mh5_robot

Release C.1

Alex Sonea

May 13, 2021

MH5 ROBOT ROS PACKAGES

This repo contains ROS packages for working with the MH5 humanoid robot.

CHAPTER

ONE

PACKAGE DESCRIPTION

1.1 `mh5_hardware` package

This package follows the `ros_control` design model. It contains the highly specific hardware access functions needed for:

- configuring and communicating with the Dynamixel actuators used by the robot
- configuring and reading information from the on-board IMU unit
- (to-be) configuring and retrieving information from the Force Sensitive Resistors (FSRs) in the feet

class MH5DynamixelInterface : **public** RobotHW

Main class implementing the protocol required by `ros_control` for providing access to the robot hardware.

This class performs communication with the servos using Dynamixel protocol and manages the state of these servos. It uses for this purpose Dynamixel SDK (specifically the ROS implementation of it) with the only exception that for port communication it uses a custom subclass of `PortHandler` in order to be able to configure the communication port with RS485 support, because the interface board used by RH5 robot uses SC16IS762 chips that control the flow in heardware, but need tto be connfigured in RS485 mode via `ioctl`.

The class should be instantiated by the `pluginlib` once the main mode is started and initiates the load of the `CombinedRobotHW` class.

The class uses the information from the param server to get details about the communication port configuration and the attached servos. For each dynamixel interface the following parameters are read:

The class registers itself with the `pluginlib` by calling:

```
PLUGINLIB_EXPORT_CLASS(mh5_hardware::MH5DynamixelInterface, hardware_
↪interface::RobotHW)
```

CHAPTER

TWO

PACKAGE REFERENCE

2.1 mh5_hardware reference

2.1.1 Main classes

2.1.1.1 class MH5DynamixelInterface

class mh5_hardware::**MH5DynamixelInterface** : **public** RobotHW

Main class implementing the protocol required by `ros_control` for providing access to the robot hardware.

This class performs communication with the servos using Dynamixel protocol and manages the state of these servos. It uses for this purpose Dynamixel SDK (specifically the ROS implementation of it) with the only exception that for port communication it uses a custom subclass of `PortHandler` in order to be able to configure the communication port with RS485 support, because the interface board used by RH5 robot uses SC16IS762 chips that control the flow in heardware, but need tto be connfigured in RS485 mode via `ioctl`.

The class should be instantiated by the `pluginlib` once the main mode is started and initiates the load of the `CombinedRobotHW` class.

The class uses the information from the param server to get details about the communication port configuration and the attached servos. For each dynamixel interface the following parameters are read:

The class registers itself with the `pluginlib` by calling:

```
PLUGINLIB_EXPORT_CLASS(mh5_hardware::MH5DynamixelInterface, hardware_
↪interface::RobotHW)
```

Public Functions

MH5DynamixelInterface()

Construct a new *MH5DynamixelInterface* object. Default constructor to support `pluginlib`.

~MH5DynamixelInterface()

Destroy the *MH5DynamixelInterface* object. Provided for `pluginlib` support.

bool **init**(ros::NodeHandle &*root_nh*, ros::NodeHandle &*robot_hw_nh*)

Initializes the interface.

Will call the protected methods *initPort()* and *initJoints()* to perform the initialization of the Dynamixel port and the configuration of the joints associated with this interface. If either of these fails it will return false.

Parameters

- **root_nh** – A NodeHandle in the root of the caller namespace.
- **robot_hw_nh** – A NodeHandle in the namespace from which the RobotHW should read its configuration.

Returns true if initialization was successful

Returns false If the initialization was unsuccessful

void **read**(**const** ros::Time &*time*, **const** ros::Duration &*period*)

Performs the read of values for all the servos. This is done through the sync loops objects that have been prepared in *init()*. The caller (the main ROS node owning the hardware) would call this method at an arbitrary frequency that is dictated by it's processing needs (and can be much higher than the frequency with with we need to syncronise the data with the actual servos). For this reason each sync loop is responsible to keep track of it's own processing frequency and skip executing if requests are too often.

In this particular case this method asks the following loops to run:

- Position, Velocity, Load (*pvlReader_*)
- Temperature, Voltage (*tvReader_*)

Parameters

- **time** – The current time
- **period** – The time passed since the last call to *read*

void **write**(**const** ros::Time &*time*, **const** ros::Duration &*period*)

Performs the write of position, velocity profile and acceleration profile for all servos that are marked as present. Assumes the servos have already been configured with velocity profile (see Dyanamixel manual https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/#what-is-the-profile). Converts the values from ISO (radians for position, rad / sec for velocity) to Dynamixel internal measures. Uses a Dynamixel SyncWrite to write the values to all servos with one communication packet.

Parameters

- **time** – The current time
- **period** – The time passed since the last call to *read*

Protected Functions

bool **initPort**()

Initializes the Dynamixel port.

Returns true if initialization was successfull

Returns false if initialization was unsuccessfull

bool **initJoints**()

Initializes the joints.

