManager.class);
7      GripperRegistrationConfiguration registrationConfiguration = manager.
           registerAsGripper(MyProgramNodeContribution.class);
8    }
9    ...
10
11 }
```

To be able to register an HTML-based service as a PolyScope gripper device, the service must additionally implement the `ProgramNodeServiceConfigurable` interface.

4 Updating the Contribution

Besides implementing the `ProgramNodeContribtion` interface, the class specified as argument to the `registerAsGripper(Class)` method (in the `GripperRegistrationManager` interface) must additionally implement the `GripperConfigurable` interface to allow a template designer to retrieve and apply a configuration for the gripper. This means that the particular implementation of the `ProgramNodeContribtion` must be able to map between the PolyScope representation of a configuration for a Gripper program node and its own representation. The minimum requirement for a conformant gripper device is that it is able to perform basic grip and release actions. So the mapping of configurations must at least be able to support this.

When the `setConfig(GripperNodeConfig)` method is invoked, the supplied configuration must be applied (after having been adapted to the node's own representation) as if it were the end user who configured the Gripper program node. Going the other way, when the `getConfig()` method is invoked, a configuration must be returned reflecting what the Gripper program node is currently configured to. The type returned must be in the format dictated by the API. Use the supplied `GripperNodeConfigBuilders` to create the configuration to return. This is demonstrated in Listing 2.

Listing 2: Implementing the `GripperConfigurable` interface on a program node contribution

```
1  public class MyProgramNodeContribution implements ProgramNodeContribution,
       GripperConfigurable {
2    ...
3
4    @Override
5    public void setConfig(GripperNodeConfig config) {
6      switch (config.getConfigType()) {
7        case GRIP_ACTION:
8          myGripper.setCloseFingers();
9          break;
10       case RELEASE_ACTION:
11         myGripper.setOpenFingers();
12         break;
13       case UNDEFINED:
14         myGripper.clearGripperActionSelection();
15         break;
16     }
17   }
18
19   @Override
20   public GripperNodeConfig getConfig(GripperNodeConfigBuilders builders) {
21     if (myGripper.isClosingFingers()) {
22       return builders.createGripActionConfigBuilder().build();
23     } else if (myGripper.isOpeningFingers()){
24       return builders.createReleaseActionConfigBuilder().build();
25     } else {
26       return builders.createUndefinedConfig();
27     }
28   }
29   ...
30
31 }
```

5 Registering Capabilities

When registering the contribution as a gripper, a `GripperRegistrationConfiguration` is returned. This object allows the conformant gripper to retrieve the `GripperCapabilities` object where registration of capabilities is possible. This informs PolyScope and other URCaps about the capabilities of the gripper. This can for instance be used when building a template and preconfiguring it to help the end user.

All capabilities described in the separate *Gripper Driver* document can be registered. However as of version 1.11.0 of the URCap API, a template user can only build configurations containing information about gripper action (grip or release) as well as a selected gripper if the gripper supports this capability. There is no harm in registering all supported capabilities as this will carry over to when the functionality is supported for building the configurations of the grippers.

An example of registering the multi-gripper capability is shown in listing 4. Notice how the gripper list is provided as described in the previously linked document.

Listing 3: Registering the multi-gripper capability

