

## Mobile Tutorial 2

### Mobile Tutorial 2 project build order

- A) If you have built all the other projects from the previous solutions step by step, you just need to build the projects in Column A.
- B) If you haven't built any of the previous tutorial solutions, you need to build all included projects in the order shown in Column B.

#### Column A

1. KUKATutorial2Orchestration
2. KUKATutorial2Dashboard
3. KUKAMobileTutorial2Simulation

#### Column B

1. Util
2. KUKACommandTypes
3. ArticulatedArm (in abstract services)
4. DriveDifferentialTwoWheel (in abstract services)
5. KUKARobotTool (in abstract services)
6. KUKAUniversalMotionPlanning (in abstract services)
7. Transformation
8. SimulatedLBR3Arm
9. KUKATutorial3MotionPlanning
10. KUKASimulatedLBR3Gripper
11. SimulatedDifferentialDrive
12. KUKATutorial2Orchestration
13. KUKATutorial2Dashboard
14. KUKAMobileTutorial2Simulation

### ***Queuing commands***

In this tutorial, we will use an orchestration service to queue commands and to dispatch these commands to their services. Just like in arm tutorial 4, the dashboard for this tutorial has a Command queue window that shows all commands which have not yet been processed.

#### *Add commands*

Using one of the motion buttons for the arm or any of the command buttons for platform and gripper will add a command into the queue.

### Delete commands

To delete a command highlight the command in the command queue window and click on 'Delete'. 'Delete All' empties the queue.

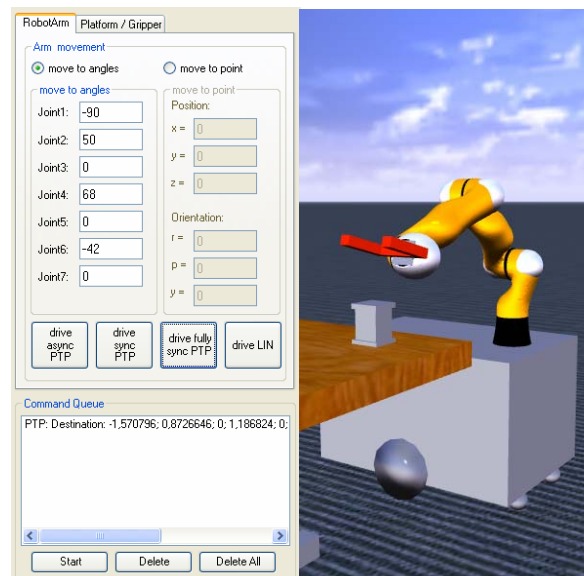
### Execute commands

Click on 'Start' to execute the current command list. After a command is finished, it is removed from the queue and the next command is executed.

## Example mobile robot application "Clearing the table"

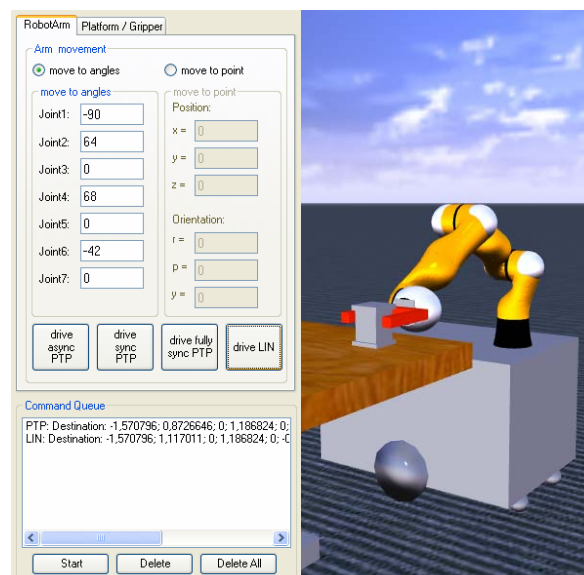
The following steps show you what commands to add to generate a first mobile robot application:

1) Move gripper to pre-grasp position:



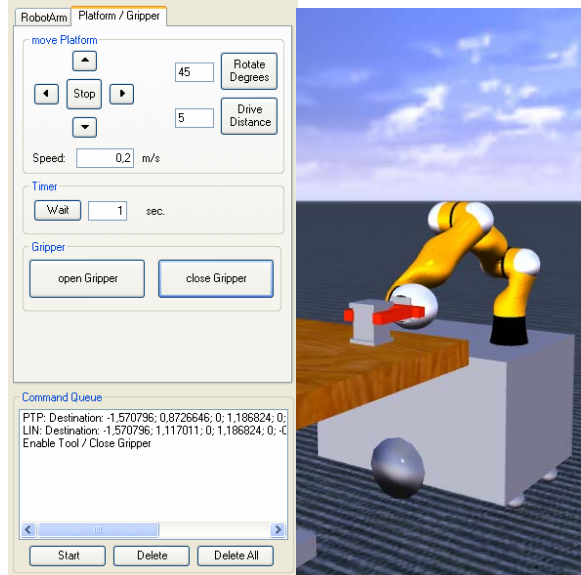
Often a "pre"-position is needed in robotics application to allow for a precise tool-part mating. These movements are usually done in a PTP fashion, since it allows for a time optimized approach.

2) Move Gripper to grasp position



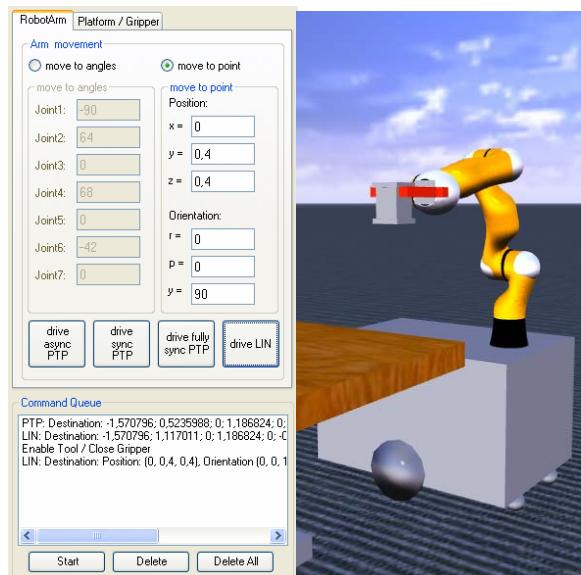
To mate the tool with the part a linear movement is usually the best choice to have a predictable tool center point (TCP) path.

### 3) Close Gripper



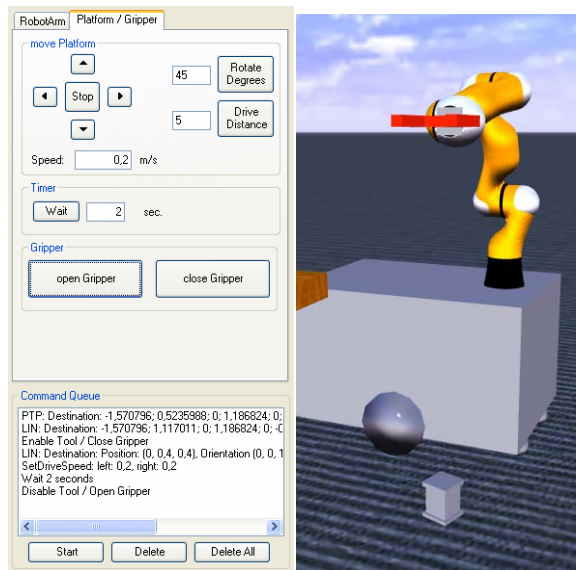
This is the first time the tool makes contact with the part. The advantage of the physics simulation is that we do not need to 'attach' the part to the gripper at this point, like it is often done in other existing simulation systems. The gripping is automatically simulated as well. We adjusted the part shape so that it can't slide out of the gripper. Things like this have now to be considered already at simulation time and spare the user some surprises in real life on the plant floor.

### 4) Pick up part



Here we use a LIN motion for handling the part. This is normally necessary especially when a process relative movement is executed, or when a complicated but predictable movement is needed (e.g. inserting a dashboard into a car). For time optimized motions use fully synchronous PTP motions.

## 5) Move and place part



Now the platform can drive the part to its destination and place it in its appropriate location.