Returns true

Returns false

template<class **Loop**>
Loop ***setupLoop**(std::string *name*, **const** double *default_rate*)

Convenience function that constructs a loop, reads parameters "rates/<loop_name>" from parameter server or, if not found, uses a default rate for initialisation. It also calls prepare() and registers it communication handle (from getCommStatHandle() with the HW communication status inteface)

Template Parameters **Loop** – the class for the loop

Parameters

- **name** – the name of the loop
- **default_rate** – the default rate to use incase no parameter is found in the parameter server

Returns Loop* the newly created loop object

bool **setupDynamixelLoops** ( )
Creates and initializes all the loops used by the HW interface:

- Read: position, velocity, load (pvl_reader)
- Read: temperature, voltage (tv_reader)
- Write: position, velocity (pv_writer)
- Write: torque (t_writer)

Returns true

Protected Attributes

ros::NodeHandle **nh_**

const char ***nss_**

std::string **port_**

int **baudrate_**

bool **rs485_**

double **protocol_**

mh5_port_handler::*PortHandlerMH5* ***portHandler_**

dynamixel::PacketHandler ***packetHandler_**

mh5_hardware::*PVLReader* ***pvlReader_**
Sync Loop for reading the position, velocity and load.

mh5_hardware::TVReader ***tvReader_**
Sync Loop for reading the temperature and voltage.

mh5_hardware::*PVWriter* ***pvWriter_**
SyncLoop for writing the position and velocity.

mh5_hardware::TWriter ***tWriter_**
SyncLoop for writing the torque status command.

hardware_interface::JointStateInterface **joint_state_interface**

hardware_interface::PosVelJointInterface **pos_vel_joint_interface**

mh5_hardware::*ActiveJointInterface* **active_joint_interface**

mh5_hardware::*CommunicationStatsInterface* **communication_stats_interface**

int **num_joints_**

std::vector<*Joint**> **joints_**

2.1.1.2 class MH5I2CInterface

class mh5_hardware::**MH5I2CInterface** : **public** RobotHW

Main class implementing the protocol required by `ros_control` for providing access to the robot hardware connected on an I2C bus.

This class performs communication with the devices using ioctl.

The class should be instantiated by the `pluginlib` once the main mode is started and initiates the load of the `CombinedRobotHW` class.

The class uses the information from the param server to get details about the communication port configuration and the attached devices. For each device interface the following parameters are read:

...

The class registers itself with the `pluginlib` by calling:

```
PLUGINLIB_EXPORT_CLASS(mh5_hardware::MH5I2CInterface, hardware_
↪interface::RobotHW)
```

Public Functions

MH5I2CInterface()

Construct a new *MH5I2CInterface* object. Default constructor to support `pluginlib`.

~MH5I2CInterface()

Destroy the *MH5I2CInterface* object. Provided for `pluginlib` support.

bool **init**(ros::NodeHandle &*root_nh*, ros::NodeHandle &*robot_hw_nh*)

Initializes the interface.

Will open the system port port and the configuration of the devices associated with this interface. If either of these fails it will return false.

Parameters

- **root_nh** – A NodeHandle in the root of the caller namespace.
- **robot_hw_nh** – A NodeHandle in the namespace from which the RobotHW should read its configuration.

Returns true if initialization was successful

Returns false If the initialization was unsuccessful

void **read**(**const** ros::Time &*time*, **const** ros::Duration &*period*)

Performs the read of values for all the devices. Devices might have specific frequency preferences and would compare the time / period provided with their own to decide if they indeed need to do anything.

Parameters

- **time** – The current time
- **period** – The time passed since the last call to *read*

void **write**(**const** ros::Time &*time*, **const** ros::Duration &*period*)

Performs the write of values for all the devices. Devices might have specific frequency preferences and would compare the time / period provided with their own to decide if they indeed need to do anything.

Parameters

- **time** – The current time

- **period** – The time passed since the last call to *read*

Protected Functions

double **calcLPF**(double *old_val*, double *new_val*, double *factor*)

Protected Attributes

ros::NodeHandle **nh_**

const char ***nss_**

std::string **port_name_**

int **port_**

LSM6DS3 ***imu_**
IMU object.

double **ang_vel_**[3] = {0.0, 0.0, 0.0}
Stores the read velocities from the IMU converted to rad/s.

double **lin_acc_**[3] = {0.0, 0.0, 0.0}
Stores the read accelerations from the IMU converted in m/s^2.

double **imu_lpf_** = 0.1
Low-pass filter factor for IMU.

double **imu_loop_rate_**
Keeps the desired execution rate (in Hz) the for IMU.

ros::Time **imu_last_execution_time_**
Stores the last time the IMU read was executed.

std::vector<double> **imu_orientation_** = {0.0, 0.0, 0.0, 1.0}

hardware_interface::ImuSensorHandle **imu_h_**

hardware_interface::ImuSensorInterface **imu_sensor_interface_**

2.1.2 Supporting classes

2.1.2.1 class MH5PortHandler

class mh5_port_handler**::PortHandlerMH5** : **public** PARENT

Public Functions

inline **PortHandlerMH5**(**const** char **port_name*)

inline bool **setRS485**()

2.1.2.2 class Joint

class mh5_hardware::**Joint**

Represents a Dynamixel servo with the registers and communication methods.

Also has convenience methods for creating HW interfaces for access by controllers.

Public Functions

inline Joint()

Default constructor.

void **fromParam**(ros::NodeHandle *&hw_nh*, std::string *&name*, mh5_port_handler::*PortHandlerMH5* **port*, dynamixel::PacketHandler **ph*)

Uses information from the paramter server to initialize the *Joint*.

It will look for the following paramters in the server, under the joint name:

- `id`: the Dynamixel ID of the servo; if missing the joint will be marked as not prosent (ex. present_ = false) and this will exclude it from all communication
- `inverse`: indicates that the joint has position values specified CW (default) are CCW see https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/#drive-mode10 bit 0. If not present the default is `false`
- `offset`: a value [in radians] that will be added to converted raw position from the hardware register to report present position of servos in radians. Conversely it will be substracted from the desired command position before converting to the raw position value to be stored in the servo.

Initializes the jointStateHandle_, jointPosVelHandle_ and jointActiveHandle_ attributes.

