

## Arm Tutorial 4

### *Arm Tutorial 4 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. KUKATutorial4Orchestration
2. KUKATutorial4Dashboard

#### Column B

1. Util
2. KUKACommandTypes
3. ArticulatedArm (in abstract services)
4. KUKAUniversalMotionPlanning (in abstract services)
5. SimulatedLBR3Arm
6. KUKAArmTutorialSimulation
7. Transformation
8. KUKATutorial3MotionPlanning
9. KUKATutorial4Orchestration
10. KUKATutorial4Dashboard

### ***Queuing motion statements***

Industrial robot programs usually consist of long series of motion commands. These motion commands describe paths that move a tool to its destination positions, e.g. a grip position, a place position, a tool-reload position, or of course process positions like welding spots or gluing paths. Often the paths between process positions and non process positions need a couple of extra points that help maneuver the robotic safely in a very crowded cell.

Even today in industrial environments, path positions for a robot arm are usually recorded manually in a cell-ramp-up phase. For this, the operator brings the robot arm into position and uses a record function to tell the robot how to move to this position within the path. These recording steps generate the list of motion commands. In more modern cases, paths come predefined out of a CAD/CAM system. But even in these cases human operators usually need to adjust path positions to fit the real cell environment, since the

simulation model and the reality never match completely. So also in the half-automated case robot operators need to deal with lists of adjacent motion commands.

In this tutorial we want to show a way of how we can queue motion commands for our simulated robot arm. The challenge in this highly parallel environment is not so much about where to define the queue, but how to wait with the next command until the arm has finished the current command.

The dashboard has now a list control that queues motion commands. After a cartesian or axis-specific target position has been entered, choosing the motion type (PTP or LIN) will add a motion command to the queue.

PTP movement

move to angles  move to point

move to angles

Joint1: 0  
Joint2: 0  
Joint3: 0  
Joint4: 0  
Joint5: 0  
Joint6: 0  
Joint7: 0

move to point

Position:  
x = 0,2  
y = 0,2  
z = 0,2

Orientation:  
r = 0  
p = 0  
y = 0

add async PTP   add sync PTP   add fully sync PTP   add LIN

Command Queue

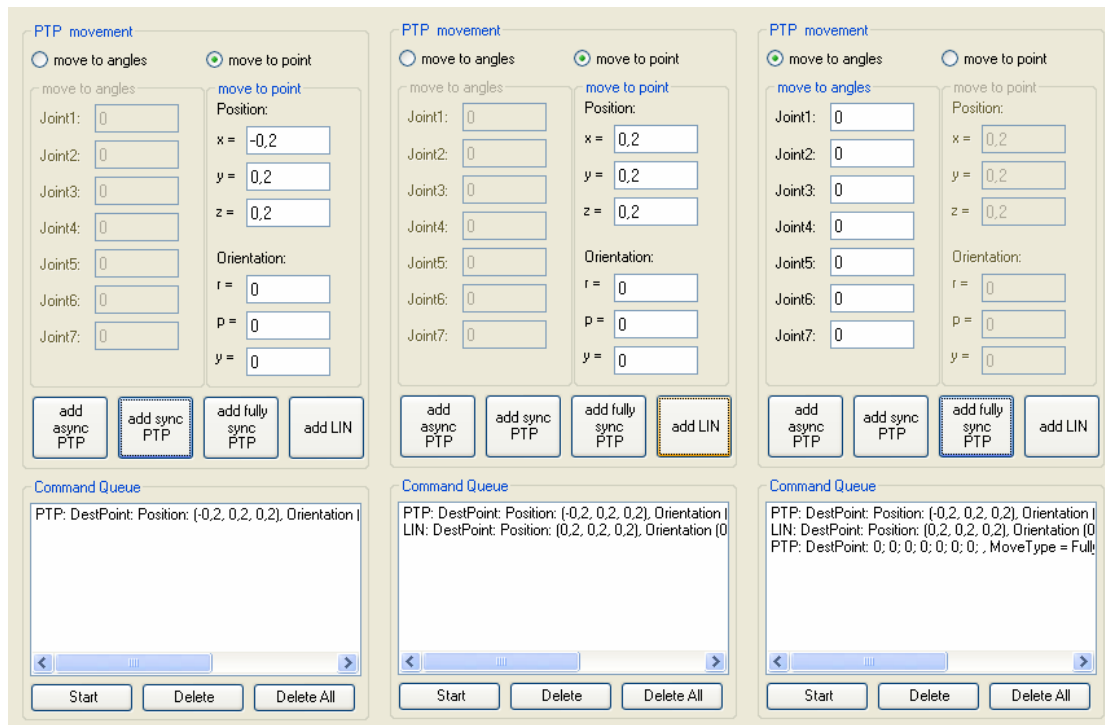
PTP: Destination: Position: (0,2, 0,2, 0,2), Orientation

