URCap Software Development Tutorial HTML

Universal Robots A/S

Version 1.13.0

Abstract

URCaps make it possible to seamlessly extend any Universal Robot with customized functionality. Using the URCap Software Platform, a URCap developer can define customized installation screens and program nodes for the end user. These can, for example, encapsulate new complex robot programming concepts, or provide friendly hardware configuration interfaces.

This tutorial explains how to use the URCap Software Platform version 1.13.0 to develop and deploy URCaps with HTML-based user interfaces for PolyScope version 5.11.0 running on e-Series robots. Older PolyScope versions can be targeted using older versions of the URCap Software Platform. URCaps can be developed for CB3 robots with PolyScope version 3.3.x to 3.15.x using version 1.12.0 or older of the URCap Software Platform.

Contents

1 Introduction **3**
1.1 Features in URCap Software Platform 1.13.0 . . . 3

2 Prerequisites **5**

3 URCap SDK **6**

4 Building and deploying URCaps **8**
4.1 Building . . . 8
4.2 Manual Deployment using PolyScope . . . 8

5 Structure of a URCap Project **13**

6 Deployment with Maven **16**

7 Contribution of an Installation Node **17**
7.1 Layout of the Installation Node . . . 17
7.2 Making the customized Installation Node available to PolyScope . . . 18
7.3 Functionality of the Installation Node . . . 19

8 Contribution of a Program Node **21**
8.1 Layout of the Program Node . . . 21
8.2 Making the customized Program Nodes available to PolyScope . . . 22
8.3 Functionality of the Program Node . . . 23
8.4 Updating the data model . . . 25
8.4.1 Restrictions . . . 25
8.4.2 Undo/redo Functionality . . . 25
8.5 Loading Programs with Program Node Contributions . . . 26

9 Contribution of a Daemon **27**
9.1 Daemon Service . . . 27
9.2 Interaction with the Daemon . . . 28
9.3 C/C++ Daemon Executables . . . 30
9.4 Tying the different Contributions together . . . 31

10 URCap Examples Overview **33**
10.1 Regular Samples . . . 33
10.2 Driver Samples . . . 36

11 Creating new thin Projects using a Maven Archetype **41**

12 Compatibility **44**
12.1 Robot Series Compatibility . . . 44
12.1.1 Compatibility Flags . . . 44
12.1.2 Installation and Startup of URCaps . . . 45
12.1.3 Updating Existing URCaps . . . 45
12.2 API Compatibility . . . 46
12.2.1 Advanced compatibility . . . 47

13 Exception Handling **49**

14 Troubleshooting **50**

A URCaps and Generated Script Code **52**

B CSS and HTML Support **53**

C My Daemon Program and Installation Node **54**

1 Introduction

The first official version of the URCap Software Platform (version 1.0.0) was released with PolyScope version 3.3.0 running on CB3 robots. This tutorial describes features supported in version 1.13.0 of the URCap Software Platform which is released together with PolyScope version 5.11.0 for e-Series robots. To create URCaps for PolyScope version 3.3.x to 3.15.x running on CB3 robots, older versions of the URCap Software Platform must be used.

This platform can be used to develop external contributions to PolyScope that are loaded when PolyScope starts up. This makes it possible for a URCap developer to provide customized functionality within PolyScope.

For example, a customized installation screen can serve the end user to comfortably configure a new tool. Similarly, a customized program node may serve as a way to perform complex tasks while hiding unnecessary detail.

The layout of a customized screen is defined using a subset of HTML and CSS. The behaviour of a customized node, data persistence and script code generation is implemented in Java. The URCap along with its resources is packaged and distributed as a single Java Jar file with the `.urcap` file extension. A URCap can be installed from the URCap Settings/Setup screen in PolyScope.

The tutorial is organized in the following manner:

- Section 2. Prerequisites and 3. URCap SDK explain what you need to start developing URCaps.
- Section 4. Building and deploying URCaps to 6. Deployment with Maven guide you through the basic project setup including build and deployment.
- Section 7. Contribution of an Installation Node to 9. Contribution of a Daemon introduces the concept behind URCaps and explains the different software components.
- Section 10. URCap Examples Overview provides an overview of technical URCap examples distributed with the SDK that focus on specific features of the URCap API.
- Section 11. Creating new thin Projects using a Maven Archetype demonstrates how to create an empty URCap project. We recommend that you also have a look at the examples when you want to start from scratch.
- Section 14. Troubleshooting describes different debugging and troubleshooting options. Also visit the support forum at `www.universal-robots.com/plus`.

To get started we use the *Hello World* and the *My Daemon* URCaps as running examples. These are very simple and basic URCaps.

1.1 Features in URCap Software Platform 1.13.0

The following entities can be contributed to PolyScope using a URCap:

- Customized installation nodes and corresponding screens with text, images, and input widgets (e.g. a text field).
- Customized program nodes and corresponding screens with text, images, and input widgets.
- Daemon executables that run as separate background processes on the control box.

The customized installation nodes support:

- Saving and loading of data underlying the customized installation node as part of the currently loaded installation in PolyScope.
- Script code that an installation node contributes to the preamble of a robot program.
- Behaviour for widgets and other elements on the customized screens.

The customized program nodes support:

- Saving and loading of data underlying the customized program nodes as part of the currently loaded PolyScope program.
- Script code that a program node contributes to the script code generated by the robot program.
- Behaviour for widgets and other elements on the customized screens.

2 Prerequisites

A working version of Java SDK 6 is required for URCap development along with Apache Maven 3.0.5. You will also need PolyScope version 5.11.0 in order to install the developed URCap, if it is using URCap API version 1.13.0. Previous versions of the API will have lower requirements for the PolyScope version. Any Universal Robots robot (e.g., a UR3, UR5, UR10, etc.) can be used for that purpose or the Universal Robots offline simulator software (URSim). PolyScope and the offline simulator can be found in the download area of the tech support website at:

`www.universal-robots.com/support`

Select the applicable version and follow the given installation instructions. The offline simulator is available for Linux and non-Linux operating systems through a virtual Linux machine.

The script language and pre-defined script functions are defined in the script manual, which can also be found in the download area of the tech support website.

The URCap SDK is freely available on the Universal Robots+ website at:

`https://plus.universal-robots.com`

It includes the sources for the URCap examples.

The *My Daemon* example of this tutorial additionally requires either Python 2.5 (compatible) or the Universal Robots urtool3 cross-compiler toolchain. The urtool3 cross-compiler is included in the SDK.

The following section 3. URCap SDK describes the content of the URCap SDK.

3 URCap SDK

The URCap SDK provides the basics to create a URCap. It contains a Java package with the API that the developer will program against, documentation, the Hello World and other URCap examples, the urtool3 cross-compiler toolchain and means of easily creating a new empty Maven-based template URCap project (See section 11. Creating new thin Projects using a Maven Archetype) as well as easily updating an existing URCap.

```
com.ur.urcap.sdk
├── artifacts
│   ├── api
│   │   ├── ⋮
│   │   └── 1.13.0
│   │       ├── com.ur.urcap.api-1.13.0.jar
│   │       ├── com.ur.urcap.api-1.13.0-javadoc.jar
│   │       └── com.ur.urcap.api-1.13.0-sources.jar
│   ├── archetype
│   │   └── com.ur.urcap.archetype-1.13.0.jar
│   └── other
│       ├── commons-httpclient-3.1.0.0.jar
│       ├── ws-commons-util-1.0.2.0.jar
│       ├── xmlrpc-client-3.1.3.0.jar
│       └── xmlrpc-common-3.1.3.0.jar
├── doc
│   ├── urcap_tutorial_html.pdf
│   ├── urcap_tutorial_swing.pdf
│   ├── program_node_configuration.pdf
│   ├── working_with_variables.pdf
│   └── ⋮
├── samples
│   ├── driver
│   │   └── ⋮
│   ├── html
│   │   ├── com.ur.urcap.examples.helloworld
│   │   │   └── ⋮ (See Figure 2, page 13)
│   │   ├── com.ur.urcap.examples.mydaemon
│   │   │   └── ⋮ (See Figure 3, page 14)
│   │   └── ⋮
│   └── swing
│       └── ⋮
├── urtool
│   └── urtool3_0.3_amd64.deb
├── install.sh
├── newURCap.sh
├── readme.txt
└── upgradeURCap.sh
```

Figure 1: File structure of the URCaps SDK

The URCap SDK is distributed as a single ZIP file. Figure 1 shows the structure of the file.

A description of the directories and files contained in this file is given below:

`/artifacts/`: This directory holds all released versions of the URCap API in separate folders, the Maven archetype and other necessary files. Each folder named with a version number holds the Java packages, (e.g. `com.ur.urcap.api-1.13.0*.jar` files), that contains Java interfaces, Javadoc and sources of the URCap API that are necessary to implement the Java portion of a URCap. The Maven archetype folder holds the `com.ur.urcap.archetype-1.13.0.jar` file, that can be used to create a new empty template URCap project.

`/doc/`: The directory contains this tutorial (both in a HTML-based version and a Swing-based version) as well as a document describing how to configure child program nodes in a sub-tree and a document explaining how to work with variables. Included is also a document with a guide on how to convert an existing URCap with HTML-based user interface to a Swing-based one.

`/samples/`: A folder containing example projects demonstrating different features of the software framework. A description of the examples is found in section 10. URCap Examples Overview.

`/urtool/`: Contains the urtool3 cross-compiler toolchain that should be used when building C/C++ daemon executables for the CB3.0/3.1 and CB5.0 control boxes.

`install.sh`: A script which should be run as a first step to install the URCap SDK and urtool3 cross-compiler toolchain (see section 4. Building and deploying URCaps). This will install all released versions of the URCap API and the Maven archetype in your local Maven repository as well as the cross-compiler toolchain in `/opt/urtool-3.0` (should you choose so).

`newURCap.sh`: A script which can be used to create a new empty Maven-based template URCap project in the current working directory (see section 11. Creating new thin Projects using a Maven Archetype).

`upgradeURCap.sh`: A script which can be used to update an existing URCap (project) to specify which robot series the URCap is compatible with (see section 12.1. Robot Series Compatibility).

`readme.txt`: A *readme* file describing the content of the SDK.

4 Building and deploying URCaps

4.1 Building

To get started unzip the SDK zip file to a suitable location and run the install script inside the target location:

```
$ ./install.sh
```

This installs the SDK on your machine.

Next, enter the `samples/html/com.ur.urcap.examples.helloworld` directory and compile the example by:

```
$ cd samples/html/com.ur.urcap.examples.helloworld
$ mvn install
```

A new URCap with file name `target/helloworld-1.0-SNAPSHOT.urcap` has been born!

A similar procedure should be followed to compile the other URCap examples.

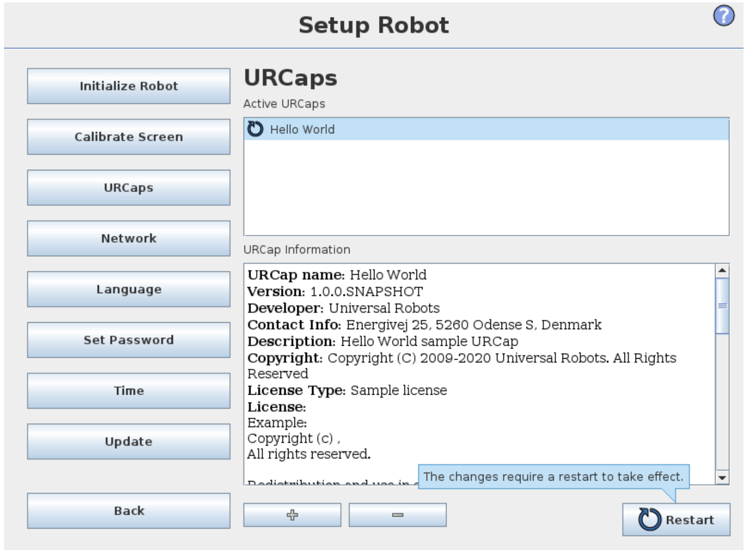
4.2 Manual Deployment using PolyScope

The URCap can be added to PolyScope with these steps:

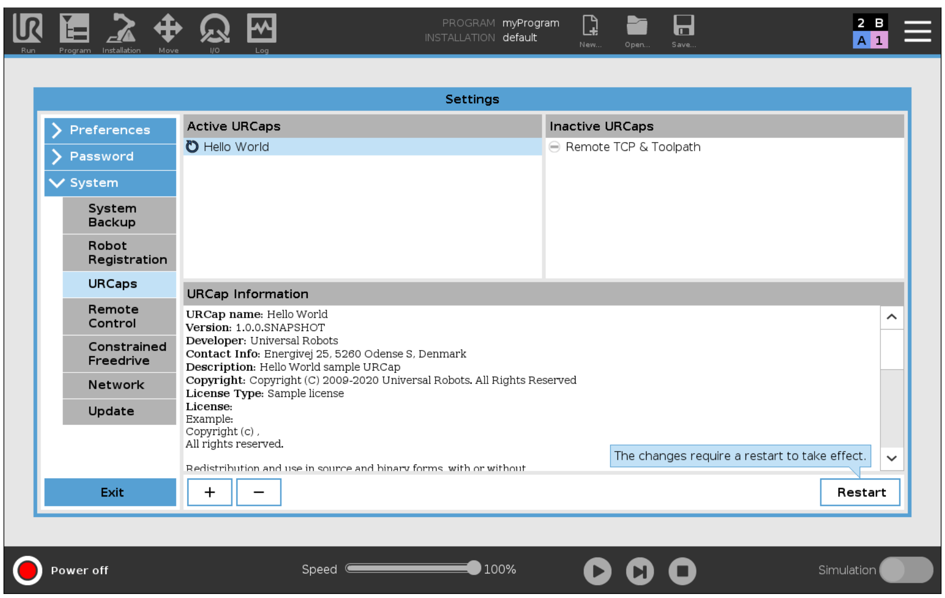
1. Copy the `helloworld-1.0-SNAPSHOT.urcap` file from above to your `programs` directory used by PolyScope or to a USB stick and insert it into a robot.
2. **CB3 robot:** Tap the Setup Robot button from the PolyScope Robot User Interface Screen (the Welcome screen)

   e-Series robot: In the right corner of the Header, tap the Hamburger menu and select Settings
3. **CB3 Robot:** Tap the URCaps button from the Setup Robot screen.

   e-Series robot: In the Settings screen under System, select URCaps.
4. Tap the + button.
5. Select a `.urcap` file, e.g. `helloworld-1.0-SNAPSHOT.urcap` and tap the Open button.
6. Restart PolyScope using the Restart button in the bottom of the screen:

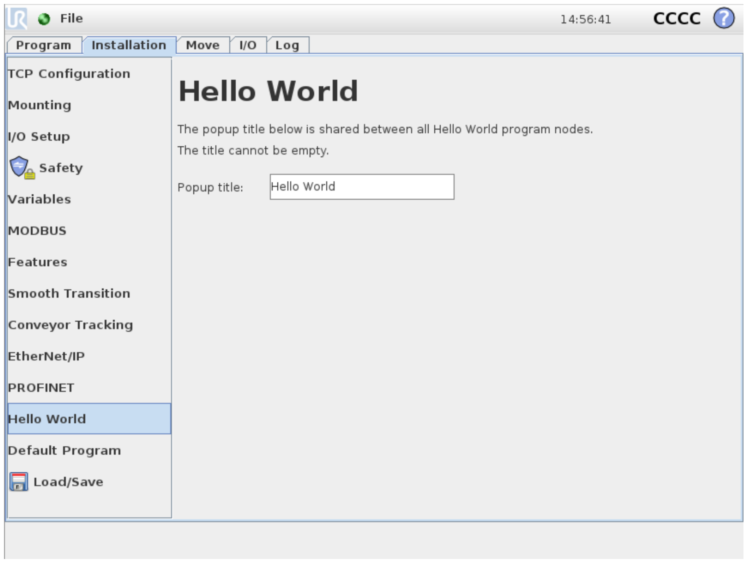


CB3 robot

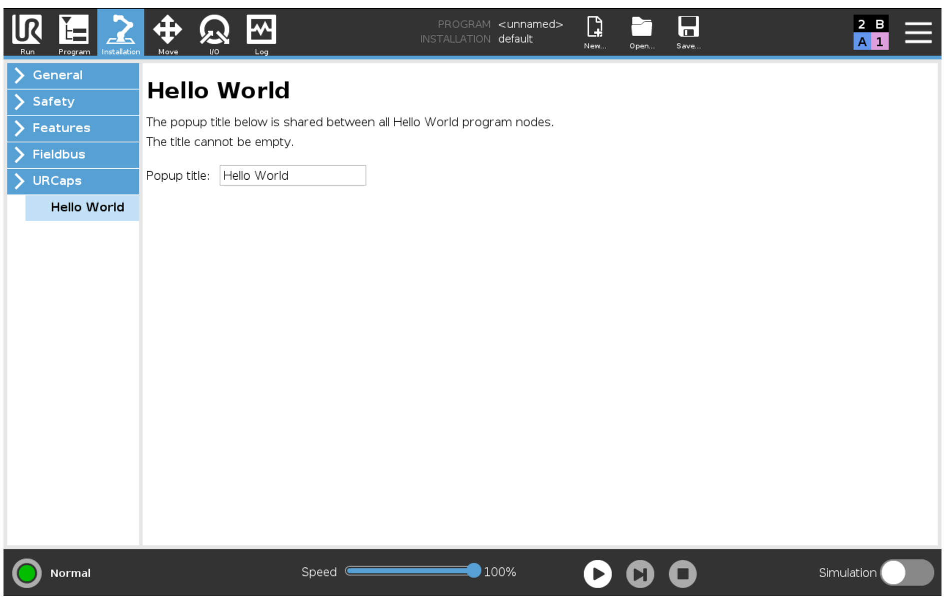


e-Series robot

When the Hello World URCap is deployed, the following installation screen is accessible from the Installation tab:

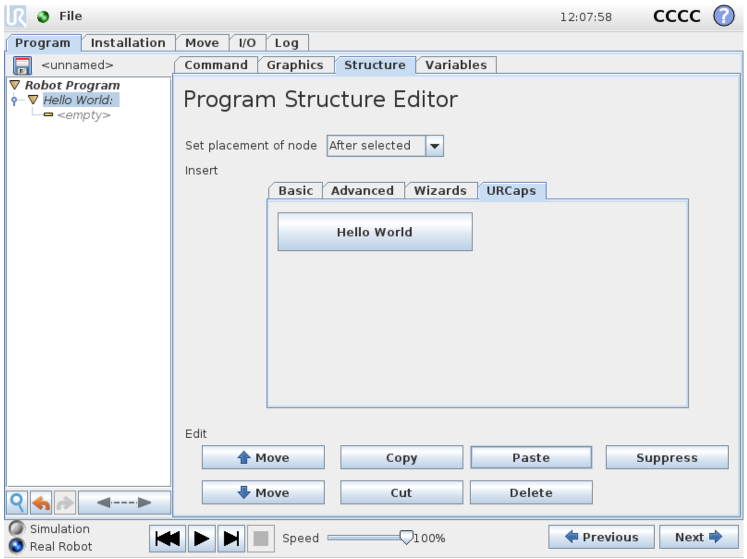


CB3 robot



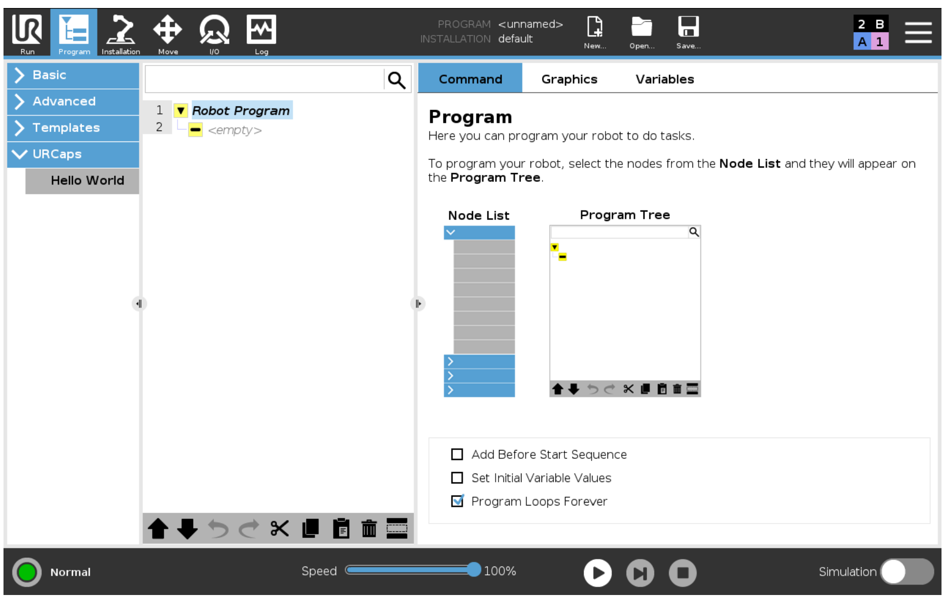
e-Series robot

Furthermore, the Hello World program node is available in the Program tab. On CB3 robots, the node is visible within the Structure tab after selecting the URCaps subtab:



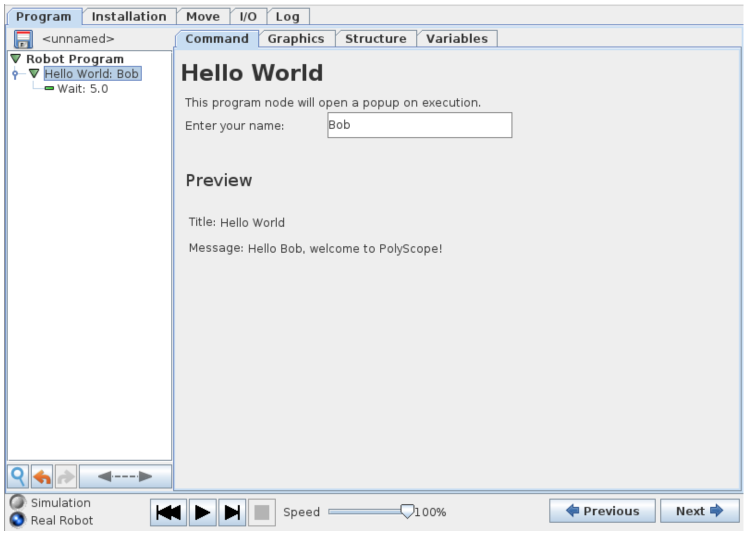
CB3 robot

On e-Series robots, the node is visible on the left of the screen after tapping URCaps:

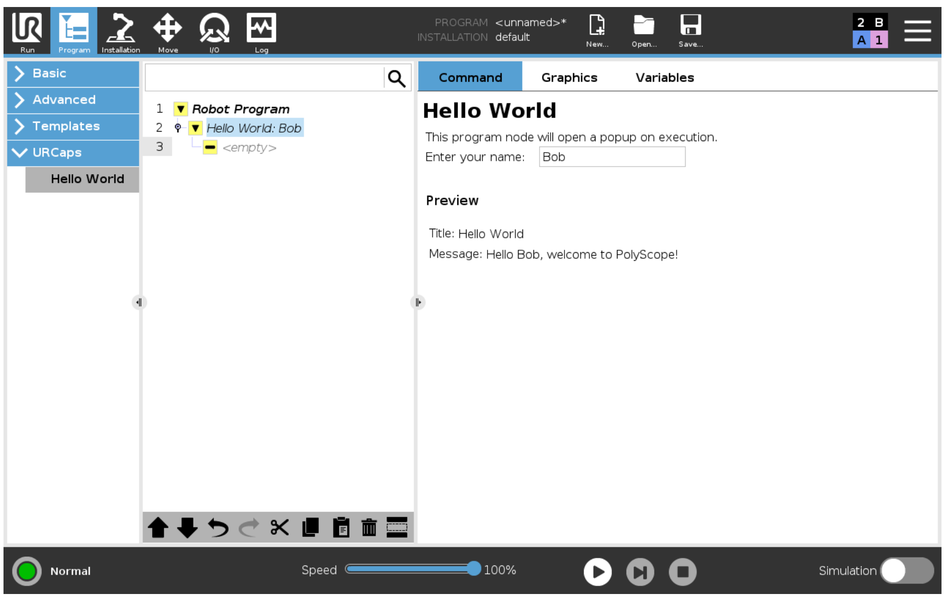


e-Series robot

The screen for the program node looks as follows:



CB3 robot



e-Series robot

When the program displayed above runs, a pop-up is shown with the title "Hello World" (configured in the installation screen) and message "Hello Bob, welcome to PolyScope!" (using the name defined in the program node).

5 Structure of a URCap Project

A URCap is a Java Archive (`.jar`) file with the `.urcap` file extension. The Java file may contain a number of new installation nodes, program nodes, and daemon executables.

```
com.ur.urcap.examples.helloworld
├── pom.xml
├── assembly.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── ur
        │           └── urcap
        │               └── examples
        │                   └── helloworld
        │                       └── impl
        │                           ├── Activator.java
        │                           ├── HelloWorldInstallationNodeService.java
        │                           ├── HelloWorldInstallationNodeContribution.java
        │                           ├── HelloWorldProgramNodeService.java
        │                           └── HelloWorldProgramNodeContribution.java
        └── resources
            ├── com
            │   └── ur
            │       └── urcap
            │           └── examples
            │               └── helloworld
            │                   └── impl
            │                       ├── installation.html
            │                       └── programnode.html
            └── META-INF
                └── LICENSE
```

Figure 2: Structure of the Hello World URCap project

Figure 2 shows the structure of the Hello World URCap project. This project consists of the following parts:

1. A *view* part consisting of two screens with the layout specified in the files `installation.html` and `programnode.html`.
2. A Java *implementation* for the screens above, namely:
   (a) `HelloWorldInstallationNodeService.java` and `HelloWorldInstallationNodeContribution.java`
   (b) `HelloWorldProgramNodeService.java` and `HelloWorldProgramNodeContribution.java`
3. A *license* `META-INF/LICENSE` with the license information that are shown to the user when the URCap is installed.
4. Maven configuration files `pom.xml` and `assembly.xml` for building the project.

The My Daemon URCap is an extended version of the Hello World URCap, that exemplifies the integration of an external daemon process. Figure 3 shows the structure of the My Daemon URCap project. Compared to the Hello World project it additionally offers the following parts:

1. A Python 2.5 *daemon* executable in the file `hello-world.py`.
2. C++ *daemon* sources in directory `daemon`.
3. A Java *implementation* `MyDaemonDaemonService.java` that defines and installs a daemon and makes it possible to control the daemon.

The Python and C++ daemons are alternatives that provide the same functionality.

```
com.ur.urcap.examples.mydaemon
├── pom.xml
├── assembly.xml
├── daemon
│   └── ⋮ (See Figure 6, page 31)
└── src
    └── main
        ├── java
        │   └── com
        │       └── ur
        │           └── urcap
        │               └── examples
        │                   └── mydaemon
        │                       └── impl
        │                           ├── Activator.java
        │                           ├── MyDaemonDaemonService.java
        │                           ├── MyDaemonInstallationNodeService.java
        │                           ├── MyDaemonInstallationNodeContribution.java
        │                           ├── MyDaemonProgramNodeService.java
        │                           ├── MyDaemonProgramNodeContribution.java
        │                           ├── UnknownResponseException.java
        │                           └── XMLRPCMyDaemonInterface.java
        └── resources
            ├── com
            │   └── ur
            │       └── urcap
            │           └── examples
            │               └── mydaemon
            │                   └── impl
            │                       ├── installation.html
            │                       ├── programnode.html
            │                       └── daemon
            │                           └── hello-world.py
            └── META-INF
                └── LICENSE
```

Figure 3: Structure of the My Daemon URCap project

The services:

- HelloWorldInstallationNodeService.java
- HelloWorldProgramNodeService.java

are registered in Activator.java and thereby a new installation node and program node are offered to PolyScope. The My Daemon additionally registers its MyDaemonDaemonService.java service to make the daemon executable available to PolyScope.

The file pom.xml contains a subsection (under the <properties> section) with a set of properties for the URCap with meta-data specifying the vendor, contact address, copyright, description, and short license information which will be displayed to the user when the URCap is installed in PolyScope. See Listing 1 for the Hello World version of these properties.