Parameters

- **hw_nh** – node handle to the harware interface
- **name** – name given to this joint
- **port** – Dynamixel port used for communication; should have been checked and opened prior by the HW interface
- **ph** – Dynamixel port handler for communication; should have been checked and initialized priod by the HW interface

inline uint8_t **id**()

Returns the Dynamixel ID of the joint.

Returns uint8_t the ID of the joint.

inline std::string **name**()

Returns the name of the joint.

Returns std::string the name of the joint.

inline bool **present**()

Returns if the joint is present (all settings are ok and communication with it was successfull).

Returns true if the joint is physically present

Returns false if the joint could not be detected

inline void **setPresent**(bool *state*)

Updates the present flag of the joint.

Parameters **state** – the desired state (true == present, false = not present)

bool **ping**(**const** int *num_tries*)

Performs a Dynamixel ping to the joint. It will try up to num_tries times in case there is no answer or there are communication errors.

Parameters **num_tries** – how many tries to make if there are no answers

Returns true if the joint has responded

Returns false if the joint failed to respond after num_tries times

void **initRegisters**()

Hard-codes the initialization of the following registers in the joint (see https://emanual.robotis.com/docs/en/dxl/x/xl430-w250/#control-table).

The registers are initialized as follows:

Register	Address	Value	Comments
return delay	9	0	0 us delay time
drive mode	10	4	if no "inverse" mode set
drive mode	10	5	if "inverse" mode set
operating mode	11	3	position control mode
temperature limit	31	75	75 degrees Celsius
max voltage	32	135	13.5 V
velocity limit	44	1023	max velocity
max position	48	4095	max value
min position	52	0	min value

Other registers might be added in the future.

bool **writeRegister**(**const** uint16_t *address*, **const** int *size*, **const** long *value*, **const** int *num_tries*)

Convenience method for writing a register to the servo. Depending on the size parameter it will call write1ByteTxRx(), write2ByteTxRx() or write4ByteTxRx().

Parameters

- **address** – the address of the register to write to
- **size** – the size of the register to write to
- **value** – a value to write; it will be type casted to uint8_t, uint16_t or unit32_t depending on the size parameter
- **num_tries** – number of times to try in case there are errors

Returns true if the write was sucessful

Returns false if there was a communication or hardware error

bool **readRegister**(**const** uint16_t *address*, **const** int *size*, long &*value*, **const** int *num_tries*)

Convenience method for reading a register frpm the servo. Depending on the size parameter it will call read1ByteTxRx(), read2ByteTxRx() or read4ByteTxRx().

Parameters

- **address** – the address of the register to read from
- **size** – the size of the register to read
- **value** – a value to store the read result; it will be type casted to uint8_t, uint16_t or unit32_t depending on the size parameter

- **num_tries** – number of times to try in case there are errors

Returns true if the read was sucessful

Returns false if there was a communication or hardware error

bool **reboot**(**const** int *num_tries*)
Reboots the device by invoking the REBOOT Dynamixel instruction.

Parameters **num_tries** – how many tries to make if there are no answers

Returns true if the reboot was successful

Returns false if there were communication of harware errors

bool **isActive**(bool *refresh* = false)
Returns if the joint is active (torque on).

Parameters **refresh** – if this parameter is true it will force a re-read of the register 64 from the servo otherwise it will report the cached value

Returns true the torque is active

Returns false the torque is inactive

bool **torqueOn**()
Sets torque on for the joint. Forces writing 1 in the register 64 of the servo.

Returns true if the activation was successfull

Returns false if there was an error (communication or hardware)

bool **torqueOff**()
Sets torque off for the joint. Forces writing 0 in the register 64 of the servo.

Returns true if the deactivation was successfull

Returns false if there was an error (communication or hardware)

inline bool **shouldToggleTorque**()
Indicates if there was a command to change the torque that was not yet completed. It simply returns the active_command_flag_ member that should be set whenever a controllers wants to switch the torque status and sets the active_command_.

Returns true there is a command that was not syncronised to hardware

Returns false there is no change in the status

inline void **resetActiveCommandFlag**()
Resets to false the active_command_flag_. Normally used by the sync loops after successful processing of an update.

bool **toggleTorque**()
Changes the torque by writing into register 64 in the hardware using the active_command_ value. If the change is successfull it will reset the active_command_flag_.

Returns true successful change

Returns false communication or harware error

inline bool **shouldReboot**()
Indicates if there was a command to reboot the joint that was not yet completed. It simply returns the reboot_command_flag_ member that should be set whenever a controllers wants to reboot the joint.

Returns true there is a reset that was not syncronised to hardware

Returns false there is no change in the status

inline void **resetRebootCommandFlag**()
Resets to false the reboot_command_flag_. Normally used by the sync loops after successful processing of an update.

inline uint8_t **getRawTorqueActiveFromCommand**()
Produces an internal format for torque status based on a desired command.

Returns uint8_t value suitable for writing to the hardware for the desired torque status.

inline void **setPositionFromRaw**(int32_t *raw_pos*)
Set the position_state_ (represented in radians) from a raw_pos that represents the value read from the hardware. It takes into account the servo's charactistics, and the offset with the formula:

position_state_ = (raw_pos - 2047) * 0.001533980787886 + offset_

Parameters **raw_pos** – a raw position as read from the hardware; this will already contain the "inverse" classification.

inline void **setVelocityFromRaw**(int32_t *raw_vel*)
Set the velocity_state_ (represented in radians/sec) from a raw_vel that represents the value read from the hardware. It takes into account the servo's charactistics with the formula:

velocity_state_ = raw_vel * 0.023980823922402

Parameters **raw_vel** – a raw velocity as read from the hardware; this will already contain the "inverse" classification and is also signed

inline void **setEffortFromRaw**(int32_t *raw_eff*)
Set the effort_state_ (represented in Nm) from a raw_eff that represents the value read from the hardware. It takes into account the servo's charactistics with the formula:

effort_state_ = raw_eff * 0.0014

Parameters **raw_eff** – a raw effort as read from the hardware; this will already contain the "inverse" classification and is also signed

inline void **setVoltageFromRaw**(int16_t *raw_volt*)
Set the voltage_state_ (represented in V) from a raw_volt that represents the value read from the hardware. The method simply divides with 100 and converts to double.