Start   Delete   Delete All

Unlike the previous tutorials, the robot will only move after the Start button triggers the execution of motion commands within the queue.

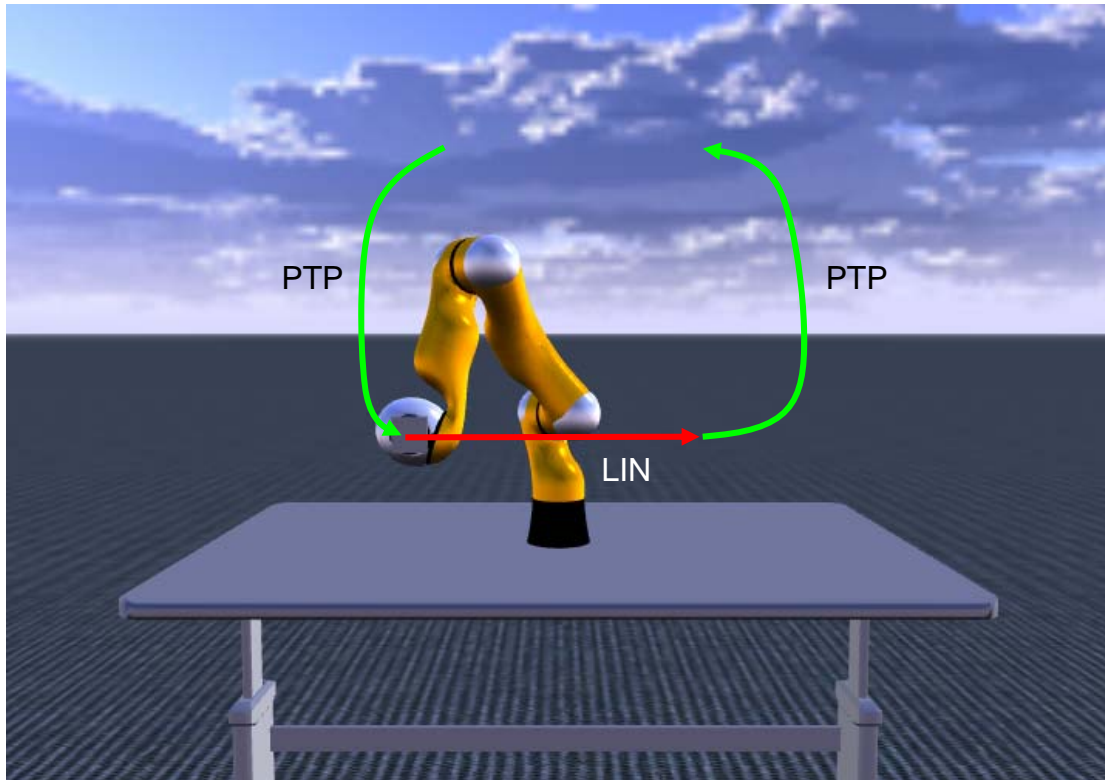
In the following example, we queue three motions:

- 1) PTP motion to the cartesian position (-0,2; 0,2; 0,2);
- 2) LIN motion to the cartesian position (0,2; 0,2; 0,2);
- 3) PTP motion to the axis position (0, 0, 0, 0, 0, 0, 0);