Listing 1: Section with meta-data properties inside the pom.xml file for the Hello World URCap

```
<!--***************************************************************-->
<!--   Note: Update this section with relevant meta data           -->
<!--         that comes along with your URCap                      -->
<!--***************************************************************-->
<!--****************** BEGINNING OF URCAP META DATA ******************-->
<urcap.symbolicname>com.ur.urcap.examples.helloworld</urcap.symbolicname>
<urcap.vendor>Universal Robots</urcap.vendor>
<urcap.contactAddress>Energivej 25, 5260 Odense S, Denmark</urcap.
    contactAddress>
<urcap.copyright>Copyright (C) 2009-2021 Universal Robots. All Rights Reserved
    </urcap.copyright>
<urcap.description>Hello World sample URCap</urcap.description>
<urcap.licenseType>Sample license</urcap.licenseType>
<urcap.compatibility.CB3>true</urcap.compatibility.CB3>
<urcap.compatibility.eSeries>true</urcap.compatibility.eSeries>
<!--********************* END OF URCAP META DATA *********************-->
```

The URCap meta-data subsection also contains two boolean compatibility flag properties (urcap.compatibility.CB3 and urcap.compatibility.eSeries) which specify the URCap's compatibility with the CB3 and the e-Series robot series, respectively. See section 12.1. Robot Series Compatibility for more information about these compatibility flags.

6 Deployment with Maven

In order to ease development, a URCap can be deployed using Maven.

Deployment to a robot with Maven Given the IP address of the robot, e.g. `10.2.128.64`, go to your URCap project folder and type:

```
$ cd samples/html/com.ur.urcap.examples.helloworld
$ mvn install -Premote -Durcap.install.host=10.2.128.64
```

and the URCap is deployed and installed on the robot. During this process PolyScope will be restarted.

You can also specify the IP address of the robot via the property `urcap.install.host` inside the `pom.xml` file. Then you can deploy by typing:

```
$ cd samples/html/com.ur.urcap.examples.helloworld
$ mvn install -Premote
```

Deployment to URSim If you are running Linux then URSim can be installed locally. Otherwise it needs to run in a virtual machine (VM). It is possible to deploy to both environments with Maven. As shown above parameters can be supplied either directly on the command line or in the `pom.xml` file.

- To deploy to a *locally running URSim* specify the path to the extracted URSim with the property `ursim.home`.
- To deploy to a *URSim running in a VM* specify the IP address of the VM using the property `ursimvm.install.host`.

Once the properties are configured you can deploy to a local URSim by using the `ursim` profile:

```
$ cd samples/html/com.ur.urcap.examples.helloworld
$ mvn install -P ursim
```

or the URSim running in a VM using the `ursimvm` profile:

```
$ cd samples/html/com.ur.urcap.examples.helloworld
$ mvn install -P ursimvm
```

Note, if you are using VirtualBox to run the VM you should make sure that the network of the VM is operating in bridged mode.

7 Contribution of an Installation Node

A URCap can contribute installation nodes. An installation node will support a customized installation node screen and customized functionality.

7.1 Layout of the Installation Node

The layout of a customized installation node screen is specified using a HTML document. At the moment only a fragment of valid CSS styling properties and HTML are supported. The most important HTML elements that are supported are various form input elements, labels, headings, paragraphs, and divisions. For a detailed overview of supported HTML and CSS, consult appendix B. CSS and HTML Support.

Listing 2: HTML document that specifies the layout of the customized Hello World installation screen

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World</title>
    <style>
      input {
        display: inline-block;
        width: 200px;
        height: 28px;
      }

      label {
        display: inline-block;
        width: 100px;
        height: 28px;
      }
    </style>
  </head>
  <body>
    <h1>Hello World</h1>
    <form>
      <p>The popup title below is shared between all Hello World program nodes
        .</p><br \>
      <p>The title cannot be empty.</p><br \>
      <div class="spacer"> </div>
      <label>Popup title:</label><input id="popuptitle" type="text"/>
    </form>
  </body>
</html>
```

Listing 2 shows the content of the `installation.html` file which defines the layout of the screen used for the Hello World installation node. The listing begins with CSS styling properties (defined within the `<style>` tag). After that follows a simple document with a heading, a paragraph, a label and a single text input widget.

In order to connect the HTML GUI specification to your Java implementation an `id` attribute is specified for the text field to access its value. This creates a model-view separation where the `id` is used to wire a Java object with the HTML widgets. In this way, methods in the Java implementation can be used to define behaviour for widgets identified by `id` attributes. The corresponding Java code is presented in the following two sections.

7.2 Making the customized Installation Node available to PolyScope

In order to make the layout specified in HTML and the customized installation nodes available to PolyScope, a Java class that implements the interface `InstallationNodeService` must be defined. Listing 3 shows the Java code that makes the Hello World installation node available to PolyScope.

Listing 3: Hello World Installation node service

```java
package com.ur.urcap.examples.helloworld.impl;

import com.ur.urcap.api.contribution.InstallationNodeContribution;
import com.ur.urcap.api.contribution.InstallationNodeService;
import com.ur.urcap.api.domain.URCapAPI;

import java.io.InputStream;

import com.ur.urcap.api.domain.data.DataModel;

public class HelloWorldInstallationNodeService implements InstallationNodeService {

  public HelloWorldInstallationNodeService() { }

  @Override
  public InstallationNodeContribution createInstallationNode(URCapAPI api, DataModel model) {
    return new HelloWorldInstallationNodeContribution(model);
  }

  @Override
  public String getTitle() {
    return "Hello World";
  }

  @Override
  public InputStream getHTML() {
    InputStream is = this.getClass().getResourceAsStream("/com/ur/urcap/examples/helloworld/impl/installation.html");
    return is;
  }
}
```

The `InstallationNodeService` interface requires the following methods to be defined:

- `getHTML()` returns an input stream with the HTML which is passed into PolyScope.
- `getTitle()` returns the title for the node, to be shown on the left side of the Installation tab to access the customized installation screen. For simplicity, the title is specified simply as `"Hello World"`. In a more realistic example, the return value of the `getTitle()` method would be translated into the language specified by standard Java localization, based on the locale provided by `Locale.getDefault()`.
- `createInstallationNode(URCapAPI, DataModel)` is called by PolyScope when it needs to create an instance of the installation node. Both a `URCapAPI` and a `DataModel` object is passed in as arguments. The first gives access to the domain of PolyScope and the second provides a data model with automatic persistence. The constructor used in the implementation of the method `createInstallationNode(...)` is discussed in section 7.3. Functionality of the Installation Node. All modifications to the supplied data model from the installation node constructor are ignored when existing installation is loaded. This means that ideally the installation node constructor should not set anything in the data model.

7.3 Functionality of the Installation Node

The functionality behind a customized installation node must be defined in a Java class that implements the `InstallationNodeContribution` interface. Listing 4 shows the Java code that defines the functionality of the Hello World installation screen. An instance of this class is returned by the `createInstallationNode(...)` method in the `HelloWorldInstallationNodeService` class described in previous section.

In essence, the `InstallationNodeContribution` interface requires the following to be defined:

1. What happens when the user enters and exits the customized installation screen.
2. Script code that should be added to the preamble of any program when run with this URCap installed.

In addition, the class contains code that links the HTML widgets to concrete field variables, gives access to a data model with automatic persistence, listening for GUI events and UR-Script generation associated with the node.

Listing 4: Java class defining functionality for the Hello World installation node

```java
package com.ur.urcap.examples.helloworld.impl;

import com.ur.urcap.api.contribution.InstallationNodeContribution;
import com.ur.urcap.api.domain.data.DataModel;
import com.ur.urcap.api.domain.script.ScriptWriter;
import com.ur.urcap.api.ui.annotation.Input;
import com.ur.urcap.api.ui.component.InputEvent;
import com.ur.urcap.api.ui.component.InputTextField;

public class HelloWorldInstallationNodeContribution implements
    InstallationNodeContribution {

  private static final String POPUPTITLE_KEY = "popuptitle";
  private static final String DEFAULT_VALUE = "Hello World";

  private DataModel model;

  public HelloWorldInstallationNodeContribution(DataModel model) {
    this.model = model;
  }

  @Input(id = POPUPTITLE_KEY)
  private InputTextField popupTitleField;

  @Input(id = POPUPTITLE_KEY)
  public void onMessageChange(InputEvent event) {
    if (event.getEventType() == InputEvent.EventType.ON_CHANGE) {
      setPopupTitle(popupTitleField.getText());
    }
  }

  @Override
  public void openView() {
    popupTitleField.setText(getPopupTitle());
  }

  @Override
  public void closeView() { }

  public boolean isDefined() {
    return !getPopupTitle().isEmpty();
  }

  @Override
```

```
  public void generateScript(ScriptWriter writer) {
    // Store the popup title in a global variable so it is globally available to all HelloWorld program nodes.
    writer.assign("hello_world_popup_title", "\"" + getPopupTitle() + "\"");
  }

  public String getPopupTitle() {
    return model.get(POPUPTITLE_KEY, DEFAULT_VALUE);
  }

  private void setPopupTitle(String message) {
    if ("".equals(message)) {
      resetToDefaultValue();
    } else {
      model.set(POPUPTITLE_KEY, message);
    }
  }

  private void resetToDefaultValue() {
    popupTitleField.setText(DEFAULT_VALUE);
    model.set(POPUPTITLE_KEY, DEFAULT_VALUE);
  }
}
```

The data model which was mentioned in section 7.2. Making the customized Installation Node available to PolyScope is passed into the constructor through a `DataModel` object. All data that needs to be saved and loaded along with a robot installation must be stored in and retrieved from this model object.

The HTML text input widget has an `id` attribute equal to `"popuptitle"`. This attribute maps the HTML widget to an object of type `InputTextField` which permits basic operations on text fields. This is achieved using the annotation `@Input` by specifying its argument `id`.

Particularly, when the user interacts with the widget by, e.g., entering some text, the method `onMessageChange(InputEvent)` is called. Its argument indicates what kind of interaction has occurred. The code within that method takes care of storing the contents of the text input widget in the data model under the key `POPUPTITLE_KEY` whenever the content of the widget changes. By saving and loading the robot installation you will notice that values are stored and read again from and back to the `popupTitleField`.

The `openView()` method is called whenever the user enters the screen. It sets the contents of the text input widget to the current value stored within the member field `message`. The `closeView()` method is called when the user leaves the screen.

Finally, the preamble of each program run with this URCap installed will contain an assignment in its preamble, as specified in the `generateScript(ScriptWriter)` method. In the assignment, the script variable named `"hello_world_popup_title"` is assigned to a string that contains the popup title stored within the data model object.

8 Contribution of a Program Node

A URCap can contribute program nodes. A node is supplied by a customized view part and a part with the customized functionality.

8.1 Layout of the Program Node

The layout for customized program node screens is defined similarly as the layout of customized installation node screens (see section 7.1. Layout of the Installation Node). Listing 5 defines the layout of a simple program node. It contains a single input widget where the user can type a name and two labels that provide a preview of the popup that will be displayed at runtime. The label with id `"titlePreviewLabel"` will be used to display the title set in the installation. The name that is entered by the end user in the Hello World program node will be used to construct a customized popup message. This message will also be shown in the preview label with id `"messagePreviewLabel"`. The Java code that underlies these widgets is presented in the following two sections.

Listing 5: Layout of the customized Hello World program node

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World</title>
    <style>
      input {
        display: inline-block;
        width: 200px;
        height: 28px;
      }

      label {
        display: inline-block;
        width: 150px;
        height: 28px;
      }

      #preview {
        display: block;
        padding: 35px 20px 20px 0px;
      }

      #header {
        padding: 0px 0px 20px 0px;
      }
    </style>
  </head>
  <body>
    <form>
      <p>This program node will open a popup on execution.</p><br \>
      <label>Enter your name:</label> <input id="yourname" type="text"/>
      <br/>
      <div id="preview">
        <h3 id="header">Preview</h3><br \>
        Title: <label id="titlePreviewLabel" style="width:␣400px;"/> <br \>
        Message: <label id="messagePreviewLabel" style="width:␣400px;"/>
      </div>
    </form>
  </body>
</html>
```

8.2 Making the customized Program Nodes available to PolyScope

To make the Hello World program node available to PolyScope, a Java class that implements the `ProgramNodeService` interface is required. Listing 6 shows the Java code that makes the Hello World program node available to PolyScope.

The `getId()` method returns the unique identifier for this type of program node. The identifier will be used when storing programs that contain these program nodes. Do not change the return value of this method in released URCaps, since it will break backwards compatibility for existing programs. URCap program nodes in such existing programs will not be loaded properly and the program can not run anymore.

Its `getTitle()` method supplies the text for the button in the Structure Tab that corresponds to this type of program node. It is also used as the heading on the Command tab for such nodes.

Letting the method `isDeprecated()` return `true` makes it impossible to create new program nodes of this type, but still support loading program nodes of this type in existing programs.

If the method `isChildrenAllowed()` returns `true`, it signals that it is possible for the program node to contain other (child) program nodes.

Finally, `createNode(URCapAPI, DataModel)` creates program nodes with references to the `URCapAPI` and the `DataModel`. The first gives access to the domain of PolyScope and the second gives the user a data model with automatic persistence. The `createNode(...)` method creates a new Java object for each node of this type occurring in the program tree. The constructor used in the implementation of the `createNode(...)` method is discussed in section 8.3. Functionality of the Program Node. The `createNode(...)` method call during program load is discussed in section 8.5. Loading Programs with Program Node Contributions

The returned object is used when interacting with widgets on the customized program node screen for the particular node selected in the program tree. It must use the supplied data model object to retrieve and store data that should be saved and loaded within the robot program along with the corresponding node occurrence. Please note that only data related to the current configuration of that particular program node instance should be stored in the data model, i.e. no global or shared state, state irrevelant to this node instance, etc. should be stored.

Listing 6: Java class defining how Hello World program nodes are created

```java
package com.ur.urcap.examples.helloworld.impl;

import com.ur.urcap.api.contribution.ProgramNodeContribution;
import com.ur.urcap.api.contribution.ProgramNodeService;
import com.ur.urcap.api.domain.URCapAPI;
import com.ur.urcap.api.domain.data.DataModel;

import java.io.InputStream;

public class HelloWorldProgramNodeService implements ProgramNodeService {

  public HelloWorldProgramNodeService() {
  }

  @Override
  public String getId() {
    return "HelloWorldNode";
  }

```

```
  @Override
  public String getTitle() {
    return "Hello␣World";
  }

  @Override
  public InputStream getHTML() {
    InputStream is = this.getClass().getResourceAsStream("/com/ur/urcap/
        examples/helloworld/impl/programnode.html");
    return is;
  }

  @Override
  public boolean isDeprecated() {
    return false;
  }

  @Override
  public boolean isChildrenAllowed() {
    return true;
  }

  @Override
  public ProgramNodeContribution createNode(URCapAPI api, DataModel model) {
    return new HelloWorldProgramNodeContribution(api, model);
  }
}
```

8.3 Functionality of the Program Node

The functionality of the Hello World program node is implemented in the Java class in Listing 7. This class implements the `ProgramNodeContribution` interface and instances of this class are returned by the `createNode(URCapAPI, DataModel)` of the `HelloWorldProgramNodeService` class described in the previous section.