Parameters **raw_volt** – the value of voltage as read in hardware

inline void **setTemperatureFromRaw**(int8_t *raw_temp*)
Set the temperature_state_ (represented in degrees Celsius) from a raw_temp that represents the value read from the hardware. The method simply converts to double.

Parameters **raw_temp** –

inline int32_t **getRawPositionFromCommand**()
Produces an internal format for position based on a desired command position (expressed in radians) using the formula:

result = (position_command_ - offset_) / 0.001533980787886 + 2047

Returns int32_t a value suitable for writing to the hardware for the desired position in position_command_ expressed in radians.

inline uint32_t **getVelocityProfileFromCommand**()
The velocity_command_ indicates the desired velocity (in rad/s) for the execution of the position commands. Since we configure the servo in time profile mode, the command is translated into a desired duration for the execution of the position command, that is after that stored into register 112. For this the method calculates the delta between the desired position and the current position divided by the desired

velocity, obtaining thus the desired duration for the move. The number is then multiplied with 1000 as the harware expect the duration in ms. The full formula for the value is:

result = abs((position_command_ - position_state_) / velocity_command_) * 1000

Returns uint32_t a value suitable for writing to the hardware profile velocity for the desired position in velocity_command_ expressed in radians/s.

inline const hardware_interface::JointStateHandle &**getJointStateHandle**()
Returns the handle to the joint position interface object for this joint.

Returns const hardware_interface::JointStateHandle&

inline const hardware_interface::PosVelJointHandle &**getJointPosVelHandle**()
Returns the handle to the joint position / velocity command interface object for this joint.

Returns const hardware_interface::PosVelJointHandle&

inline const mh5_hardware::*JointTorqueAndReboot* &**getJointActiveHandle**()
Returns the handle to the joint activation command interface object for this joint.

Returns const *mh5_hardware::JointTorqueAndReboot*&

Protected Attributes

std::string **name_**
The name of the joint.

mh5_port_handler::*PortHandlerMH5* ***port_**
The communication port to be used.

dynamixel::PacketHandler ***ph_**
Dynamixel packet handler to be used.

ros::NodeHandle **nh_**
The node handler of the owner (hardware interface)

const char ***nss_**
Name of the owner as a c_str() - for easy printing of messages.

uint8_t **id_**
Servo ID.

bool **present_**
Servo is present (true) or not (false)

bool **inverse_**
Servo uses inverse rotation.

double **offset_**
Offest for servo from 0 position (center) in radians.

double **position_state_**
Current position in radians.

double **velocity_state_**
Current velocity in radians/s.

double **effort_state_**
Current effort in Nm.

double **active_state_**
Current torque state [0.0 or 1.0].

double **voltage_state_**
Current voltage [V].

double **temperature_state_**
Current temperature deg C.

double **position_command_**
Desired position in radians.

double **velocity_command_**
Desired velocity in radians/s.

bool **poistion_command_flag_**
Indicates that the controller has updated the desired poistion / velocity and is not yet syncronised.

double **active_command_**
Desired torque state [0.0 or 1.0].

bool **active_command_flag_**
Indicates that the controller has updated the desired torque state and is not yet syncronised.

bool **reboot_command_flag_**
Controller requested a reboot and is not yet syncronised.

hardware_interface::JointStateHandle **jointStateHandle_**
A handle that provides access to position, velocity and effort.

hardware_interface::PosVelJointHandle **jointPosVelHandle_**
A handle that provides access to desired position and desired velocity.

mh5_hardware::*JointTorqueAndReboot* **jointActiveHandle_**
A handle that provides access to desired torque state.

2.1.2.3 class LSM6DS3

class LSM6DS3 : **public** LSM6DS3Core

Public Functions

LSM6DS3(int *port*, uint8_t *address*)

~LSM6DS3() = default

status_t **initialize**(SensorSettings **pSettingsYouWanted* = NULL)

int16_t **readRawAccelX**(void)

int16_t **readRawAccelY**(void)

int16_t **readRawAccelZ**(void)

int16_t **readRawGyroX**(void)

int16_t **readRawGyroY**(void)

int16_t **readRawGyroZ**(void)

double **readFloatAccelX**(void)

double **readFloatAccelY**(void)

double **readFloatAccelZ**(void)

double **readFloatGyroX**(void)

double **readFloatGyroY**(void)

double **readFloatGyroZ**(void)

int16_t **readRawTemp**(void)

float **readTempC**(void)

float **readTempF**(void)

void **fifoBegin**(void)

void **fifoClear**(void)

int16_t **fifoRead**(void)

uint16_t **fifoGetStatus**(void)

void **fifoEnd**(void)

double **calcGyro**(int16_t)

double **calcAccel**(int16_t)

Public Members

SensorSettings **settings**

uint16_t **allOnesCounter**

uint16_t **nonSuccessCounter**

2.1.3 Syncronization Loops

2.1.3.1 class LoopWithCommunicationStats

class mh5_hardware::**LoopWithCommunicationStats**

Class that wrapps around a Dynaxmiel GroupSync process and can be executed with a given frequency. It also keeps tabs on the communication statistics: total (since the start of the node) number of Dynamixel packs executed, total number of errors encountered, as well as a shorter timeframe count of packets and errors that can be reset and can be used to report "recent" statistics.