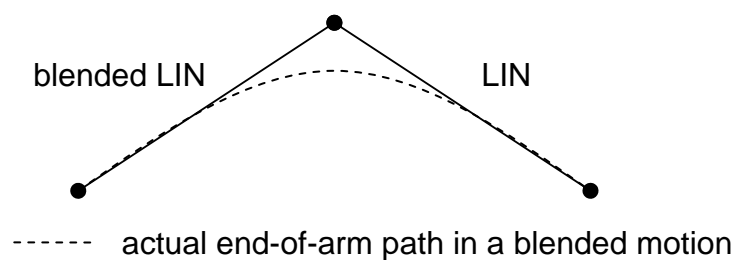


Now we have a very simple path for the arm. It would be nice of course if you could save or load these robot programs. A simple serialization of the command queue would already be a start, but since this is out of scope for this tutorial we did not include this in our source code.

When pressing the start button, the upper most command in the queue is processed. As soon as the motion is finished, the command is taken out of the queue and the next command – if available – is processed. After the processing of each command, the end-of-move notification for the orchestration also contains the current position as axis values and in cartesian coordinates. These are then posted to the dashboard. After finishing the movement, all commands reappear in the queue and they can be executed again.



You will notice that the arm makes a short 'stop' at the end of each motion. This is due to the fact that every motion is planned by itself with its ramps for acceleration and deceleration. Commercial robot controllers usually offer the programmer to define 'blended' motions. When interpreting such a motion, the controller knows that it shouldn't come to a stop at the destination point, but to continue with the next motion without losing too much speed. To enable the arm to execute such a motion, the programmer needs to allow a certain path deviation around the destination point.