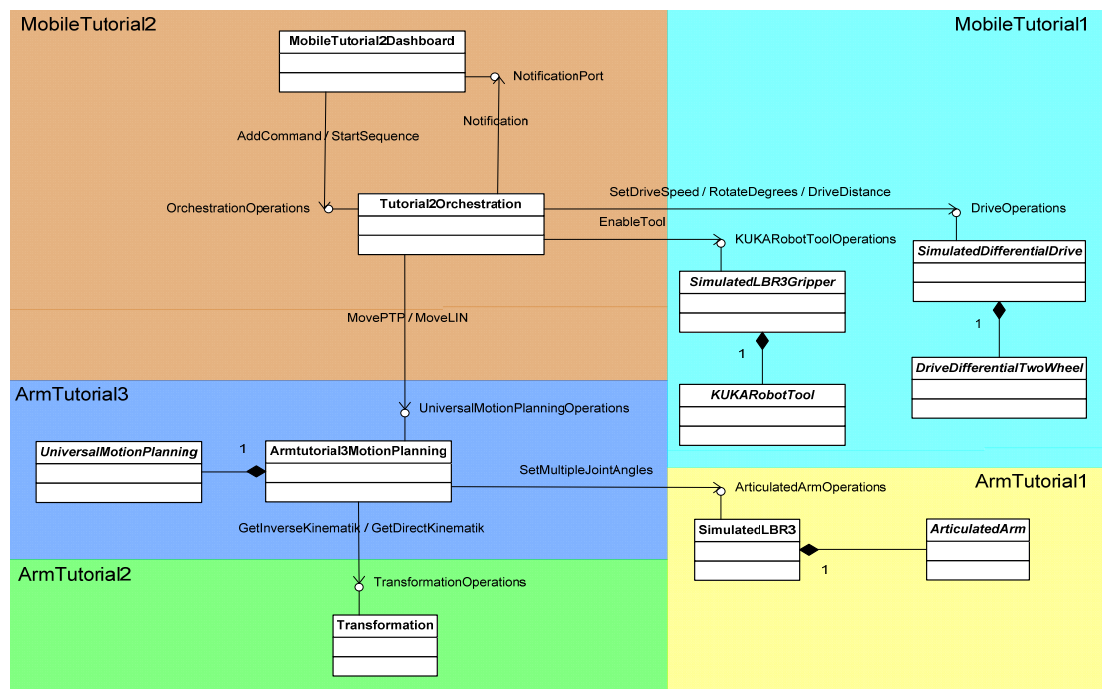
The dashboard offers a 'wait' command to enable time based actions. For example: if you want to drive forward for 2 seconds, issue the following commands:

- Forward
- Wait 2 sec
- Stop

## Coding for mobile Tutorial 2:

Much of the coding will look similar as to what has been done in arm tutorial 4. The main difference is that now we also have the platform as well as the gripper connected to the orchestration service. The dashboard still knows only the orchestration service. All commands coming from the dashboard, now including gripper and platform commands as well, are sent to the orchestration service. This leads to a higher level of abstraction and a lower coupling between the services: the GUI needs to know only one service representing the machine as a whole, consisting of the platform, the arm and the gripper.

### Mobile Tutorial 2 Services Overview:



### New services for mobile tutorial 2:

- MobileTutorial2Dashboard: User Interface with an added list control for queued commands
- Tutorial2Orchestration: Gathers and distributes commands for the mobile machine

The diagram shows the two connections between the dashboard and the orchestration service:

- Channel to the orchestration service: Adding commands, deleting commands and start of the current command queue

- Channel *from* the orchestration service: Notifications about changes in the current command queue (e.g. when a command has finished)

Sending commands to the arm service, the platform service and the gripper service is done by the orchestration service, just in the same fashion as they were sent by the dashboard service in mobile tutorial 1.