The class can produce a *CommunicationStatsHandle* for the registering with a controller that can publish these statistics.

Subclassed by *mh5_hardware::GroupSyncRead*, *mh5_hardware::GroupSyncWrite*

Public Functions

inline LoopWithCommunicationStats(**const** std::string &*name*, double *loop_rate*)

Construct a new Communication Stats object.

Initializes the communication statistics to 0 and the last_execution_time_ to the current time.

Parameters

- **name** – will be the name used for the loop when registering with the resource manager
- **loop_rate** – the rate (in Hz) that the loop should execute. The *Execute()* method checks if enough time has passed since last run, otherwise it will not be executed. This permits the loop to be configured to run on a much lower rate than the owner loop.

inline ~LoopWithCommunicationStats ()
Destroy the Communication Stats object.

inline const std::string **getName** ()
Returns the name of the loop. Used for message genration.

Returns const std::string the name of the loop.

inline void **resetStats** ()
Resets the recent statistics. Only the packets_ and errors_ are reset to 0, the total_packets_ and total_errors_ (that keep the cummulative packets since the start of the node) are not affected.

inline void **resetAllStats** ()
Resets all statistics, including the totals.

inline const *CommunicationStatsHandle* &**getCommStatHandle** ()
Returns a `ros_control` resource Handle to the communication statistics. Intendent to be called by the main hardware interface in order to register the loop statistics as a resource with a controller that will publish this statistics.

Returns const *CommunicationStatsHandle*& a `ros_control` resource handle

virtual bool **prepare** (std::vector<*Joint**> *joints*) = 0
Prepare the loop (if necessary) based on the specifics of the loop and the joint information. This should be called only once by the owner of the loop, imidiately after the constructor. The method needs to be implemented in the subclass to perform (or just return a true) whatever is needed for that type of loop.

Parameters **joints** – an array of joints that might be needed in the preparation step

Returns true if the activity was successful

Returns false if there was an error performing the activity

virtual bool **beforeCommunication** (std::vector<*Joint**> *joints*) = 0
This is an activity that needs to be performed each time in the loop just before the communication. This allows the particular implementation of the loop to do activities required before the actual communication.

Parameters **joints** – an array of joints that might be needed in this step

Returns true if the activity was successful

Returns false if there was an error performing the activity

inline bool **Execute** (**const** ros::Time *&time*, **const** ros::Duration *&period*, std::vector<*Joint**> *joints*)
Wraps the actual communication steps so that it takes into account the requested processing rate and keeps track of the communication statistics. If the call to *Execute()* is too early (no enough time has passed since last run to account for the execution rate) the method will simply return true.

If enough time has passed, the method checks first if there was a request to reset the statistics then it will call *resetStats()*. It will then call: *beforeCommunication()* and if this is not successfule it will stop and return false. If the step above is successful it will increment the packets statistics and then call *Communicate()* and check again the result. If this is not successfull it will increment the number of errors and return false. If the communication was successfull it will call *afterCommunication()* and return the result of that processing.

Parameters

- **time** – time to execute the method (typically close to now)
- **period** – the time passed since the last call to this method
- **joints** – an array of joints that need to be processed

Returns true if the processing (including the call to *Communicate()*) was successfull

Returns false the call to *Communicate()* was unsuccessfull

virtual bool **Communicate**() = 0
Virtual method that needs to be impplemented by the subclasses depending on the actual work the loop is doing (reading or writing).

Returns true the communication was successfull

Returns false the communication was not successfull

virtual bool **afterCommunication**(std::vector<*Joint**> *joints*) = 0
This is an activity that needs to be performed each time in the loop just after the communication. This allows the particular implementation of the loop to do activities required after the actual communication (ex. for an read loop to retrieve the data from the response package and store it in the joints attributes).

Parameters **joints** – an array of joints that might be needed in this step

Returns true if the activity was successful

Returns false if there was an error performing the activity

Protected Functions

inline void **incPackets**()
Convenience method to increment the number of packets and total packets.

inline void **incErrors**()
Convenience method to increment the number of errors and total total.

Protected Attributes

double **loop_rate_**
Keeps the desired execution rate (in Hz) the for loop.

ros::Time **last_execution_time_**
Stores the last time the loop was executed.

long **packets_**
Number of packets transmited since last reset.

long **errors_**
Number of errors encountered since last reset.

long **tot_packets_**
Total number of packets transmitted since the start of node.

long **tot_errors_**
Total number of errors encountered since the start of node.

bool **reset_**
Keeps asyncronously the requests (from the controllers) to reset the statistics. The *Execute()* method will check this and if set to true it will reset the statistics.

const *CommunicationStatsHandle* **comm_stats_handle_**
A `ros_control` resource type handle for passing to the resource manager and to be used by the controller that publishes the statistics.

2.1.3.2 class GroupSyncRead

class mh5_hardware::**GroupSyncRead** : **public** *GroupSyncRead*, **public** mh5_hardware::*LoopWithCommunicationStats*

A specialization of the loop using a Dynamixel *GroupSyncRead*. Intended for reading data from a group of dynamixels.

This specialization needs a start address and a data length that the loop will handle, implements the *prepare()* method that calls addParam() for all IDs of joints that are marked as "present" and provides a specific implementation of the *Communicate()* method.

Subclassed by *mh5_hardware::PVLReader*, mh5_hardware::TVReader

Public Functions

inline GroupSyncRead(**const** std::string &*name*, double *loop_rate*, dynamixel::PortHandler **port*, dynamixel::PacketHandler **ph*, uint16_t *start_address*, uint16_t *data_length*)

Construct a new *GroupSyncRead* object which is an extension on a standard dynamixel *GroupSyncRead*.