Listing 7: Java class defining functionality for the Hello World program node

```
package com.ur.urcap.examples.helloworld.impl;

import com.ur.urcap.api.contribution.ProgramNodeContribution;
import com.ur.urcap.api.domain.URCapAPI;
import com.ur.urcap.api.domain.data.DataModel;
import com.ur.urcap.api.domain.script.ScriptWriter;
import com.ur.urcap.api.ui.annotation.Input;
import com.ur.urcap.api.ui.annotation.Label;
import com.ur.urcap.api.ui.component.InputEvent;
import com.ur.urcap.api.ui.component.InputTextField;
import com.ur.urcap.api.ui.component.LabelComponent;

public class HelloWorldProgramNodeContribution implements
    ProgramNodeContribution {
  private static final String NAME = "name";

  private final DataModel model;
  private final URCapAPI api;

  public HelloWorldProgramNodeContribution(URCapAPI api, DataModel model) {
    this.api = api;
    this.model = model;
  }

  @Input(id = "yourname")
  private InputTextField nameTextField;

```

```java
  @Label(id = "titlePreviewLabel")
  private LabelComponent titlePreviewLabel;

  @Label(id = "messagePreviewLabel")
  private LabelComponent messagePreviewLabel;

  @Input(id = "yourname")
  public void onInput(InputEvent event) {
    if (event.getEventType() == InputEvent.EventType.ON_CHANGE) {
      setName(nameTextField.getText());
      updatePopupMessageAndPreview();
    }
  }

  @Override
  public void openView() {
    nameTextField.setText(getName());
    updatePopupMessageAndPreview();
  }

  @Override
  public void closeView() {
  }

  @Override
  public String getTitle() {
    return "Hello World: " + (model.isSet(NAME) ? getName() : "");
  }

  @Override
  public boolean isDefined() {
    return getInstallation().isDefined() && !getName().isEmpty();
  }

  @Override
  public void generateScript(ScriptWriter writer) {
    // Directly generate this Program Node's popup message + access the popup title through a global variable
    writer.appendLine("popup(\"" + generatePopupMessage() + "\", hello_world_popup_title, False, False, blocking=True)");
    writer.writeChildren();
  }

  private String generatePopupMessage() {
    return model.isSet(NAME) ? "Hello " + getName() + ", welcome to PolyScope!" : "No name set";
  }

  private void updatePopupMessageAndPreview() {
    messagePreviewLabel.setText(generatePopupMessage());
    titlePreviewLabel.setText(getInstallation().isDefined() ? getInstallation().getPopupTitle() : "No title set");
  }

  private String getName() {
    return model.get(NAME, "");
  }

  private void setName(String name) {
    if ("".equals(name)){
      model.remove(NAME);
    }else{
      model.set(NAME, name);
    }
  }

  private HelloWorldInstallationNodeContribution getInstallation() {
```

```
        return api.getInstallationNode(HelloWorldInstallationNodeContribution.
            class);
    }

}
```

The openView() and closeView() methods specify what happens when the user selects and unselects the underlying program node in the program tree.

The getTitle() method defines the text which is displayed in the program tree for the node. The text of the node in the program tree is updated when values are written to the DataModel.

The isDefined() method serves to identify whether the node is completely defined (green) or still undefined (yellow). Note that a node, which can contain other program nodes (see section 8.2. Making the customized Program Nodes available to PolyScope), remains undefined as long as it has a child that is undefined. The isDefined() method is called when values are written to the DataModel to ensure that the program tree reflects the proper state of the program node.

Finally, generateScript(ScriptWriter) is called to add script code to the spot where the underlying node occurs in the robot program.

As the user interacts with the text input widget, the constructed message is displayed on the screen in the label with id "messagePreviewLabel". Each Hello World node is defined (green) if both the Hello World installation node is defined and the name in the program node is non-empty. When executed, it shows a simple popup dialog with the title defined in the installation and the message constructed from the name.

The popup title is the value of the script variable hello_world_popup_title. This variable is initialized by the script code contributed by the Hello World installation node. Thus, the script variable serves to pass data from the contributed installation node to the contributed program node. Another approach to pass information between these two objects is by directly requesting the installation object through the URCapAPI class. The Hello World program node utilizes this approach in its updatePopupMessageAndPreview() method.

8.4 Updating the data model

8.4.1 Restrictions

It is not allowed to modify the supplied data model in the implementation of any overridden method defined in the program node contribution interface (described in section 8.3. Functionality of the Program Node), which PolyScope calls. The data model should rather be updated due to a user-initiated event, e.g. a button click.

Furthermore, if the URCap contains an installation node contribution, the data model of the installation contribution should not be modified from the program node contribution. If such modifications occur in the call to openView() while a program is running, PolyScope will automatically stop the program every time the URCap program node is encountered in the program tree, because the installation was changed. It is, however, allowed to read from the data model in the installation node contribution.

8.4.2 Undo/redo Functionality

The supplied data model automatically supports undo/redo functionality in the sense that every bulk change to the data model is recorded on the undo stack. A bulk change is every change that happens to the data model in the scope of a user-initiated event to the URCap. Changes to the sub-tree and its properties done in the scope of the event are also part of the bulk change.

If changes made to properties of nodes in the sub-tree happen outside the scope of a user-initiated event, these changes will not be recorded on the undo stack.

For example, if two changes are made to the data model and a node with a property set is added to the sub-tree as a result of the user clicking a button in the user interface, this counts as one undo action. When a user clicks undo the previous values are restored in the data model as well as the program tree and a call to the `openView()` method is made, for an opportunity to display the new values.

This also means that no values should be cached in member variables, but always retrieved from the data model, as there is no guarantee that things have not changed. Also keep in mind, that the user might not select the Command tab of the URCap, so there is no guaranteed call to `openView()`. This can be the case when loading a program that has already been setup.

8.5 Loading Programs with Program Node Contributions

Program node contributions contain a data model and an interface to manipulate the sub-tree (introduced in URCap API version 1.1.0).

When a contributable program node is created in PolyScope by the user, the data model given to the method `createNode(URCapAPI, DataModel)` is empty.

When a program is loaded, the method `createNode(URCapAPI, DataModel)` is called for each persisted program node contribution to re-create the program tree. In contrast to newly creating the program node, the data model now contains the data from the persisted node. All modifications to the data model from the program node constructor are ignored. This means that ideally the program node constructor should not set anything in the data model.

For creating sub-trees a program model can be used. In some of the URCap examples in Chapter 10. URCap Examples Overview it is demonstrated how a sub-tree can be generated programmatically. The program model provides the interface `TreeNode` to create and manipulate the sub-tree. When a contributable program node is created in PolyScope by the user, the tree node has no children. The program model can be requested through the `URCapAPI` interface.

When a program is loaded each program node is deserialized on its own, this includes sub-trees previously created through the program model. Also now, the tree node requested through the `URCapAPI` interface is empty. The program node factory returned by `getProgramNodeFactory()` in the `ProgramModel` interface will return program nodes without any functionality. In particular, the method `createURCapProgramNode(Class<? extends URCapProgramNodeService>)` does not call the `createNode(...)` method in the specified service. Therefore, modifications are ignored during the `createNode(...)` call.

9 Contribution of a Daemon

A daemon can be any executable script or binary file that runs on the control box. The My Daemon URCap serves as the running example for explaining this functionality and is an extension of the Hello World example. The My Daemon example offers the same functionality from the user's point of view as the Hello World example.

However, the My Daemon URCap performs its tasks through an executable, which acts as a sort of driver or server. The executable is implemented as Python 2.5 script and C++ binary. The executables communicate with the Java front-end and URScript executor through XML encoded Remote Procedure Calls (XML-RPC). Figure 3, page 14, shows the structure of the My Daemon URCap project.

9.1 Daemon Service

A URCap can contribute any number of daemon executables through implementation of the `DaemonService` interface (see Listing 8):

- The `init(DaemonContribution)` method will be called by PolyScope with a `DaemonContribution` object which gives the URCap developer the control to install, start, stop, and query the state of the daemon. An example of how to integrate start, stop, and query a daemon will be discussed in Section 9.2. Interaction with the Daemon.
- The `installResource(URL url)` method in the `DaemonContribution` interface takes an argument that points to the source inside the URCap Jar file (`.urcap` file). This path may point to a single executable daemon or a directory that contains a daemon and additional files (e.g. dynamic linked libraries or configuration files).
- The implementation of `getExecutable()` provides PolyScope with the path to the executable that will be started.

The `/etc/service` directory contains links to the URCap daemon executables currently running. If a daemon executable has a link present but is in fact not running, the `ERROR` state will be returned upon querying the daemon's state. The links to daemon executables follow the lifetime of the encapsulating URCap and will be removed when the URCap is removed. The initial state for a daemon is `STOPPED`, however if it is desired, auto-start can be achieved by calling `start()` in the `init(DaemonContribution)` method right after the daemon has had its resources installed.

Listing 8: The My Daemon Service

```
package com.ur.urcap.examples.mydaemon.impl;

import com.ur.urcap.api.contribution.DaemonContribution;
import com.ur.urcap.api.contribution.DaemonService;

import java.net.MalformedURLException;
import java.net.URL;


public class MyDaemonDaemonService implements DaemonService {

  private DaemonContribution daemonContribution;

  public MyDaemonDaemonService() {
  }

  @Override
  public void init(DaemonContribution daemonContribution) {
    this.daemonContribution = daemonContribution;
```

```
    try {
      daemonContribution.installResource(new URL("file:com/ur/urcap/examples/mydaemon/impl/daemon/"));
    } catch (MalformedURLException e) { }
  }

  @Override
  public URL getExecutable() {
    try {
      // Two equivalent example daemons are available:
      return new URL("file:com/ur/urcap/examples/mydaemon/impl/daemon/hello-world.py"); // Python executable
      // return new URL("file:com/ur/urcap/examples/mydaemon/impl/daemon/HelloWorld"); // C++ executable
    } catch (MalformedURLException e) {
      return null;
    }
  }

  public DaemonContribution getDaemon() {
    return daemonContribution;
  }

}
```

Log information with respect to the process handling of the daemon executable are saved together with the daemon executable (follow the symbolic link of the daemon executable in `/etc/service` to locate the log directory).

Note, that script daemons must include an interpreter directive at the first line to help select the right program for interpreting the script. For instance, Bash scripts use `"#!/bin/bash"` and Python scripts use `"#!/usr/bin/env␣python"`.

9.2 Interaction with the Daemon

The My Daemon installation screen is shown in Figure 4 and Figure 5 for a CB3 robot and a e-Series robot, respectively. The code can be found in Listing 18, page 54, in Appendix C. My Daemon Program and Installation Node.

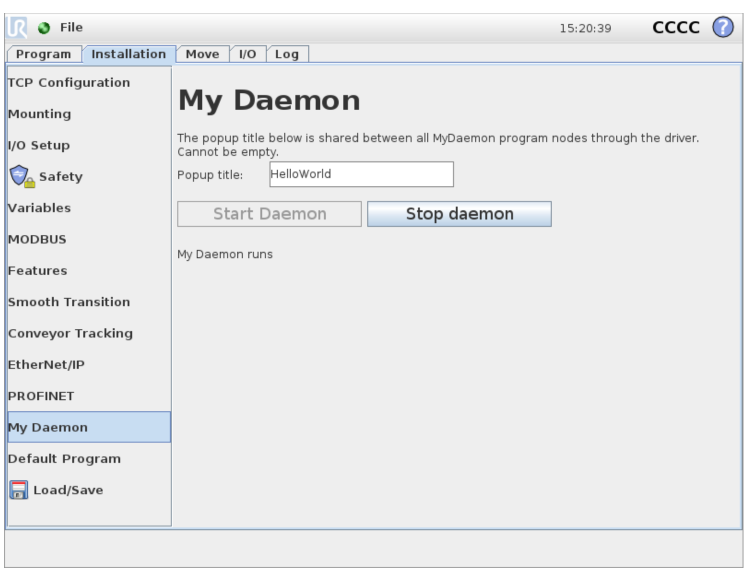


Figure 4: My Daemon installation screen on CB3 robot

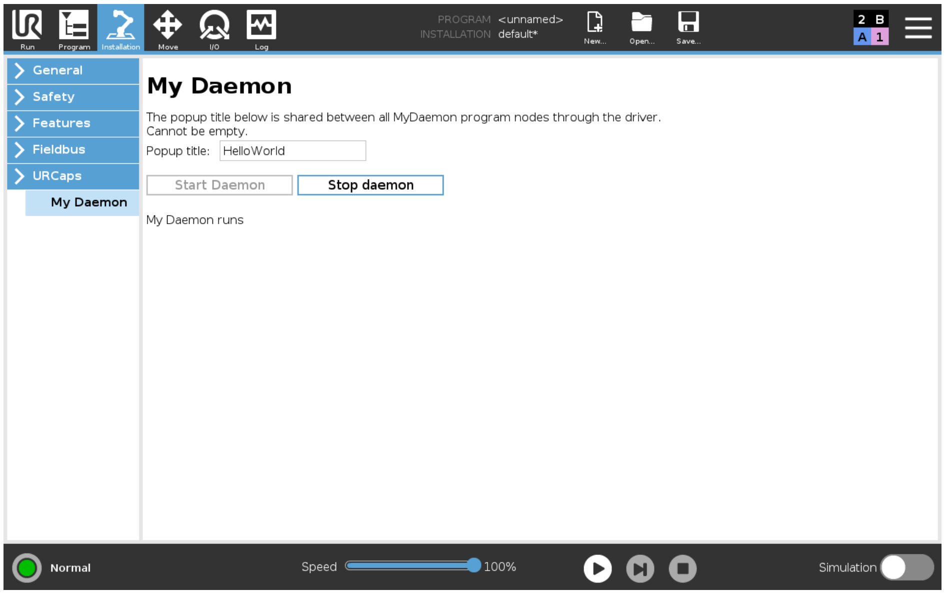


Figure 5: My Daemon installation screen on e-Series robot

Two buttons have been added to the installation screen to enable and disable the daemon. The button widgets are accessible in Java in a similar manner as described in Section 7.3. Functionality of the Installation Node, i.e. through annotations with `id` attributes (`btnEnableDaemon` and `btnDisableDaemon` respectively). In this example the daemon is enabled by default when a new installation is created, and future changes to the desired run state will be stored in the data model.