## Sending commands to the orchestration service

The handler in the following code snippet comes from the **dashboard** service of mobile tutorial 2. It is the handler that processes the request for opening or closing the gripper. Just like it was done for motion requests in arm tutorial 4, the enable/disable tool request is wrapped into a 'DSSPOperationWrapper' class. That way, the request can be packed into a CommandRequest type.

```
/// <summary>
/// changes the state of the gripper
/// </summary>
/// <param name="grip">gripper parameter</param>
/// <returns>tasks to perform</returns>
IEnumerator<ITask> OnChangeGripperHandler(OnChangeGripper gripRequest)
{
    //If the request is to open the gripper, DisableTool will do that
    if (gripRequest.Open)
    {
        tool.DisableTool disableTool = new tool.DisableTool(new tool.DisableToolRequest());
        yield return Arbiter.Choice(
            _orchPort.AddCommand(new orchestration.AddCommandRequest(new util.DsspOperationWrapper(disableTool))),
            delegate(DefaultUpdateResponseType resp)
            {
                },
            DefaultFaultHandler
        );
    }
    //else close the gripper with EnableTool
    else
    {
        tool.EnableTool enableTool = new tool.EnableTool(new tool.EnableToolRequest());
        yield return Arbiter.Choice(
            _orchPort.AddCommand(new orchestration.AddCommandRequest(new util.DsspOperationWrapper(enableTool))),
            delegate(DefaultUpdateResponseType resp)
            {
                },
            DefaultFaultHandler
        );
    }
}
```

## Executing gripper commands from the command queue

For further explanation of how the command queue is executed in general, please also read the section 'Executing the command queue' of arm tutorial 4.

The processing of the tool request is found inside the 'StartHandler' of the **orchestration** service.

```

public IEnumerator<ITask> StartHandler(Start start)
{
    . . .

    //process message to tool service
    else if (op.Body is Util.ToolRequest)
    {
        success = _toolPort.TryPostUnknownType(op);

        //Handle the result, especially the fault
        PortSet<DefaultUpdateResponseType, Fault> resp = op.ResponsePort as PortSet<DefaultUpdateResponseType,
        yield return Arbiter.Choice(
            resp,
            delegate(DefaultUpdateResponseType umpr) { },
            delegate(Fault f)
            {
                notFault.Body.ReceivedFault = f;
                notFault.Body.Commands = not.Body.Commands;
                base.SendNotification<NotificationFault>(_subMgrPort, notFault);
                exceptionInTargetService = true;
            }
        );
    }
}

```

Here you can see how the operation 'op' (of type 'DSSPOperation') is sent to the \_toolPort. Again the command 'TryPostUnknownType' is used to send the operation to the tool port. There the right handler is searched for automatically. If no appropriate handler is found, the function call returns false.

## Executing motion commands from the command queue

The execution of motion commands from the command queue is already part of arm tutorial 4. There is one interesting addition in this tutorial though that warrants another mentioning. In the fault case for sending a motion command to the motion planning service, not only a 'NotificationFault' is sent, but also a 'NotificationMoveFinished' is sent to the **own** notification port for motion planning messages, so that the following code 'Wait for notification' will unblock immediately.

```

//Send message to Motion Planning service
if (op.Body is Util.UniversalMotionPlanningRequest)
{
    success = _mpPort.TryPostUnknownType(op);

    //Handle the result, especially the fault
    PortSet<ump.UniversalMotionPlanningResponse, Fault> resp = op.ResponsePort as
PortSet<ump.UniversalMotionPlanningResponse, Fault>;
    yield return Arbiter.Choice(
        resp,
        delegate(ump.UniversalMotionPlanningResponse umpr)
        { },
        delegate(Fault f)
        {
            notFault.Body.ReceivedFault = f;
            notFault.Body.Commands = not.Body.Commands;
            base.SendNotification<NotificationFault>(_subMgrPort, notFault);
            exceptionInTargetService = true;

            //fake notification to unblock handler
            _mpNot.Post(new ump.NotificationMoveFinished());

        }
    );

    //Wait for notification
    yield return (Arbiter.Receive<ump.NotificationMoveFinished>(
        false, _mpNot, delegate(ump.NotificationMoveFinished not2)
        {
            if (!exceptionInTargetService)
            {
                not.Body.DestAngles = not2.Body.DestinationAngles;
                not.Body.DestPosition = not2.Body.DestinationPoint;
            }
        }
    ));
}

```

This is a more robust solution for the orchestration service since it allows for further processing of other queued commands even if some command failed. The handling of failed commands depends on the type of application though, so here it is up to the programmer of how to proceed.