Parameters

- **name** – the name of the loop; used for messages and for registering resources
- **loop_rate** – the rate the loop will be expected to run
- **port** – the dynamixel::PortHandler needed for the communication
- **ph** – the dynamixel::PacketHandler needed for communication
- **start_address** – the start addres for reading the data for all servos
- **data_length** – the length of the data to be read

virtual bool **prepare**(std::vector<*Joint**> *joints*) **override**

Adds all the joints that are marked "present" to the processing loop by invoking the addParam() methods of the dynamixel::GroupSyncRead. If there are errors there will be a warning printed.

Parameters **joints** – a vector of joints to used in the loop

Returns true if at least one joint has been added to the loop

Returns false if no joints has been suucessfully added to the loop

inline virtual bool **beforeCommunication**(std::vector<*Joint**> *joints*) **override**

Simply returns true. SyncReads do not need any additional preparation before the communication.

Parameters **joints** – an array of joints that might be needed in this step

Returns true always

virtual bool **Communicate**() **override**

Particular implementation of the communication, specific to the *GroupSyncRead*. Calls txrxPacket() of dynamixel::GroupSyncRead and checks the communication result.

Returns true if the communication was successful

Returns false if there was a communication error

2.1.3.3 class GroupSyncWrite

class mh5_hardware::**GroupSyncWrite** : **public** *GroupSyncWrite*, **public** mh5_hardware::*LoopWithCommunicationSta*

A specialization of the loop using a Dynamixel *GroupSyncWrite*. Intended for writing data to a group of dynamixels.

This specialization needs a start address and a data length that the loop will handle, implements the beforeExecute() method that calls addParam() for all IDs of joints that are marked as "present" and provides a specific implementation of the *Communicate()* method.

Subclassed by *mh5_hardware::PVWriter*, mh5_hardware::TWriter

Public Functions

inline GroupSyncWrite(**const** std::string &*name*, double *loop_rate*, dynamixel::PortHandler **port*, dynamixel::PacketHandler **ph*, uint16_t *start_address*, uint16_t *data_length*)

inline virtual bool **prepare**(std::vector<*Joint**> *joints*)

Simply returns true. SyncWrites need to pre-prepare data foar each execution and this is implemented in beforeExecute().

Parameters **joints** – an array of joints that might be needed in this step

Returns true always

inline virtual bool **afterCommunication**(std::vector<*Joint**> *joints*)

Simply returns true. SyncWrites do not need any activities after communication.

Parameters **joints** – an array of joints that might be needed in this step

Returns true always

virtual bool **Communicate**() **override**

Particular implementation of the communication, specific to the *GroupSyncWrite*. Calls txPacket() of dynamixel::GroupSyncWrite and checks the communication result.

Returns true if the communication was successful

Returns false if there was a communication error

2.1.3.4 class PVLReader

class mh5_hardware::**PVLReader** : **public** mh5_hardware::*GroupSyncRead*

Specialization of the *GroupSyncRead* to perform the read of the following registers for XL430 Dynamixel series: present position, present velocity, present load (hence the name PVL).

Public Functions

inline PVLReader(**const** std::string &*name*, double *loop_rate*, dynamixel::PortHandler **port*, dynamixel::PacketHandler **ph*)

Construct a new *PVLReader* object. Uses 126 as the start of the address and 10 as the data_lenght.

Parameters

- **name** – the name of the loop; used for messages and for registering resources
- **loop_rate** – the rate the loop will be expected to run
- **port** – the dynamixel::PortHandler needed for the communication

- **ph** – the dynamixel::PacketHandler needed for communication

virtual bool **afterCommunication**(std::vector<*Joint**> *joints*) **override**

Postprocessing of data after communication, specific to the position, velocity and load registers. Unpacks the data from the returned response and calls the joints' setPositionFromRaw(), setVelocityFromRaw(), setEffortFromRaw() to update them. If there are errors there will be ROS_DEBUG messages issued but the processing will not be stopped.

Parameters **joints** –

Returns true

Returns false

2.1.3.5 class PVWriter

class mh5_hardware::**PVWriter** : **public** mh5_hardware::*GroupSyncWrite*

Specialization of the *GroupSyncWrite* to perform the write of the following registers for XL430 Dynamixel series: goal position, goal velocity (profile), (hence the name *PVWriter*). The *Joint* object handles the conversion of commands (position, velocity) into (position, velocity profile) needed to control dynamixel XL430s in velocity profile mode.

Public Functions

inline PVWriter(**const** std::string &*name*, double *loop_rate*, dynamixel::PortHandler **port*, dynamixel::PacketHandler **ph*)

Initializes the writer object with start address 108 and 12 bytes of information to be written (4 for position, 4 for velocity profile and 4 for acceleration profile)

Parameters

- **name** – the name of the loop
- **loop_rate** – the rate to be executed
- **port** – the Dynamixel port handle to be used for communication
- **ph** – the Dynamixel protocol handle to be used for communication

virtual bool **beforeCommunication**(std::vector<*Joint**> *joints*) **override**

For each joint retrieves the desired position and velocity profile (determined internally by the *Joint* class from the velocity command) and prepares a data buffer with the 12 bytes needed to update the goal position (reg 116), velocity profile (reg. 112) and acceleration profile (reg. 108). Acceleration profile is hard-coded to 1/4 of the velocity profile. Only joints that are "present" are taken into account.

Parameters **joints** – vector of joints for processing

Returns true if there is at least one joint that has been added to the loop

Returns false if no joints were added to the loop

2.1.4 `ros_control` Hardware Interface

2.1.4.1 class JointHandleWithFlag

class mh5_hardware::**JointHandleWithFlag** : **public** JointHandle

Extends the hardware_interface::JointHandle with a boolean flag that indicates when a new command was posted. This helps the HW interface decide if that value needs to be replicated to the servos or not.