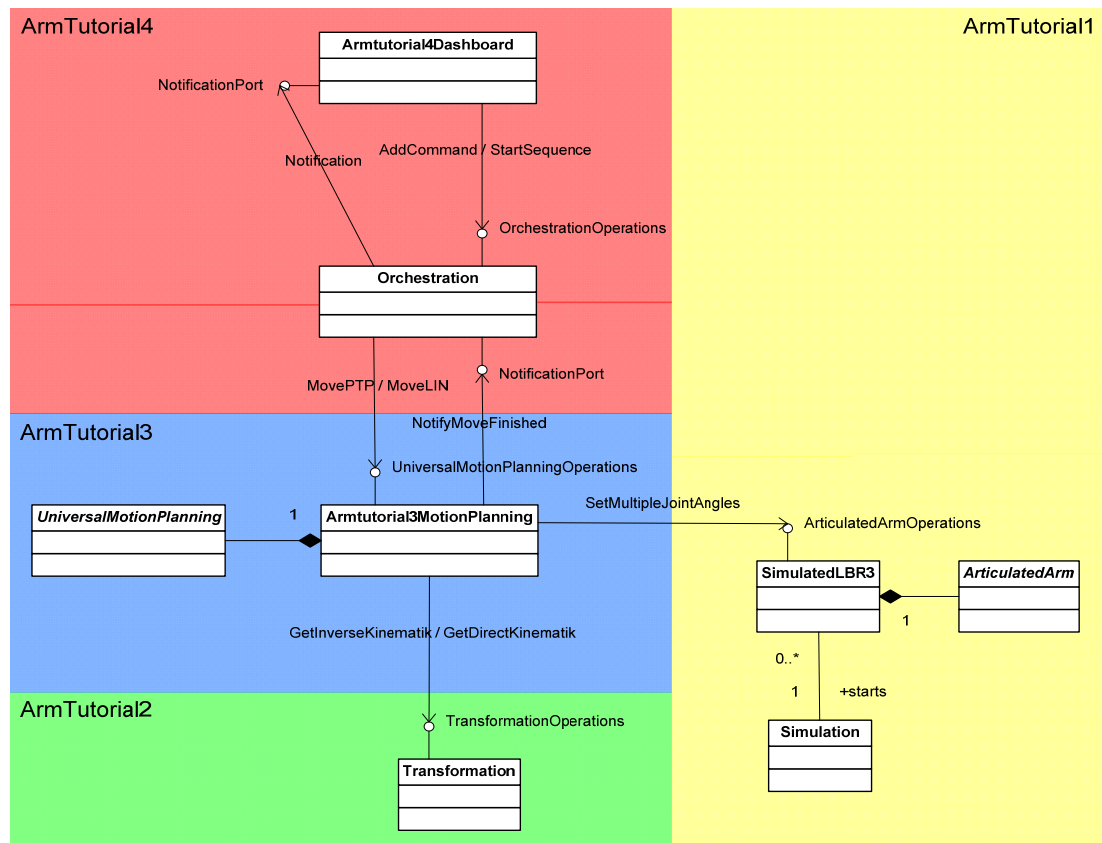


The blending algorithm is also out of scope for this tutorial and might come with a future release. For now, since we offer the programmer to 'look' programmatically at several motion commands in the queue at the same time before motion execution, we encourage our readers to try generating a blending algorithm at this point.

## Coding for Arm Tutorial 4:

All commands coming from the dashboard are sent to the orchestration service. So the only partner the dashboard needs to know is the orchestration service. In this case, it might seem overkill to insert an extra orchestration service between the dashboard and the arm. But already in the second mobile tutorial, when new devices are attached and controlled through the dashboard, the benefits of the orchestration service become clear.

Arm Tutorial 4 Services Overview:



New services for arm tutorial 4:

- **ArmTutorial4Dashboard:** User Interface with an added list control for queued motions
- **Orchestration:** Queues motion commands for the robot arm

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)

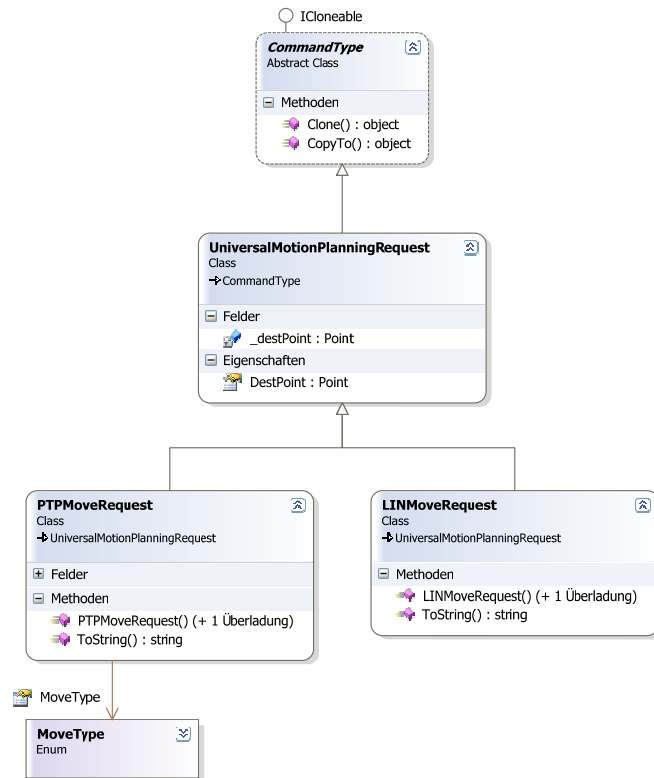
## Sending commands to the orchestration service

In the following code snippet you see how a request for a PTP motion from the **dashboard** service is processed. A new instance of the type PTPMove is wrapped into a 'DSSPOperationWrapper' class. This wrapping is needed to be able to send the PTPMove within the body of a CommandRequest. ("ump" stands for the universal motion planning proxy, where the PTPMove type is defined.)

```
/// <summary>
/// handler for PTP movements to a point defined by angles
/// </summary>
/// <param name="ptp">ptp parameters</param>
/// <returns>tasks to perform</returns>
IEnumerator<ITask> OnDrivePTPHandler(OnDrivePTP ptp)
{
    //Generate Operation
    ump.PTPMove op = new ump.PTPMove();
    ump.PTPMoveRequest req = new ump.PTPMoveRequest();
    req.DestPoint = ptp.DestPoint;
    req.MoveType = ptp.Type;
    op.Body = req;

    //Send operation to orchestration service
    yield return Arbiter.Choice(
        _orcMainPort.AddCommand(new orch.AddCommandRequest(new util.DsspOperationWrapper(op))),
        delegate(DefaultUpdateResponseType resp)
        {
            LogInfo("Successful Response received");
        }
    );
    DefaultFaultHandler
};
yield break;
}
```

The PTPMove object is ultimately derived from the abstract class 'CommandType'. This base class type is used for the command queue itself, so that all incoming command requests can be handled the same within the orchestration service.