```
1  public class MyProgramNodeService implements SwingProgramNodeService<
       MyProgramNodeContribution, MyProgramNodeView> {
2
3    ...
4    @Override
5    public void configureContribution(ContributionConfiguration configuration) {
6      GripperRegistrationManager manager = configuration.
           getDeviceRegistrationManager(GripperRegistrationManager.class);
7      GripperRegistrationConfiguration registrationConfiguration = manager.
           registerAsGripper(MyProgramNodeContribution.class);
8
9      GripperCapabilities gripperCapabilities = registrationConfiguration.
           getGripperCapabilities();
10
11     gripperCapabilities.registerMultiGripperCapability(
12         new GripperListProvider() {
13           @Override
14           public GripperList getGripperList(GripperListBuilder
                 gripperListBuilder, Locale locale) {
15             gripperListBuilder.createGripper("ID_A", "A", true);
16             gripperListBuilder.createGripper("ID_B", "B", true);
17             return gripperListBuilder.buildList();
18           }
19         });
20
21   }
22   ...
23
24 }
```

Since a conformant gripper is a normal URCap offering gripper functionality it usually consists of both a `ProgramNodeService` and an `InstallationNodeService`. If a need arises to share either the created grippers or the registered capabilities, these can be shared via a common context object. This object can hold whatever is necessary and passed as a constructor argument to both aforementioned services. When the program node service then registers a capability, it sets the created objects on the common context object so the installation node service can also work with them. This way the installation could for instance offer functionality to enable/disable grippers even though these are created in the program node service.

6 Registering Capabilities

When registering the contribution as a gripper, a `GripperRegistrationConfiguration` instance is returned. This object allows the conformant gripper to access the `GripperCapabilities` interface where registration of capabilities is possible. Registering capabilities informs PolyScope and other URCaps about the capabilities of the gripper. This can, for instance, be used when building a template node to help the end user by preconfiguring the value of the capability parameters for Gripper nodes.

All capabilities described in the separate *Gripper Driver* document can be registered. However, as of version 1.11.0 of the URCap API, a template designer can only build Gripper node configurations with preconfigured gripper action (grip or release) as well as gripper selection, if the gripper supports the multi-gripper capability. There is no harm in registering all supported capabilities as this will carry over to when the functionality is supported for building the configurations.

An example of registering the multi-gripper capability is shown in listing 4. Notice how the gripper list is provided as described in the separate *Gripper Driver* document.

Listing 4: Registering the multi-gripper capability

```
1  public class MyProgramNodeService implements SwingProgramNodeService<
       MyProgramNodeContribution, MyProgramNodeView> {
2
3    ...
4    @Override
5    public void configureContribution(ContributionConfiguration configuration) {
6      GripperRegistrationManager manager = configuration.
           getDeviceRegistrationManager(GripperRegistrationManager.class);
7      GripperRegistrationConfiguration registrationConfiguration = manager.
           registerAsGripper(MyProgramNodeContribution.class);
8
9      GripperCapabilities gripperCapabilities = registrationConfiguration.
           getGripperCapabilities();
10
11     gripperCapabilities.registerMultiGripperCapability(
12         new GripperListProvider() {
13           @Override
14           public GripperList getGripperList(GripperListBuilder
                 gripperListBuilder, Locale locale) {
15             gripperListBuilder.createGripper("ID_A", "A", true);
16             gripperListBuilder.createGripper("ID_B", "B", true);
17             return gripperListBuilder.buildList();
18           }
19         });
20
21   }
22   ...
23
24 }
```

6.1 Sharing Registered Capabilities

Since a conformant gripper is a regular URCap offering gripper functionality, it usually contains both a program node and an installation node contribution with corresponding implementations of the `ProgramNodeService` and the `InstallationNodeService` interfaces. If a need arises to share either the created individual grippers (only related to the multi-gripper capability) or the registered capability instances (which is returned after registration, e.g. an instance of the `MultiGripperCapability` interface), these could be shared via a custom, common context object.

This object can hold whatever is necessary and can be passed as an argument to the contribution constructor when the `createInstallationNode(...)` method and the `createNode(...)` method are called in the aforementioned services. When the program node service registers a capability, it can set the relevant created objects on the common context object, so that the installation node contribution can also access them. This way the installation node could, for instance, offer functionality to enable/disable grippers (through a UI) even though these are created by the program node service.

This is demonstrated in Listings 5, 6, 7, 8, 9 and 10. Listing 5 shows a class, `MyGripperContext`, which is used for sharing the registered multi-gripper capability and the two added individual grippers with the program node and installation node contributions. An implementation of an activator which passes an instance of this class to the program node and installation node service is shown in Listing 6.

Listing 5: Example of a class for sharing a registered capability