The daemon runs in parallel with PolyScope and can in principle change its state independently. Therefore, the label below the buttons displays the current run status of the daemon. This label is updated with a 1 Hz frequency, utilizing the `java.util.Timer` class. Since the UI update is initiated from a different thread than the Java AWT thread, the timer task must utilize the `EventQueue.invokeLater` functionality. Note, the `Timer` is added when the My Daemon Installation screen is opened (see `openView()`) and removed when the user moves away from the screen (see `closeView()`) to conserve computing resources.

Two options are available for Java and URScript to communicate with the daemon:

- TCP/IP sockets can be used to stream data.
- XML encoded Remote Procedure Calls (XML-RPC) can be used for configuration tasks (e.g. camera calibration) or service execution (e.g. locating the next object).

The advantage of XML-RPC over sockets is that no custom protocol or encoding needs to be implemented. The URScript XML-RPC implementation supports all URScript data types. Moreover, a RPC will only return when the function execution has been completed. This is desirable when the next program step relies on data retrieved from the daemon service. Plain sockets are on the other hand more efficient for data streaming, since there is no encoding overhead. Both methods can be complimentary applied and are available for Java ↔ daemon and URScript ↔ daemon communication.

Listing 9: URScript XML-RPC example

```
global mydaemon = rpc_factory("xmlrpc", "http://127.0.0.1:40404/RPC2")
global mydaemon_message = mydaemon.get_message("Bob")
popup(mydaemon_message, "My␣Title", False, False, blocking=True)
```

Listing 9 shows a small URScript example for making a XML-RPC call to a XML-RPC server.

The hello-world.py example daemon (see Listing 21, page 61) can be used as XML-RPC test server. Simply start the daemon in the My Daemon and run the URScript in a Script node.

The intention of this URScript example is to retrieve a message from the daemon to display during runtime (similar to the My Daemon program node). The rpc_factory script function creates a connection to the XML-RPC server in the daemon. The new connection is stored in the global my_daemon variable and serves as a handle. The next line then requests the XML-RPC server in the daemon to execute the get_message(...) function with the string argument "Bob" and return the result. The return value of the RPC call is stored in the mydaemon_message variable for further processing in the popup(...) script function.

Note, making XML-RPC calls from URScript does not require any additional function stubs or pre-definitions of the remote function to be executed in URScript. Until the XML-RPC returns this URScript thread is automatically blocked (i.e. no sync nor Wait is needed). The standard XML-RPC protocol does not allow void return values and XML-RPC extensions enabling this are not always compatible.

The My Daemon example also includes a Java XML-RPC client example, see the combination of the MyDaemonProgramNodeContribution and XMLRPCMyDaemonInterface classes (Listing 19, page 57 and listing 20, page 59 respectively). Note, the execution of the XML-RPC calls is not on the main Java AWT thread, but offloaded to a separate thread.

9.3 C/C++ Daemon Executables

The CB3.0/3.1 and CB5.0 control boxes all run a minimal Debian 32-bit Linux operating system. To guarantee binary compatibility all C/C++ executables should be compiled with the urtool3 cross-compiler under Linux. The urtool3 cross-compiler is included in the SDK installation.

To test if the urtool3 is properly installed type the following in a terminal:

```
echo $URTOOL_ROOT; i686-unknown-linux-gnu-g++ --version
```

The correct output is:

```
/opt/urtool-3.0
i686-unknown-linux-gnu-g++ (GCC) 4.1.2
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

If the first line is not printed directly after installing the SDK, please reboot your PC for the environment variables to be updated.

The My Daemon URCap comes with a fully functional C++ XML-RPC server example that is equivalent to the hello-world.py Python daemon. Simply switch the comments in the function getExecutable() in the MyDaemonDaemonService class (Listing 8, page 27), and recompile to use the C++ daemon implementation. The popup title should now be appended with "(C++)" instead of "(Python)" during execution of the URCap.

The C++ daemon directory structure is shown in Figure 6, page 31. For managing the software construction process of the C++ daemon a tool called Scons is used. The SConstruct file among other things contains the main configuration, the urtool3 cross-compiler, and libxmlrpc-c integration. The SConscript files are used to define the compilation targets, e.g. the Hello World binary.

```
com.ur.urcap.examples.mydaemon
├── ...
├── daemon
│   ├── service
│   │   ├── AbyssServer.cpp
│   │   ├── AbyssServer.hpp
│   │   ├── SConscript
│   │   ├── XMLRPCMethods.cpp
│   │   └── XMLRPCMethods.hpp
│   ├── Data.cpp
│   ├── Data.hpp
│   ├── HelloWorld.cpp
│   ├── HelloWorld.cpp
│   ├── SConscript
│   └── SConstruct
└── ...
```

Figure 6: Structure of the C++ daemon of the My Daemon project

For the example URCap, the daemon will be build as part of the URCap build process by maven. However, the daemon can also be compiled manually by typing the following in a terminal:

```
cd com.ur.urcap.examples.mydaemon/daemon
scons release=1
```

This will build a release version of the daemon. Using `release=0` will build an executable with debugging symbols.

The XML-RPC functionality in the C++ daemon relies on the open-source library libxmlrpc-c (`http://xmlrpc-c.sourceforge.net`). This library is by default available on the CB3.0/3.1 and CB5.0 control boxes. The `service` directory contains all relevant XML-RPC code. The AbyssServer is one of the XML-RPC server implementations supported by libxmlrpc-c. Please look in the C++ code for more programming hints and links to relevant documentation.

9.4 Tying the different Contributions together

The new My Daemon URCap installation node, program node, and daemon executable are registered and offered to PolyScope through the code in Listing 10.

Three services are registered:

- `MyDaemonInstallationNodeService`
- `MyDaemonProgramNodeService`
- `MyDaemonDaemonService`

The `MyDaemonInstallationNodeService` class has visibility to an instance of the `MyDaemonDaemonService` class. This instance is passed in the constructor when a new installation node instance of the type `MyDaemonInstallationNodeContribution` is created with the `createInstallationNode(...)` method. In this way, the daemon executable can be controlled from the installation node.

Listing 10: Tying different URCap contributions together

```java
package com.ur.urcap.examples.mydaemon.impl;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import com.ur.urcap.api.contribution.InstallationNodeService;
import com.ur.urcap.api.contribution.ProgramNodeService;
import com.ur.urcap.api.contribution.DaemonService;

public class Activator implements BundleActivator {
  @Override
  public void start(final BundleContext context) throws Exception {
    MyDaemonDaemonService daemonService = new MyDaemonDaemonService();
    MyDaemonInstallationNodeService installationNodeService = new
        MyDaemonInstallationNodeService(daemonService);

    context.registerService(InstallationNodeService.class,
        installationNodeService, null);
    context.registerService(ProgramNodeService.class, new
        MyDaemonProgramNodeService(), null);
    context.registerService(DaemonService.class, daemonService, null);
  }