## Filling the command queue

The command queue is defined as a List of DsspOperations.

```

/// <summary>
/// Kukatutorial 4orchestration State
/// </summary>
[DataContract()]
public class Kukatutorial4orchestrationState
{
    List<DsspOperation> _operations = new List<DsspOperation>();

    public List<DsspOperation> Operations
    {
        get { return _operations; }
        set { _operations = value; }
    }
}
  
```

Any incoming command is processed in the AddCommandHandler. It is required for each command to be derived from “CommandType”. If this is the case, the command is added to the command queue (‘\_state.Operations’) and is inserted into a notification object (‘not’), which holds all current commands in a list. This List is then sent back to the Dashboard through the subscriber mechanism, so that the Dashboard can now update its queue display with the current remaining commands in the queue.

```

/// <summary>
/// Handler for adding commands to the queue
/// </summary>
/// <param name="command">command parameters</param>
/// <returns>tasks to perform</returns>
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask> AddCommandHandler(AddCommand command)
{
    //Only add CommandTypes to the OperationsToDo List and the notification message
    if (command.Body.Operation.Operation.Body is commands.CommandType)
    {
        _state.OperationsToDo.Add(command.Body.Operation.Operation);
        not.Body.Commands.Add(command.Body.Operation.Operation.Body as commands.CommandType);
    }
    else throw new Exception("Command type not supported!");

    command.ResponsePort.Post(DefaultUpdateResponseType.Instance);

    //update notification and send it
    not.Body.DestAngles = null;
    not.Body.DestPosition = null;
    not.Body.SequenceFinished = false;
    base.SendNotification<Notification>(_subMgrPort, not);

    yield break;
}

```

## Executing the command queue

As soon as a “Start” message arrives at the **orchestration** service, the start message handler will process the command queue. As long as commands are in the queue, the first command is examined to see the general type of the command. Depending on the type, the orchestration service decides on where to forward this command. In the following code snippet, you can see that the `op.Body` is checked for being a ‘UniversalMotionPlanningRequest’. If that is the case, the command is sent to the Motionplanning port (`_mpPort`), and two anonymous handlers are activated for a fault or a success response.

The command ‘TryPostUnknownType’ is used because the operation that is sent is of the general type ‘DSSPOperation’. ‘TryPostUnknownType’ will check by itself if any of the main port handlers of the target service can handle this operation. If not, this function call returns ‘false’.

```

while (_state.Operations.Count > 0)
{
    op = _state.Operations[0];

    //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;
            }
        );
    }
    if (exceptionInTargetService)
        yield break;
    if (!success) LogError("Command not accepted by receiver!");

    //update state and notification
    _state.Operations.RemoveAt(0);
    _state._finishedOperations.Add(op);
    not.Body.Commands.RemoveAt(0);

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

//After finishing the sequence, reset the motion commands and send notification
foreach (DsspOperation operation in _state._finishedOperations)
{
    _state.OperationsToDo.Add(operation);
    not.Body.Commands.Add((commands.CommandType)operation.Body);
}
_state._finishedOperations.Clear();
not.Body.SequenceFinished = true;
base.SendNotification<Notification>(_subMgrPort, not);
not.Body.SequenceFinished = false;

yield break;
}

```

Now that the motion planning service has answered the sent command with either a success or a fault message, the code waits for a notification that indicates that the motion is finished. Therefore a handler for the notification 'NotificationMoveFinished' is activated.

If the motion planning service answered the command positively with a 'UniversalMotionPlanningResponse', it will send a 'NotificationMoveFinished' as soon as the motion has finished execution by the arm. Then the destination values of this position are written into a notification object from the orchestration service. This notification then is sent to all subscribers. In our code, this notification is received by the dashboard service that now shows the new position values of the end of arm. Finally, the original command queue is restored and sent to the GUI.