```
1  // Class for sharing the registered multi-gripper capability and
2  // the individual grippers
3  public class MyGripperContext {
4    MultiGripperCapability capability;
5
6    public void setCapability(MultiGripperCapability capability) {
7      this.capability = capability;
8    }
9
10   public MultiGripperCapability getCapability() {
11     return capability;
12   }
13
14   public SelectableGripper getGripperA() {
15     return capability.getSelectableGrippers().get(0);
16   }
17
18   public SelectableGripper getGripperB() {
19     return capability.getSelectableGrippers().get(1);
20   }
21 }
```

Listing 6: Passing shared object in the Activator

```
1  public class Activator implements BundleActivator {
2    @Override
3    public void start(final BundleContext context) {
4      // Instantiate shared context object holding the multi-gripper
5      // capability
6      MyGripperContext myGripperContext = new MyGripperContext();
7
8      context.registerService(SwingProgramNodeService.class, new
           MyProgramNodeService(myGripperContext),null);
9      context.registerService(SwingInstallationNodeService.class, new
           MyInstallationNodeService(myGripperContext),null);
10   }
11   ...
12 }
```

In Listing 7 the program node service sets the instance of the multi-gripper capability on the `MyGripperContext` instance. Each of the individual grippers are passed to the contribution constructor in the `createNode(...)` method, so that the program node contribution can use them when accepting and returning a Gripper node configuration (see section 6.2. Accepting and Returning Configurations with Capabilities for more details). Listing 8 shows how the `MyGripperContext`

instance is passed to the contribution constructor in the method `createInstallationNode(...)` in the installation node service.

Listing 7: Using shared object in program node service

```
1  public class MyProgramNodeService implements SwingProgramNodeService<
       MyProgramNodeContribution, MyProgramNodeView> {
2
3    private MyGripperContext myGripperContext;
4
5    public MyProgramNodeService(MyGripperContext myGripperContext) {
6      this.myGripperContext = myGripperContext;
7    }
8    ...
9
10   @Override
11   public void configureContribution(ContributionConfiguration configuration) {
12     ...
13     MultiGripperCapability multiGripperCapability = gripperCapabilities.
           registerMultiGripperCapability(
14         new GripperListProvider() {
15           @Override
16           public GripperList getGripperList(GripperListBuilder
                 gripperListBuilder, Locale locale) {
17             SelectableGripper gripperA = gripperListBuilder.createGripper("
                   ID_A", "A", true);
18             myGripperContext.setGripperA(gripperA);
19
20             SelectableGripper gripperB = gripperListBuilder.createGripper("
                   ID_B", "B", true);
21             myGripperContext.setGripperB(gripperB);
22
23             return gripperListBuilder.buildList();
24           }
25         });
26     myGripperContext.setCapability(multiGripperCapability);
27   }
28
29   ...
30
31   @Override
32   public MyProgramNodeContribution createNode(ProgramAPIProvider apiProvider,
         MyProgramNodeView view, DataModel model, CreationContext context) {
33     // Pass the two individual grippers of the multi-gripper to the program
           node
34     return new MyProgramNodeContribution(myGripperContext.getGripperA(),
           myGripperContext.getGripperB());
35   }
36 }
```

Listing 8: Using shared object in installation node service

```
1  public class MyInstallationNodeService implements SwingInstallationNodeService
       <MyInstallationNodeContribution, MyInstallationNodeView> {
2
3    private MyGripperContext myGripperContext;
4
5    public MyInstallationNodeService(MyGripperContext myGripperContext) {
6      this.myGripperContext = myGripperContext;
7    }
8    ...
9
10   @Override
11   public MyInstallationNodeContribution createInstallationNode(
         InstallationAPIProvider apiProvider, MyInstallationNodeView view,
         DataModel model, CreationContext context) {
```

```
12      return new MyInstallationNodeContribution(myGripperContext);
13    }
14  }
```

In Listing 9 the multi-gripper capability is accessed via the `MyGripperContext` instance in the installation node and used to implement functionality for enabling and disabling the individual grippers. This functionality is used in the code in Listing 10 where the installation node view enables or disables the second gripper based on whether the end user has selected to use a single (single mounting) or multi-gripper (dual mounting) setup.

Listing 9: Using shared object in installation node