  @Override
  public void stop(BundleContext context) throws Exception {
  }
}
```

10 URCap Examples Overview

This section contains a short description of each of the URCap examples included with the URCaps SDK.

10.1 Regular Samples

Hello World serves as the primary example throughout this tutorial and introduces all the core concepts of a URCap. This includes contributions to PolyScope of program nodes and installation nodes that seamlessly hook into:

- The UI
- Persistence of program and installation files
- Creation and execution of programs
- Program undo/redo functionality

Information:

- *Available from:*
  - URCap API version 1.0.0.
  - PolyScope version 3.3.0.
- *Main API interfaces:* ProgramNodeService, ProgramNodeContribution, InstallationNodeService, InstallationNodeContribution, DataModel, ScriptWriter.

My Daemon is an extension to the Hello World URCap and demonstrates how a Python 2.5 or C++ daemon can be integrated with the URCap Software Platform. This is useful when a URCap depends on e.g. a driver or server which is not implemented in Java. Furthermore, the URCap shows how the XML-RPC protocol can be used to communicate with the daemon from an installation node and in the script code generated by a program node.

Information:

- *Available from:*
  - URCap API version 1.0.0.
  - PolyScope version 3.3.0.
- *Main API interfaces:* DaemonContribution, DaemonService.

Script Function demonstrates how to add functions to the list of available script functions in the Expression Editor. Script functions often used by end users of a URCap should be added to this list.

Information:

- *Available from:*
  - URCap API version 1.1.0.
  - PolyScope version 3.4.0.
- *Main API interfaces:* Function, FunctionModel

Pick or Place is a toy example that shows how to make changes to the program tree through the TreeNode API. The marker interface NonUserInsertable is used for program node contributions that can only be inserted into the program tree by a URCap and not from the UI of PolyScope by the end user.

Information:

- *Available from:*
  - URCap API version 1.1.0.
  - PolyScope version 3.4.0.
- *Main API interfaces:* ProgramModel, TreeNode, ProgramNodeFactory

Ellipse is a toy example, where a pose is used to define the center point for an ellipse-like movement. The movement is achieved by inserting a pre-configured MoveP program node containing pre-defined and named Waypoint nodes. This example demonstrates how to:

- Obtain a pose for the robot position by requesting the end user to define it using the Move Tab
- Name waypoints
- Request the end user to move the robot to a given target position
- Allow the end user to use the builtin PolyScope support for starting from and pausing/breaking on a selected program node in the program tree. In this case, the end user can start from or break on a specific Waypoint child node under the Ellipse (URCap) program node.

Note:

- The functionality of assigning the Waypoint nodes custom names is only available from URCap API version 1.4.0 (released with PolyScope version 3.7.0/5.1.0)
- Requesting the end user to move the robot to a defined center point is only available from URCap API version 1.5.0 (released with PolyScope version 3.8.0/5.2.0).
- From URCap API version 1.6.0 (released with PolyScope version 3.9.0/5.3.0) the use of the deprecated move node config factory (MoveNodeConfigFactory interface) has been replaced with the equivalent builder and the TCP selection of the MoveP node is pre-configured.
- Support for allowing the end user to start from and break on child nodes is only available from URCap API version 1.9.0 (released with PolyScope version 5.6.0).
- From URCap API version 1.11.0 (released with PolyScope 3.14.0/5.9.0) the use of the deprecated method `getUserDefinedRobotPosition(RobotPositionCallback)` has been replaced with the equivalent `getUserDefinedRobotPosition(RobotPositionCallback2)` method. Furthermore, the use of the deprecated factory method for creating fixed position Waypoint node configurations (`createFixedPositionConfig(...)` without TCP offset) has been replaced with the equivalent method taking the TCP offset as parameter as well.

Information:

- *Available from:*
  - URCap API version 1.2.56.
  - PolyScope version 3.5.0.

- *Main API interfaces:* UserInteraction, RobotPositionCallback2, RobotMovement, RobotMovementCallback, WaypointNodeConfig, MovePMoveNodeConfig, MoveNodeConfigBuilders, MovePConfigBuilder, PoseFactory, Pose, SimpleValueFactory, JointPositions, ProgramNodeServiceConfigurable

Cycle Counter demonstrates how to work with variables. In this example, the chosen variable will be incremented each time the program node is executed.

This sample also demonstrates how to allow the end user to use the builtin PolyScope support for starting from and pausing/breaking on a selected program node in the program tree. In this case, the end user can start from or break on any child node under the Cycle Counter (URCap) program node.

Note:

- Support for allowing the end user to start from and break on child nodes is only available from URCap API version 1.9.0 (released with PolyScope version 5.6.0)

Information:

- *Available from:*
  - URCap API version 1.2.56.
  - PolyScope version 3.5.0.
- *Main API interfaces:* Variable, VariableFactory, ExpressionBuilder, ProgramNodeServiceConfigurable

Idle Time demonstrates how to work with the ProgramNodeVisitor to traverse all program nodes in a sub-tree. In this example, all Wait nodes will be visited. If a Wait node is configured to wait for an amount of time, that amount of idle time (in seconds) will accumulate in the selected variable.

Information:

- *Available from:*
  - URCap API version 1.2.56.
  - PolyScope version 3.5.0.
- *Main API interfaces:* ProgramNodeVisitor, WaitNodeConfig

Coordinate Map demonstrates how to capture click or touch coordinates when the user interacts with an image.

Information:

- *Available from:*
  - URCap API version 1.2.56.
  - PolyScope version 3.5.0.
- *Main API interfaces:* TouchEvent

Localization demonstrates how to implement localization in URCaps. PolyScope localization settings is accessed through the SystemSettings API.

Information:

- *Available from:*
  - URCap API version 1.2.56.
  - PolyScope version 3.5.0.
- *Main API interfaces:* SystemSettings, Localization, Translatable, UnitType, SimpleValueFactory

Node Ordering demonstrates how to define a specific sort order in PolyScope for the program node contributions from a URCap.

Information:

- *Available from:*
  - URCap API version 1.5.0.
  - PolyScope version 3.8.0/5.2.0.
- *Main API interfaces:* ProgramNodeServiceConfigurable, ProgramNodeConfiguration.

10.2 Driver Samples

Custom User Inputs demonstrates how to use different types of user inputs (e.g. combo box and checkbox inputs) and other (non-user input) UI elements (e.g text components) when defining a custom UI for a driver contribution.

The URCap also shows how to use the filler UI element for controlling/grouping the layout and how to add a custom input validator for detecting errors for enterable user inputs, in this case an IP address user input (text field).

Information:

- *Available from:*
  - URCap API version 1.7.0.
  - PolyScope version 3.11.0/5.5.0.
- *Main API interfaces:* CustomUserInputConfiguration.

Simple Gripper demonstrates how to create a gripper driver contribution for a basic gripper that only supports the mandatory "default" grip and release actions. The URCap uses digital outputs in the script code generation to trigger grip and release actions.

For further details about gripper driver contributions, please see the separate *Gripper Driver* document.

Information:

- *Available from:*
  - URCap API version 1.8.0.
  - PolyScope version 3.11.0/5.5.0.
- *Main API interfaces:* GripperContribution.

Advanced Gripper demonstrates how to create a gripper driver contribution for a more advanced gripper that supports some of the optional gripper capabilities and controls the Output Voltage setting of the Tool I/O Interface resource. The URCap shows how to:

- Configure gripper capabilities for, e.g. width, speed, force, vacuum and feedback for grip and release detection
- Request exclusive control of the Tool I/O Interface resource
- Configure the Output Voltage I/O setting of the Tool I/O Interface

Additional information:

- For details about gripper driver contributions, please see the separate *Gripper Driver* document.
- For information about system resource control, please see the separate *Resource Control* document.
- To see an example of how some of the other settings of the Tool I/O Interface can be configured, see the *Tool I/O Control Swing* URCap example.

Note:

- Feedback capabilities of the gripper is only available from URCap API version 1.9.0 (released with PolyScope version 3.12.0/5.6.0)

Information:

- *Available from:*
  - URCap API version 1.8.0.
  - PolyScope version 3.11.0/5.5.0.
- *Main API interfaces:* GripperContribution, GripperCapabilities, SystemConfiguration, GripActionParameters, ReleaseActionParameters, GripperFeedbackCapabilities.

Custom Gripper Setup demonstrates how to create a gripper driver contribution which defines a custom UI in the installation node for setting up the gripper as well as adds a TCP for the gripper to PolyScope.

For further details about gripper driver contributions, please see the separate *Gripper Driver* document.

Information:

- *Available from:*
  - URCap API version 1.8.0.
  - PolyScope version 3.11.0/5.5.0.

- *Main API interfaces:* GripperContribution, TCPConfiguration, CustomUserInputConfiguration.

Dual Zone Gripper demonstrates how to create a gripper driver contribution for a multi-gripper that supports multiple permanently enabled, individual grippers as well as dynamically adjusts the parameters of a capability (after it has been registered) for all individual grippers.

In this example, the gripper is a vacuum dual gripper with two independent physical grippers (zones) "built into" the gripping device. The URCap provides the user the option to select from three individual zones (grippers), named *Zone A*, *Zone B* and *Zone A+B*. The user can operate *Zone A* and *Zone B* independently of one another or choose to use both zones at the same by selecting *Zone A+B*.

A custom UI is defined in the installation node which allows the user to enable the "Use Fragile Handling" option. When enabled, this will restrict the maximum vaccum level available for a grip action. This is achieved by reducing the maximum value and default value parameters of the registered vacuum capability simultaneously for all zones.

The URCap shows how to:

- Configure the multi-gripper capability to support multiple permanently enabled, individual grippers
- Add a dedicated TCP for each individual gripper (zone)
- Dynamically update the parameter values of a registered (parameter-based) capability for all individual grippers (zones). In this case, the registered capability is the grip vaccum capability.

For further details about gripper driver contributions, please see the separate *Gripper Driver* document.

Information:

- *Available from:*
  - URCap API version 1.11.0.
  - PolyScope version 3.14.0/5.9.0.
- *Main API interfaces:* GripperContribution, GripperCapabilities, GripperListProvider, GripperListBuilder, SelectableGripper, SystemConfiguration, TCPConfiguration, GripVacuumCapability

Dynamic Multi-Gripper demonstrates how to create a gripper driver contribution that supports both a single gripper setup and multi-gripper setup as well as dynamically adjusts the parameters of a capability (after it has been registered) exclusively for each individual gripper.

In this example, the gripper driver provides the user the option of choosing between using a setup with only a single gripper mounted on the robot, or one where two separate, identical grippers are mounted on the robot at the same time. Each individual gripper supports moving to a user configurable position (open/close to a configurable width).

A custom UI is defined in the installation node which allows the user to select how many gripper are mounted on the robot. The available options are *Single* and *Dual*. Selecting the *Single* option will disable the secondary gripper, *Gripper 2*, and only the standard gripper, *Gripper 1*, will be available to the user. Choosing the *Dual* option will enable both grippers making both of them available to the user. Depending on the option selected, the offset of TCP for the standard gripper (*Gripper 1*) is updated accordingly.

For each individual gripper (*Gripper 1* and *Gripper 2*), the user can configure the type of fingertips attached to the gripper (in the installation node). There are two options, *Standard* and *Extended*. The *Standard* option is the default whereas the *Extended* option can be selected, when fingertips with wide range is attached. Selecting the *Extended* option will increase the maximum value and default value parameters of the registered width capability exclusively for the specific individual gripper (independently of the other gripper).

The URCap shows how to:

- Configure the multi-gripper capability to support multiple individual grippers where some of the grippers are initially disabled
- Dynamically enable and disable individual grippers
- Add a dedicated TCP for each individual gripper
- Dynamically update the parameter values of a registered (parameter-based) capability exclusively for an individual gripper (independently of the other grippers). In this case, the registered capability is the width capability.

For further details about gripper driver contributions, please see the separate *Gripper Driver* document.

Information:

- *Available from:*
  - URCap API version 1.11.0.
  - PolyScope version 3.14.0/5.9.0.
- *Main API interfaces:* GripperContribution, GripperCapabilities, GripperListProvider, GripperListBuilder, SelectableGripper, MultiGripperCapability, SystemConfiguration, TCPConfiguration, WidthCapability

Simple Screwdriver demonstrates how to create a screwdriver driver contribution for a basic screwdriver that only supports the mandatory start and stop screwdriver operations. The URCap uses digital outputs in the script code generation to start and stop the screwdriver.

For further details about screwdriver driver contributions, please see the separate *Screwdriver Driver* document.

Information:

- *Available from:*
  - URCap API version 1.9.0.

– PolyScope version 5.6.0.

Advanced Screwdriver demonstrates how to create a screwdriver driver contribution that supports some of the optional operation and operation feedback capabilities, e.g. Program Selection, Feed Screw, Drive Screw OK and Screwdrive Ready. The URCap primarily uses I/Os for the implementation of the capabilities.

For further details about screwdriver driver contributions, please see the separate *Screwdriver Driver* document.

Information:

- *Available from:*
  – URCap API version 1.9.0.
  – PolyScope version 5.6.0.

Custom Screwdriver demonstrates how to create a screwdriver contribution driver which defines a custom UI in the Screwdriving installation node for setting up the screwdriver as well as adds a default TCP for the screwdriver to PolyScope.

For further details about screwdriver driver contributions, please see the separate *Screwdriver Driver* document.

Information:

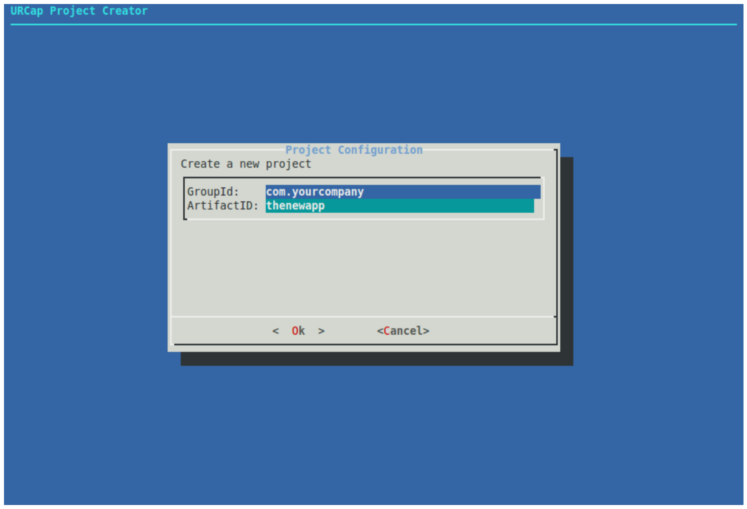
- *Available from:*
  – URCap API version 1.9.0.
  – PolyScope version 5.6.0.

11 Creating new thin Projects using a Maven Archetype

There are different ways to get started with URCap development. One is to start with an existing URCap project and modify that. When you have got a hang of it you may want to start with an empty skeleton with the basic Maven structure. So enter the directory of the URCaps SDK and type:

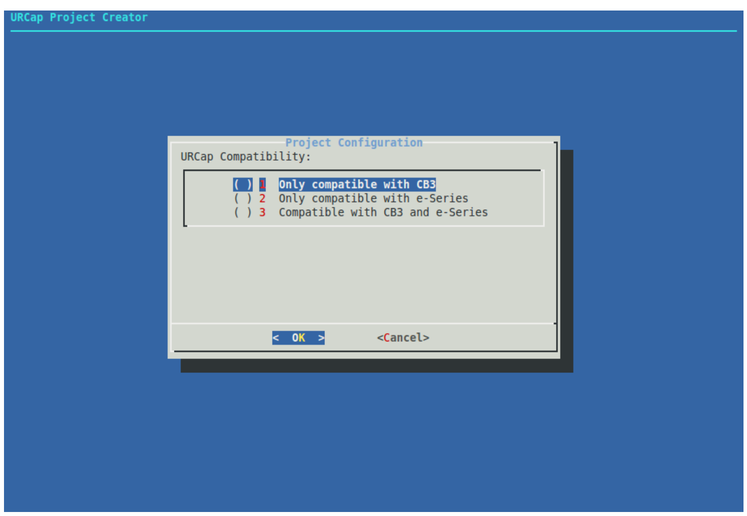
```
$ ./newURCap.sh
```

This prompts you with a dialog box where you select a group and artifact-id for your new URCap:



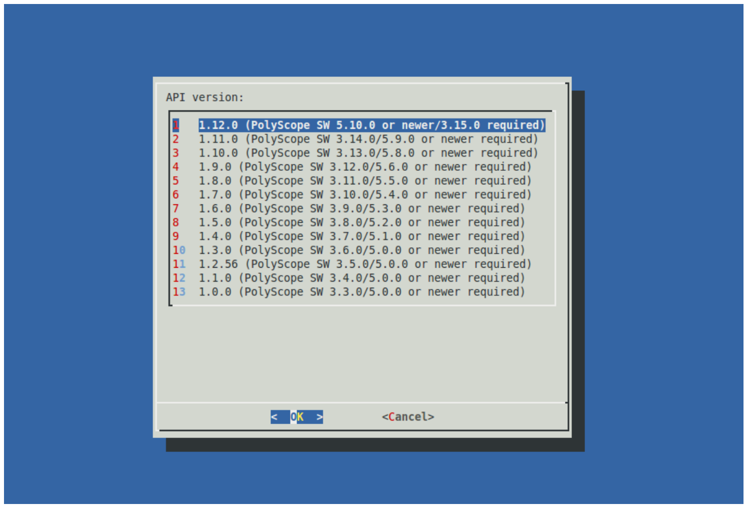
An example could be `com.yourcompany` as group-id and `thenewapp` as artifact-id. Consult best practices naming conventions for Java group-ids.

Next you must specify which Universal Robots robot series your URCap is compatible with:



Maybe your URCap requires specific hardware and/or software features, which are only available on a specific robot series (e.g., e-Series robots only), or the URCap might be compatible with all robot series. Such considerations should determine your choice here. See section 12.1. Robot Series Compatibility for more details about robot series compatibility.

Finally, you must also specify the target URCap API version. Choosing an earlier version of the API will make your URCap compatible with earlier PolyScope versions, but also limit the functionality accessible through the API. URCap API compatibility is described in section 12.2. API Compatibility.



Pressing the Ok button in the last dialog creates a new Maven project under the sub-folder `./com.yourcompany.thenewapp`. This project can easily be imported into an IDE for Java, e.g. Eclipse, Netbeans, or IntelliJ.

Notice that the generated `pom.xml` file has a section with a set of properties for the new URCap with meta-data for vendor, contact address, copyright, description, and short license information which will be displayed to the user when the URCap is installed in PolyScope. This section also contains boolean compatibility flag properties specifying the URCap's compatibility with the different available robot series. Update the URCap meta-data section with the data relevant for the new URCap. See Listing 1 for an example of how this section might look.

Should you need to change the version of the URCap API to depend upon after your project has been setup, this can be done in the `pom.xml` file in your project. Here you must update to the desired version in the URCap API dependency under the `<dependencies>` section of the `pom.xml`-file as well as modify the `<import-package>` element under the `maven-bundle-plugin` if version information is present. See listings 11, 12 and 15 for examples of this

Listing 11: Modify URCap API runtime dependency in `pom.xml`

```
...
<Import-Package>
    com.ur.urcap.api*,
    *
</Import-Package>
...
```

Listing 12: Specifying URCap API compile time dependency in `pom.xml`

```
...
<dependencies>
  ...
  <dependency>
    <groupId>com.ur.urcap</groupId>
    <artifactId>api</artifactId>
    <version>1.0.0.30</version>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>
...
```

12 Compatibility

12.1 Robot Series Compatibility

When developing URCaps you must mandatorily specify which robot series your URCap is compatible with, i.e. which robot series your URCap can run on. The purpose is to provide a good user experience for the URCap installation process by ensuring that end users cannot install a URCap that is not compatible with their robot and/or Universal Robots offline simulator software (URSim).

When specifying your URCap's compatibility, you need to consider, if your URCap depends on specific hardware and/or software functionality, which is not available on all robot series (e.g., only available on e-Series robots), or if the URCap can run on all available robot series. Note that your URCap's compatibility must be specified for all robot series available in the line of robots from Universal Robots (including robot series which your URCap is incompatible with).

12.1.1 Compatibility Flags

The URCap's compatibility is specified using boolean compatibility flags which are defined as properties in the `pom.xml` Maven configuration file in your URCap project. Two compatibility flag properties, `urcap.compatibility.CB3` and `urcap.compatibility.eSeries`, must be present in the URCap meta-data section in the `pom.xml` file. An example of the meta-data section with the two compatibility flag properties is shown in Listing 13.

Listing 13: Compatibility flags in the `pom.xml` file's URCap meta-data properties section