Subclassed by *mh5_hardware::JointTorqueAndReboot*

Public Functions

JointHandleWithFlag() = default

inline **JointHandleWithFlag**(**const** JointStateHandle &*js*, double **cmd*, bool **cmd_flag*)

Construct a new *JointHandleWithFlag* object by extending the hardware_interface::JointHandle with an additional boolean flag that indicates a new command has been issued.

Parameters

- **js** – the JointStateHandle that is commanded
- **cmd** – pointer to the command attribute in the HW interface
- **cmd_flag** – pointed to the bool flag in the HW interface that is used to indicate that the value was changed and therefore needs to be synchronized by the HW.

inline void **setCommand**(double *command*)

Overrides the hardware_interface::JointHandle *setCommand()* method by setting the flag in the HW to true to indicate that a new value was storred and therefore it needs to be synchronised after calling the inherited method.

Parameters **command** – the command set to the joint

Private Members

bool ***cmd_flag_** = {nullptr}

Keeps the pointed to the flag in the HW that indicates when value change.

class mh5_hardware::**JointTorqueAndReboot** : **public** mh5_hardware::*JointHandleWithFlag*

Public Functions

JointTorqueAndReboot() = default

inline **JointTorqueAndReboot**(**const** JointStateHandle &*js*, double **torque*, bool **torque_flag*, bool **reboot_flag*)

inline void **setReboot**(bool *reboot*)

inline bool **getReboot**()

Private Members

bool *__reboot_flag_ = {nullptr}

2.1.4.2 class ActiveJointInterface

__class ActiveJointInterface__ : __public__ hardware_interface::HardwareResourceManager<*JointTorqueAndReboot*>

Joint that supports activation / deactivation.

To keep track of updates to the HW resource we use and additional flag that is set to true when a new command is issued to the servo. The communication loops will use this flag to determine which servos really need to be syncronised and will reset it once the synchronisation is finished.

2.1.4.3 class CommunicationStatsHandle

__class__ mh5_hardware::__CommunicationStatsHandle__

Public Functions

__CommunicationStatsHandle__() = default

__inline CommunicationStatsHandle__(__const__ std::string &*name*, __const__ long **packets*, __const__ long **errors*, __const__ long **tot_packets*, __const__ long **tot_errors*, bool **reset*)

__inline__ std::string __getName__() __const__

__inline__ long __getPackets__() __const__

__inline__ long __getErrors__() __const__

__inline__ long __getTotPackets__() __const__

__inline__ long __getTotErrors__() __const__

__inline const__ long *__getPacketsPtr__() __const__

__inline const__ long *__getErrorsPtr__() __const__

__inline const__ long *__getTotPacketsPtr__() __const__

__inline const__ long *__getTotErrorsPtr__() __const__

__inline__ void __setReset__(bool *reset*)

Private Members

std::string __name_

__const__ long *__packets_ = {nullptr}

__const__ long *__errors_ = {nullptr}

__const__ long *__tot_packets_ = {nullptr}

__const__ long *__tot_errors_ = {nullptr}

bool *__reset_ = {nullptr}

2.1.4.4 class CommunicationStatsInterface

class CommunicationStatsInterface : **public** hardware_interface::HardwareResourceManager<*CommunicationStatsHa*

2.2 mh5_controllers reference

2.2.1 class ActiveJointController

class mh5_controllers::**ActiveJointController** : **public** controller_interface::Controller<mh5_hardware::*ActiveJo*

Controller that can swithc on or off the torque on a group of Dynamixel servos.

Requires mh5_harware::ActiveJointInterfaces to be registered with the hardware interface. Reads "groups" parameter from the param server, which should contain a list of groups that can be toggled in the same time. It is possible to nest groups in each other as long as they build on each other.

Advertises a service `/torque_control/switch_torque` of type `mh5_controllers/ActivateJoint`. The name passed in calls to this service can be individual joints or groups of joints.

```
rosservice call /torque_control/swtich_torque "{name: "head_p", state: true}"
```

of for a group:

```
rosservice call /torque_control/swtich_torque "{name: "head", state: true}"
```

Will simply turn on or off the torque on all the servos associated with the group.

Public Functions

inline ActiveJointController()

Construct a new Active Joint Controller object using a *mh5_hardware::ActiveJointInterface* interface.

inline ~ActiveJointController()

Destroy the Active Joint Controller object. Shuts also down the ROS service.

bool **init**(mh5_hardware::*ActiveJointInterface* **hw*, ros::NodeHandle &*n*)

Initializes the controller by reading the joint list from the parameter server under "groups". If no parameter is provided it will create a group "all" and assign all avaialable resources to this group. If groups are defined then they should be first listed in the "groups" parameter, then each one of them should be listed separately with the joints, or subgroups that are included. If subgroups are used they have to be fully defined first, befire they are used in a superior group.

This function also advertises the ROS service: /[controller name]/switch_torque

Parameters

- **hw** – the hardware interface that will provide the access to the repoces
- **n** – the nodehandle of the initiator controller

Returns true if there is at least one joint that has been successfully identified and registered with this controller

Returns false if either no "joints" parameter was available in the param server or no joints has been successfully retrieved from the hardware interface.

inline void **starting**(**const** ros::Time &*time*)

Does nothing in this case. Used for completing the controller interface.

Parameters **time** –

void **update**(**const** ros::Time&, **const** ros::Duration&)
Does the actual update of the joints' torque activation member. Please note that this controller only sets the field as provided by the *mh5_hardware::ActiveJointInterface* and it is not actually triggering any communication with the actual servos. It is the hardware interface respoonsibility to replicate this requests to the device.