```
1  public class MyInstallationNodeContribution implements
       InstallationNodeContribution {
2    private MyGripperContext myGripperContext;
3
4    public MyInstallationNodeContribution(MyGripperContext myGripperContext) {
5      this.myGripperContext = myGripperContext;
6    }
7    ...
8
9    public void setSingleMountingSelected() {
10     MultiGripperCapability multiGripperCapability = myGripperContext.
           getCapability();
11     multiGripperCapability.setEnabled(myGripperContext.getGripperB(), false);
12   }
13
14   public void setDualMountingSelected() {
15     MultiGripperCapability multiGripperCapability = myGripperContext.
           getCapability();
16     multiGripperCapability.setEnabled(myGripperContext.getGripperB(), true);
17   }
18 }
```

Listing 10: Controlling gripper enablement from installation node view

```
1  public class MyInstallationNodeView implements SwingInstallationNodeView<
       MyInstallationNodeContribution> {
2    ...
3    @Override
4    public void buildUI(JPanel panel, final MyInstallationNodeContribution
         installationContribution) {
5      ...
6      mountingSelector.addActionListener(new ActionListener() {
7        @Override
8        public void actionPerformed(ActionEvent e) {
9          MountingType selectedMountingType = (MountingType)mountingSelector.
               getSelectedItem();
10
11         if (selectedMountingType == MountingType.SINGLE) {
12           installationContribution.setSingleMountingSelected();
13         } else if (selectedMountingType == MountingType.DUAL) {
14           installationContribution.setDualMountingSelected();
15         }
16       }
17     });
18     ...
19   }
20 }
```

6.2 Accepting and Returning Configurations with Capabilities

When a capability has been registered, the program node contribution must also take the capability into account when accepting and returning a Gripper node configuration.

This is demonstrated for the multi-gripper capability in Listing 11 which is an extension of the example in Listing 2. The code maps between the PolyScope representation of the individual grippers (the `SelectableGripper` interface) and the URCaps's own internal representation when accepting and returning a configuration.

Listing 11: Using the multi-gripper capability in the program contribution

```
1  public class MyProgramNodeContribution implements ProgramNodeContribution,
       GripperConfigurable {
2    ...
3    private SelectableGripper gripperA;
4    private SelectableGripper gripperB;
5
6    public MyProgramNodeContribution(SelectableGripper gripperA,
         SelectableGripper gripperB) {
7      this.gripperA = gripperA;
8      this.gripperB = gripperB;
9    }
10
11   ...
12
13   @Override
14   public void setConfig(GripperNodeConfig config) {
15     switch (config.getConfigType()) {
16       case GRIP_ACTION:
17         myGripper.setCloseFingers();
18
19         Optional<SelectableGripper> gripperSelection = ((GripActionConfig)
               config).getGripperSelection();
20         if (gripperSelection.isPresent()) {
21           SelectableGripper selectedGripper = gripperSelection.get();
22           if (selectedGripper.equals(gripperA)) {
23             myGripper.selectStandardGripper();
24           } else if (selectedGripper.equals(gripperB)) {
25             myGripper.selectSecondaryGripper();
26           }
27         } else {
28           myGripper.clearGripperSelection();
29         }
30         break;
31       case RELEASE_ACTION:
32       ...
33     }
34   }
35
36   @Override
37   public GripperNodeConfig getConfig(GripperNodeConfigBuilders builders) {
38     if (myGripper.isClosingFingers()) {
39       GripActionConfigBuilder configBuilder = builders.
             createGripActionConfigBuilder();
40       if (myGripper.isStandardGripperSelected()) {
41         configBuilder.setGripperSelection(gripperA);
42       } else if (myGripper.isSecondaryGripperSelected()) {
43         configBuilder.setGripperSelection(gripperB);
44       }
45       return configBuilder.build();
46     } else if (myGripper.isOpeningFingers()){
47       ...
48     } else {
49       return builders.createUndefinedConfig();
```

```
50        }
51      }
52      ...
53
54  }
```