```
<!--********************************************************************-->
<!--   Note: Update this section with relevant meta data                -->
<!--         that comes along with your URCap                           -->
<!--********************************************************************-->
<!--****************** BEGINNING OF URCAP META DATA ******************-->
<urcap.symbolicname>com.ur.urcap.examples.helloworld</urcap.symbolicname>
<urcap.vendor>Universal Robots</urcap.vendor>
<urcap.contactAddress>Energivej 25, 5260 Odense S, Denmark</urcap.contactAddress>
<urcap.copyright>Copyright (C) 2009-2021 Universal Robots. All Rights Reserved</urcap.copyright>
<urcap.description>Hello World sample URCap</urcap.description>
<urcap.licenseType>Sample license</urcap.licenseType>
<urcap.compatibility.CB3>true</urcap.compatibility.CB3>
<urcap.compatibility.eSeries>true</urcap.compatibility.eSeries>
<!--********************* END OF URCAP META DATA *********************-->
```

These two properties specify the URCap's compatibility with the CB3 and the e-Series robot series, respectively. Acceptable values for the properties are either `true` or `false`. A value of `true` indicates that the URCap can be installed and run on the given robot series, and a value of `false` indicates otherwise. Note that both flags must present including the flag for a robot series which your URCap is not compatible with.

Using the `newURCap.sh` script mentioned in section 11. Creating new thin Projects using a Maven Archetype, the compatibility flags are handled automatically for you. On the other hand, if an existing URCap needs to be updated to support the compatibility flags, you must manually update your URCap project's `pom.xml` file. This will only be necessary, if you used a version of the URCap SDK lower than 1.12.0 to create the URCap by either using the `newURCap.sh` script, or by basing the URCap on a modification of one of the URCap example projects included in the SDK. The required changes are described in subsection 12.1.3. Updating Existing URCaps.

If the compatibility of your URCap changes, remember to update the value of the URCap's compatibility flags accordingly.

12.1.2 Installation and Startup of URCaps

When the end user attempts to install a new URCap through the URCap Setup/Settings screen in PolyScope, PolyScope will inspect the URCap's compatibilty flags. Depending on the boolean value of the URCap's compatibility flags and the type of series of the end user's robot (or offline simulator), the user will either be allowed to install or prevented from installing that URCap. If the URCap is incompatible, the end user will be informed of this through a dialog.

Similarly, a URCap's compatibility flags are examined during startup of PolyScope, if the URCap was deployed and installed using Maven (described in section 6. Deployment with Maven) instead of being installed through the PolyScope UI. If the URCap is incompatible, it will not be allowed to start and an error dialog will be displayed in PolyScope.

The presence of the compatibility flags in a URCap will be verified during startup of PolyScope (if the URCap was deployed and installed using Maven), and a URCap will not be started, if any of the flags are missing. Likewise, a URCap with missing compatibility flag(s) will not be installable through PolyScope. An error dialog will be displayed in PolyScope in these cases.

Note that only URCaps depending on URCap API version 1.12.0 or higher will be prevented from starting or being installed, if the compatibility flags are not present.

12.1.3 Updating Existing URCaps

If you have an existing URCap that needs to be updated to support the compatibility flags, changes must be applied to your URCap project's `pom.xml` file.

First of all, you must add the two compatibility flag properties, `urcap.compatibility.CB3` and `urcap.compatibility.eSeries`, to the URCap meta-data section as shown in Listing 13 in section 12.1.1. Compatibility Flags. Remember to specify boolean values corresponding to your URCap's compatibility.

Listing 14: Excerpt of the `pom.xml` file's Maven bundle plugin configuration with compatiblity flag support

```
...
<build>
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      ...
      <configuration>
        <instructions>
          <!--******** DO NOT MODIFY THE ENTRIES OF THIS SECTION ********-->
          <Bundle-Category>URCap</Bundle-Category>
          ...
          <URCapCompatibility-CB3>${urcap.compatibility.CB3}</URCapCompatibility-CB3>
          <URCapCompatibility-eSeries>${urcap.compatibility.eSeries}</URCapCompatibility-eSeries>
          <!--*******************************************************-->
        ...
        <instructions>
      <configuration>
    <plugin>
```

```
      ...
    <plugins>
  ...
```

In order for PolyScope to be able to read the specified values for the compatibility flag properties, the `<build>` section with configuration of build plugins also needs to be updated. Here the configuration for the Maven bundle plugin must be updated to contain two new instructions, `<URCapCompatibility-CB3>` and `<URCapCompatibility-eSeries>`, as shown in the excerpt in Listing 14.

For convenience, instead of manually making the changes described above, you can use the `upgradeURCap.sh` script to update your `pom.xml` file.

Below are examples of usage of the script, where `/home/myuser/myURCap` is the path to the folder with a URCap project and `/home/myuser/sdk` is the path to the folder of the URCap SDK.

Go to the folder of the URCap SDK and type:

```
$ ./upgradeURCap.sh /home/myuser/myURCap
```

This will apply the required changes to the URCap project's `pom.xml` file inside the folder `/home/myuser/myURCap`. Both compatibility flag properties are assigned the boolean value of `true` by default (i.e. your URCap is compatible with both the CB3 and e-Series robot series). Remember to change the value of the compatibility flag properties in the `pom.xml` file accordingly, if the preassigned default values does not match your URCap's compatibility.

Instead of specifying an absolute path to your URCap project folder, you can also specify a relative path:

```
$ ./upgradeURCap.sh ../myURCap
```

Alternatively, it is also possible to copy the `upgradeURCap.sh` script to your URCap project folder and execute the script from within that folder by typing:

```
$ cd myURCap
$ ./upgradeURCap.sh .
```

Finally, you have the option of directly specifying the boolean value for the compatibility flags using the `compatibilityCB3` and `compatibilityESeries` arguments:

```
$ ./upgradeURCap.sh ../myURCap compatibilityCB3=false compatibilityESeries=
   true
```

In this case, you have specified that your URCap is compatible with the e-Series robot series, but incompatible with the CB3 robot series.

12.2 API Compatibility

When developing URCaps you must specify a dependency on a version of the URCap API to compile against. Using the `newURCap.sh` script mentioned in section 11. Creating new thin Projects using a Maven Archetype, this is handled automatically for you. How to update the version of the URCap API dependency for an existing URCap project is described at the end of section 11. Creating new thin Projects using a Maven Archetype.

A given version of the API is compatible with a specific version of PolyScope (see table 1 below). PolyScope will remain backwards compatible with earlier versions of the API. This means that if you choose to use the newest API, customers using your URCap must be running at least the version of PolyScope with which the given API was released.

It is not a problem if the customer is running a newer (future) version of PolyScope. However, it is not possible for the customer to use your URCap if he is running an earlier version of PolyScope than the one the API was released with. A good rule of thumb is thus to choose the earliest possible version of the API that supports the functionality you wish to use. This will target the broadest audience.

For instance, if you specify a dependency on the API version 1.1.0, your URCap will only run on PolyScope version 3.4.0 or newer. If you wish to target the broadest possible audience, you must use version 1.0.0 of the API and the customers must be running PolyScope version 3.3.0 or newer.

Note that API version 1.12.0 is the final API version supported on CB3 robots. Higher API versions are only available on e-Series robots.

URCap API version	Min. PolyScope version
1.13.0	5.11.0
1.12.0	3.15.0/5.10.0
1.11.0	3.14.0/5.9.0
1.10.0	3.13.0/5.8.0
1.9.0	3.12.0/5.6.0
1.8.0	3.11.0/5.5.0
1.7.0	3.10.0/5.4.0
1.6.0	3.9.0/5.3.0
1.5.0	3.8.0/5.2.0
1.4.0	3.7.0/5.1.0
1.3.0	3.6.0/5.0.4
1.2.56	3.5.0
1.1.0	3.4.0
1.0.0	3.3.0

Table 1: API versions and PolyScope version requirements

12.2.1 Advanced compatibility

Contrary to what is described above, it is possible to load a URCap depending on a newer API than what is officially supported by PolyScope. By configuring your URCap to resolve its dependencies runtime rather than install time, PolyScope will start your URCap regardless of the URCap API version dependency specified. Care must be taken to have a code execution path that does not use anything not available in the API that the given version of PolyScope supports (otherwise a `NoSuchMethodError` or `NoClassDefFoundError` will be thrown).

As an example you could have a dependency on API version 1.5.0 and run it on PolyScope version 3.7.0/5.1.0 (officially only supporting API version 1.4.0), but in the actual execution path you can only use types present in API version 1.4.0.

To have the URCap resolve at runtime the `pom.xml` must have the option `;resolution:=optional` appended to the URCap API entry in the `<import-package>` section. The full `<import-package>` section could look like this:

Listing 15: Excerpt of `pom.xml` file for advanced compatibility

```
...
<Import-Package>
    com.ur.urcap.api*;resolution:=optional,
    *
</Import-Package>
...
```

This will make the URCap start up regardless of what the actual dependency states.

As mentioned you must also structure your code so no code referring unsupported API functionality is executed. Also no `import` or `catch` clauses referring to unsupported types can be present in classes that will execute. See listings 16 and 17 for an example of how to structure this. In listing 16 all code related to unsupported API functionality is located and in 17 a check for PolyScope version number is performed before creating an instance of the `AdvancedFeature` class.

Listing 16: `AdvancedFeature` class showing advanced compatiblity

```
import com.ur.urcap.api.contribution.installation.InstallationAPIProvider;
import com.ur.urcap.api.domain.InstallationAPI;
import com.ur.urcap.api.domain.value.Pose;
import com.ur.urcap.api.domain.value.PoseFactory;
import com.ur.urcap.api.domain.value.simple.Angle;
import com.ur.urcap.api.domain.value.simple.Length;

public class AdvancedFeature {
  private final InstallationAPIProvider apiProvider;

  public AdvancedFeature(InstallationAPIProvider apiProvider) {
    this.apiProvider = apiProvider;
  }

  public void addTCP() {
    InstallationAPI installationAPI = apiProvider.getInstallationAPI();
    PoseFactory poseFactory = installationAPI.getValueFactoryProvider().getPoseFactory();
    Pose pose = poseFactory.createPose(0, 0, 100, 0, 0, 0, Length.Unit.MM, Angle.Unit.RAD);
    apiProvider.getInstallationAPI().getTCPContributionModel().addTCP("GRIPPER_TCP", "Gripper", pose);
  }
}
```

Listing 17: Usage of `AdvancedFeature` class

```
    SoftwareVersion softwareVersion = apiProvider.getSystemAPI().getSoftwareVersion();
    if ((softwareVersion.getMajorVersion() == 3 && softwareVersion.getMinorVersion() >= 8) ||
      softwareVersion.getMajorVersion() == 5 && softwareVersion.getMinorVersion() >= 2) {
      new AdvancedFeature(apiProvider).addTCP();
    }
```

If you choose to use this advanced feature, you must test your URCap carefully on all PolyScope versions you wish to support making sure all code execution paths are tested.

13 Exception Handling

All exceptions thrown and not caught in a URCap will be caught by PolyScope. If this happens when the end user selects either an installation node or a program node from the URCap, the UI provided by the URCap will be replaced by a screen displaying information about the error. In all other cases, a dialog will be shown to the end user.

The error screen and dialog will show that an exception has happened in the URCap along with meta information about the URCap. It will also contain a section showing the stack trace from the exception in the URCap code.

14 Troubleshooting

Internally in PolyScope, a URCap is installed as an OSGi bundle with the Apache Felix OSGi framework. For the purpose of debugging problems, it is possible to inspect various information about bundles using the Apache Felix command shell.

You can establish a shell connection to the running Apache Felix by opening a TCP connection on port 6666. Access the Apache Felix shell console by typing:

```
$ nc 127.0.0.1 6666

Felix Remote Shell Console:
============================

->
```

Note that you need to use the `nc` command, since the `telnet` command is not available on the robot, and `127.0.0.1` because `localhost` does not work on a robot.

To view a list of installed bundles and their state type the following command:

```
-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (5.2.0)
[   1] [Active     ] [    1] aopalliance (1.0)
[   2] [Active     ] [    1] org.aspectj.aspectjrt (1.8.2)
[   3] [Active     ] [    1] org.netbeans.awtextra (1.0)
[   4] [Active     ] [    1] net.java.balloontip (1.2.4)
[   5] [Active     ] [    1] cglib (2.2)
[   6] [Active     ] [    1] com.ur.dashboardserver (3.3.0.SNAPSHOT)
[   7] [Active     ] [    1] com.ur.domain (3.3.0.SNAPSHOT)
...
[  56] [Active     ] [    1] com.thoughtworks.xstream (1.3.1)
[  57] [Active     ] [    1] HelloWorld (1.0.0.SNAPSHOT)
->
```

Inside the shell you can type `help` to see the list of the available commands:

```
-> help
uninstall
sysprop
bundlelevel
find
version
headers
refresh
start
obr
inspect
ps
stop
shutdown
help
update
install
log
cd
startlevel
resolve

Use 'help <command-name>' for more information.
->
```

For example, the `headers` command can be executed to display different properties of the individual installed bundles.

A URCaps and Generated Script Code

An URCap may generate script code for installation and program nodes. To aid debugging, the generated script code for both node types is annotated with URCap information.

The following example is taken from a small program using the Hello World URCap.

To see the generated script code for an URCap, it is required to save the program. Assuming the program is called `hello.urp`, the corresponding script code can be found in `hello.script`.

The script code of the installation node will be surrounded with begin/end URCap comments and information about the source of the URCap and its type:

```
...
# begin: URCap Installation Node
#   Source: Hello World, 1.0.0.SNAPSHOT, Universal Robots
#   Type: Hello World
hello_world_popup_title = "Hello␣World"
# end: URCap Installation Node
...
```

Similarly for program nodes, script code is surrounded with begin/end URCap comments and information about the source of the URCap and its type as shown below. The program node generated by PolyScope is the first label after the `# begin: URCap Program Node`, here `$ 2`. The remaining labels until `# end : URCap Program Node`, here the statement `$ 3`, are the nodes inserted under the Hello World program node.

```
...
# begin: URCap Program Node
#   Source: Hello World, 1.0.0.SNAPSHOT, Universal Robots
#   Type: Hello World
$ 2 "HelloWorld:␣My␣node"
popup("Hello␣My␣node,␣welcome␣to␣PolyScope!", hello_world_popup_title, False,
    False, blocking=True)
$ 3 "Wait:␣0.01"
sleep(0.01)
# end: URCap Program Node
...
```

B CSS and HTML Support

A strict subset of CSS and HTML is supported for defining the layout of customized program and installation node screens. Besides the basic HTML elements `<html>`, `<head>`, `<style>`, `<title>`, `<body>` the following HTML elements are the only supported:

1. The `<form>` element, when appearing as a child of `<body>`.
2. The `<div>` element, when appearing as a child of `<body>`, `<form>` or another `<div>`.
3. Any of the elements `<h1>`, `<h2>`, `<h3>`, `<p>`, and `<img>`, when appearing as a child of `<body>`, `<form>` or `<div>`.
4. Any of the elements `<label>`, `<input>`, and `<select>`, when appearing inside a `<form>` as a child of `<form>`, `<div>` or `<p>`.
5. The element `<option>`, when appearing as a child of `<select>`.
6. The elements `<span>`, `<em>`, `<b>`, `<i>`, `<br>`, and `<hr>`.

As for standard HTML attributes, only the `id` and `style` and `src` attributes are supported. Data input widgets correspond to HTML elements as follows:

- Text input field: `<input type="text"/>`.
- Number input field: `<input type="number"/>`.
  - A range can be specified with the attributes `min` and `max`.
  - The number input field can be restricted to the integers by setting the property `step` to the value 1, i.e. `step="1"`
- Button: `<input type="button"/>`.
- Check box: `<input type="checkbox"/>`.
- Radio button: `<input type="radio"/>`.
- Drop-down menu: `<select/>`.

Styling of HTML elements can be customized using the following CSS properties:

`display`: Modifies the layout of an element w.r.t. other elements. Allowed values are `inline`, `inline-block` and `block`.

`width`, `height`: Serve to specify the size of an element with display attribute different from `inline`. The accepted values are numbers, optionally followed by “px”, or percentages of the respective dimension of the parent element.

`padding`, `padding-top`, `padding-right`, `padding-bottom`, `padding-left`: Used to specify spacing around an element. The accepted values are numbers, optionally followed by “px”. The attribute `padding` may take 1, 2 or 4 values. All the other attributes take a single value.

`font-size`, `font-style`, `font-weight`: For modifying the size and type of font used within the element. Allowed values for `font-size` are numbers, optionally followed by “px”, or percentages of the font size of the parent element. Allowed values for `font-style` are `normal` and `italic`. Allowed values for `font-weight` are `normal` and `bold`.

`vertical-align`: Sets the vertical aligment of text within a label element. Allowed values for `vertical-align` are `top`, `middle` and `bottom`.

`text-align`: Sets the horizontal aligment of text within a label element. Allowed values for `text-align` are `left`, `center` and `right`.

C My Daemon Program and Installation Node

Listing 18: Java class defining functionality for the My Daemon installation node

```
package com.ur.urcap.examples.mydaemon.impl;

import com.ur.urcap.api.contribution.DaemonContribution;
import com.ur.urcap.api.contribution.InstallationNodeContribution;
import com.ur.urcap.api.domain.data.DataModel;
import com.ur.urcap.api.domain.script.ScriptWriter;
import com.ur.urcap.api.ui.annotation.Input;
import com.ur.urcap.api.ui.annotation.Label;
import com.ur.urcap.api.ui.component.InputButton;
import com.ur.urcap.api.ui.component.InputEvent;
import com.ur.urcap.api.ui.component.InputTextField;
import com.ur.urcap.api.ui.component.LabelComponent;

import java.awt.EventQueue;
import java.util.Timer;
import java.util.TimerTask;

public class MyDaemonInstallationNodeContribution implements
    InstallationNodeContribution {
  private static final String POPUPTITLE_KEY = "popuptitle";

  private static final String XMLRPC_VARIABLE = "my_daemon";
  private static final String ENABLED_KEY = "enabled";
  private static final String DEFAULT_VALUE = "HelloWorld";

  private DataModel model;
  private final MyDaemonDaemonService daemonService;
  private XmlRpcMyDaemonInterface xmlRpcDaemonInterface;
  private Timer uiTimer;

  public MyDaemonInstallationNodeContribution(MyDaemonDaemonService
      daemonService, DataModel model) {
    this.daemonService = daemonService;
    this.model = model;
    xmlRpcDaemonInterface = new XmlRpcMyDaemonInterface("127.0.0.1", 40404);
    applyDesiredDaemonStatus();
  }

  @Input(id = POPUPTITLE_KEY)
  private InputTextField popupTitleField;

  @Input(id = "btnEnableDaemon")
  private InputButton enableDaemonButton;

  @Input(id = "btnDisableDaemon")
  private InputButton disableDaemonButton;

  @Label(id = "lblDaemonStatus")
  private LabelComponent daemonStatusLabel;

  @Input(id = POPUPTITLE_KEY)
  public void onMessageChange(InputEvent event) {
    if (event.getEventType() == InputEvent.EventType.ON_CHANGE) {
      setPopupTitle(popupTitleField.getText());
    }
  }

  @Input(id = "btnEnableDaemon")
  public void onStartClick(InputEvent event) {
    if (event.getEventType() == InputEvent.EventType.ON_CHANGE) {
      setDaemonEnabled(true);
```

```
      applyDesiredDaemonStatus();
    }
  }

  @Input(id = "btnDisableDaemon")
  public void onStopClick(InputEvent event) {
    if (event.getEventType() == InputEvent.EventType.ON_CHANGE) {
      setDaemonEnabled(false);
      applyDesiredDaemonStatus();
    }
  }

  @Override
  public void openView() {
    enableDaemonButton.setText("Start␣Daemon");
    disableDaemonButton.setText("Stop␣daemon");
    popupTitleField.setText(getPopupTitle());

    //UI updates from non-GUI threads must use EventQueue.invokeLater (or
        SwingUtilities.invokeLater)
    uiTimer = new Timer(true);
    uiTimer.schedule(new TimerTask() {
      @Override
      public void run() {
        EventQueue.invokeLater(new Runnable() {
          @Override
          public void run() {
            updateUI();
          }
        });
      }
    }, 0, 1000);
  }

  private void updateUI() {
    DaemonContribution.State state = getDaemonState();

    if (state == DaemonContribution.State.RUNNING) {
      enableDaemonButton.setEnabled(false);
      disableDaemonButton.setEnabled(true);
    } else {
      enableDaemonButton.setEnabled(true);
      disableDaemonButton.setEnabled(false);
    }

    String text = "";
    switch (state) {
    case RUNNING:
      text = "My␣Daemon␣runs";
      break;
    case STOPPED:
      text = "My␣Daemon␣stopped";
      break;
    case ERROR:
      text = "My␣Daemon␣failed";
      break;
    }
    daemonStatusLabel.setText(text);
  }

  @Override
  public void closeView() {
    if (uiTimer != null) {
      uiTimer.cancel();
    }
  }

```

```java
  public boolean isDefined() {
    return !getPopupTitle().isEmpty() && getDaemonState() == DaemonContribution.State.RUNNING;
  }

  @Override
  public void generateScript(ScriptWriter writer) {
    writer.globalVariable(XMLRPC_VARIABLE, "rpc_factory(\"xmlrpc\", \"http://127.0.0.1:40404/RPC2\")");
    // Apply the settings to the daemon on program start in the Installation pre-amble
    writer.appendLine(XMLRPC_VARIABLE + ".set_title(\"" + getPopupTitle() + "\")");
  }

  public String getPopupTitle() {
    if (!model.isSet(POPUPTITLE_KEY)) {
      resetToDefaultValue();
    }
    return model.get(POPUPTITLE_KEY, DEFAULT_VALUE);
  }

  private void setPopupTitle(String title) {
    if ("".equals(title)) {
      resetToDefaultValue();
    } else {
      model.set(POPUPTITLE_KEY, title);
      // Apply the new setting to the daemon for real-time preview purposes
      // Note this might influence a running program, since the actual state is stored in the daemon.
      setDaemonTitle(title);
    }
  }

  private void resetToDefaultValue() {
    popupTitleField.setText(DEFAULT_VALUE);
    model.set(POPUPTITLE_KEY, DEFAULT_VALUE);
    setDaemonTitle(DEFAULT_VALUE);
  }

  private void setDaemonTitle(String title) {
    try {
      xmlRpcDaemonInterface.setTitle(title);
    } catch(Exception e){
      System.err.println("Could not set the title in the daemon process.");
    }
  }

  private void applyDesiredDaemonStatus() {
    new Thread(new Runnable() {
      @Override
      public void run() {
        if (isDaemonEnabled()) {
          // Download the daemon settings to the daemon process on initial start for real-time preview purposes
          try {
            awaitDaemonRunning(5000);
            xmlRpcDaemonInterface.setTitle(getPopupTitle());
          } catch (Exception e) {
            System.err.println("Could not set the title in the daemon process.");
          }
        } else {
          daemonService.getDaemon().stop();
        }
      }
    }).start();
```

```
  }

  private void awaitDaemonRunning(long timeOutMilliSeconds) throws
      InterruptedException {
    daemonService.getDaemon().start();
    long endTime = System.nanoTime() + timeOutMilliSeconds * 1000L * 1000L;
    while(System.nanoTime() < endTime && (daemonService.getDaemon().getState()
         != DaemonContribution.State.RUNNING || !xmlRpcDaemonInterface.
        isReachable())) {
      Thread.sleep(100);
    }
  }

  private DaemonContribution.State getDaemonState(){
    return daemonService.getDaemon().getState();
  }

  private Boolean isDaemonEnabled() {
    return model.get(ENABLED_KEY, true); //This daemon is enabled by default
  }

  private void setDaemonEnabled(Boolean enable) {
    model.set(ENABLED_KEY, enable);
  }

  public String getXMLRPCVariable(){
    return XMLRPC_VARIABLE;
  }

  public XmlRpcMyDaemonInterface getXmlRpcDaemonInterface() {return
      xmlRpcDaemonInterface; }
}
```

Listing 19: Java class defining functionality for the My Daemon program node

```
package com.ur.urcap.examples.mydaemon.impl;

import com.ur.urcap.api.contribution.ProgramNodeContribution;
import com.ur.urcap.api.domain.URCapAPI;
import com.ur.urcap.api.domain.data.DataModel;
import com.ur.urcap.api.domain.script.ScriptWriter;
import com.ur.urcap.api.ui.annotation.Input;
import com.ur.urcap.api.ui.annotation.Label;
import com.ur.urcap.api.ui.component.InputEvent;
import com.ur.urcap.api.ui.component.InputTextField;
import com.ur.urcap.api.ui.component.LabelComponent;

import java.awt.*;
import java.util.Timer;
import java.util.TimerTask;

public class MyDaemonProgramNodeContribution implements
    ProgramNodeContribution {
  private static final String NAME = "name";

  private final DataModel model;
  private final URCapAPI api;
  private Timer uiTimer;

  public MyDaemonProgramNodeContribution(URCapAPI api, DataModel model) {
    this.api = api;
    this.model = model;
  }

  @Input(id = "yourname")
  private InputTextField nameTextField;
```

```java

  @Label(id = "titlePreviewLabel")
  private LabelComponent titlePreviewLabel;

  @Label(id = "messagePreviewLabel")
  private LabelComponent messagePreviewLabel;

  @Input(id = "yourname")
  public void onInput(InputEvent event) {
    if (event.getEventType() == InputEvent.EventType.ON_CHANGE) {
      setName(nameTextField.getText());
      updatePreview();
    }
  }

  @Override
  public void openView() {
    nameTextField.setText(getName());

    //UI updates from non-GUI threads must use EventQueue.invokeLater (or
        SwingUtilities.invokeLater)
    uiTimer = new Timer(true);
    uiTimer.schedule(new TimerTask() {
      @Override
      public void run() {
        EventQueue.invokeLater(new Runnable() {
          @Override
          public void run() {
            updatePreview();
          }
        });
      }
    }, 0, 1000);
  }

  @Override
  public void closeView() {
    uiTimer.cancel();
  }

  @Override
  public String getTitle() {
    return "My Daemon: " + (model.isSet(NAME) ? getName() : "");
  }

  @Override
  public boolean isDefined() {
    return getInstallation().isDefined() && !getName().isEmpty();
  }

  @Override
  public void generateScript(ScriptWriter writer) {
    // Interact with the daemon process through XML-RPC calls
    // Note, alternatively plain sockets can be used.
    writer.assign("mydaemon_message", getInstallation().getXMLRPCVariable() +
        ".get_message(\"" + getName() + "\")");
    writer.assign("mydaemon_title", getInstallation().getXMLRPCVariable() + ".
        get_title()");
    writer.appendLine("popup(mydaemon_message, mydaemon_title, False, False, 
        blocking=True)");
    writer.writeChildren();
  }

  private void updatePreview() {
    String title = "";
    String message = "";
    try {
```

```
      // Provide a real-time preview of the daemon state
      title = getInstallation().getXmlRpcDaemonInterface().getTitle();
      message = getInstallation().getXmlRpcDaemonInterface().getMessage(getName());
    } catch (Exception e) {
      System.err.println("Could not retrieve essential data from the daemon process for the preview.");
      title = message = "<Daemon disconnected>";
    }

    titlePreviewLabel.setText(title);
    messagePreviewLabel.setText(message);
  }

  private String getName() {
    return model.get(NAME, "");
  }

  private void setName(String name) {
    if ("".equals(name)){
      model.remove(NAME);
    }else{
      model.set(NAME, name);
    }
  }

  private MyDaemonInstallationNodeContribution getInstallation(){
    return api.getInstallationNode(MyDaemonInstallationNodeContribution.class);
  }

}
```

Listing 20: Java class for XML-RPC communication

```
package com.ur.urcap.examples.mydaemon.impl;

import org.apache.xmlrpc.XmlRpcException;
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;

public class XmlRpcMyDaemonInterface {

  private final XmlRpcClient client;

  public XmlRpcMyDaemonInterface(String host, int port) {
    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
    config.setEnabledForExtensions(true);
    try {
      config.setServerURL(new URL("http://" + host + ":" + port + "/RPC2"));
    } catch (MalformedURLException e) {
      e.printStackTrace();
    }
    config.setConnectionTimeout(1000); //1s
    client = new XmlRpcClient();
    client.setConfig(config);
  }

  public boolean isReachable() {
    try {
      client.execute("get_title", new ArrayList<String>());
      return true;
```

```
    } catch (XmlRpcException e) {
      return false;
    }
  }

  public String getTitle() throws XmlRpcException, UnknownResponseException {
    Object result = client.execute("get_title", new ArrayList<String>());
    return processString(result);
  }

  public String setTitle(String title) throws XmlRpcException,
      UnknownResponseException {
    ArrayList<String> args = new ArrayList<String>();
    args.add(title);
    Object result = client.execute("set_title", args);
    return processString(result);
  }

  public String getMessage(String name) throws XmlRpcException,
      UnknownResponseException {
    ArrayList<String> args = new ArrayList<String>();
    args.add(name);
    Object result = client.execute("get_message", args);
    return processString(result);
  }

  private boolean processBoolean(Object response) throws
      UnknownResponseException {
    if (response instanceof Boolean) {
      Boolean val = (Boolean) response;
      return val.booleanValue();
    } else {
      throw new UnknownResponseException();
    }
  }

  private String processString(Object response) throws
      UnknownResponseException {
    if (response instanceof String) {
      return (String) response;
    } else {
      throw new UnknownResponseException();
    }
  }
}
```

Listing 21: hello-world.py Python 2.5 daemon example

```python
#!/usr/bin/env python

import sys

from SimpleXMLRPCServer import SimpleXMLRPCServer
from SocketServer import ThreadingMixIn

title = ""

def set_title(new_title):
  global title
  title = new_title
  return title

def get_title():
  tmp = ""
  if str(title):
    tmp = title
  else:
    tmp = "No␣title␣set"
  return tmp + "␣(Python)"

def get_message(name):
  if str(name):
    return "Hello␣" + str(name) + ",␣welcome␣to␣PolyScope!"
  else:
    return "No␣name␣set"

sys.stdout.write("MyDaemon␣daemon␣started")
sys.stderr.write("MyDaemon␣daemon␣started")

class MultithreadedSimpleXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
  pass

server = MultithreadedSimpleXMLRPCServer(("127.0.0.1", 40404))
server.RequestHandlerClass.protocol_version = "HTTP/1.1"
server.register_function(set_title, "set_title")
server.register_function(get_title, "get_title")
server.register_function(get_message, "get_message")
server.serve_forever()
```