Private Functions

bool **torqueCB**(mh5_controllers::ActivateJoint::Request &*req*, mh5_controllers::ActivateJoint::Response &*res*)
Callback for processing "switch_torque" calls. Checks if the requested group exists or if there is a joint by that name.

Parameters

- **req** – the service request; group/joint name + desired state
- **res** – the service response; if things are successful + detailed message

Returns true always

bool **rebootCB**(mh5_controllers::ActivateJoint::Request &*req*, mh5_controllers::ActivateJoint::Response &*res*)
Callback for processing "reboot" calls. Checks if the requested group exists or if there is a joint by that name.

Parameters

- **req** – the service request; group/joint name + desired state
- **res** – the service response; if things are successful + detailed message

Returns true always

Private Members

std::map<std::string, std::vector<mh5_hardware::*JointTorqueAndReboot*>> **joints_**
Map group->list of joint handles.

realtime_tools::RealtimeBuffer<ActivateJoint::Request> **torque_commands_buffer_**
Holds torque activation commands to be processed during the *update()* processings. The service callbacks only store "true" or "false" in this buffer depending on the command processed.

realtime_tools::RealtimeBuffer<ActivateJoint::Request> **reboot_commands_buffer_**
Holds reboot commands to be processed during the *update()* processings. The service callbacks only store "true" or "false" in this buffer depending on the command processed.

ros::ServiceServer **torque_srv_**
ROS Service that responds to the "switch_torque" calls.

ros::ServiceServer **reboot_srv_**
ROS Service that responds to the "reboot" calls.

2.2.2 class ExtendedJointTrajectoryController

class mh5_controllers::**ExtendedJointTrajectoryController** : **public** controller_interface::MultiInterfaceCon

Public Functions

inline ExtendedJointTrajectoryController()

bool **init**(hardware_interface::RobotHW **robot_hw*, ros::NodeHandle &*root_nh*, ros::NodeHandle &*controller_nh*)

void **starting**(**const** ros::Time &*time*)

void **stopping**(**const** ros::Time &*time*)

void **update**(**const** ros::Time &*time*, **const** ros::Duration &*period*)

Private Members

mh5_controllers::BaseJointTrajectoryController ***pos_controller_**

mh5_controllers::*ActiveJointController* ***act_controller_**

2.2.3 class CommunicationStatsController

class mh5_controllers::**CommunicationStatsController** : **public** controller_interface::Controller<mh5_hardware

Publishes communication ststistics for all the Dynamixel loops registered in the hardware interface. Requires *mh5_hardware::CommunicationStatsInterface* to access the statistics for all loops. If combined HW interface is used please note that this will get all the loops, across all the physical HW interfaces that the combined HW interface will start.

The messages are publish as diagnostic_msgs::DiagnosticArray under topic "diagnostics". Aggregators can be used to process thsese raw diagnostic messages and publish them to a RobotMonitor.

Public Functions

inline CommunicationStatsController()

Construct a new Communication Stats Controller object; defaults the publish period to 0.0.

bool **init**(mh5_hardware::*CommunicationStatsInterface* **hw*, ros::NodeHandle &*root_nh*, ros::NodeHandle &*controller_nh*)

Initializes the controller. Reads the parameter server "publish_period" [expressed in seconds] and uses it for sheduling the publishing of the communication information. It defaults to 30s if no value is avaialable. Please note that the publishing period is also used to reset the short time communication statistics that are provided by the *mh5_hardware::CommunicationStatsInterface*.

It will setup the realtime publisher and allocate the message structure to accomodate the data from the CommunicationStatsInterface.

Parameters

- **hw** – the hardware providing the loops; could be a Combined HW Interface
- **root_nh** – the top Node Handler
- **controller_nh** – the node handler of the controller; used to access the parameter server

Returns true if controller was initialized sucessfully

void **starting**(**const** ros::Time &*time*)
Resets the last_publish_time_ to the provided time.

Parameters **time** – when the controller was started

void **update**(**const** ros::Time&, **const** ros::Duration&)
Performs the actual publishing of statistics by accesing the inteface data. It will check the last time the message was published and does not do any publish if it is less than publish_period_ desired for these message publishing.

Please note that after the massage is published it invokes the setReset(true) for the CommunicationStatsInterface to reset to 0 the short-term statistics.

virtual void **stopping**(**const** ros::Time&)
Provided for completion of the controller interface.

Private Members

std::vector<mh5_hardware::*CommunicationStatsHandle*> **communication_states_**
Holds the list of handles to all the loops across all the HW interfaces.

std::shared_ptr<realtime_tools::RealtimePublisher<diagnostic_msgs::DiagnosticArray>> **realtime_pub_**
Publisher object.

ros::Time **last_publish_time_**
Keeps the last publish time. Updated every time we publish a new message.

double **publish_period_**
The desired publishing period in seconds for the diagnostoc messages.

2.3 `mh5_director` reference

2.3.1 class Director

class Director(*portfolio_path=None*)
The Director class

Reads script definitions from YAML files and can execute them by passing the pose information to the `dynamixel_control/follow_joint_trajectory` action server.

Listens to the `director/run` topic for commands to execute a script.

load_scripts()
Loads XML definitions from the param server and stores them in the `portfolios` attribute.

setup_services()
Starts the subscriptions.

Director subscribes to: - `director/run` - used to trigger the execution of a script - ...

setup_action_client()
Sets up the subscription to the `dynamixel_control/follow_joint_trajectory` action server.

The function will wait for the action server to become available.

run_script_callback(*msg*)
Callback for `director/run`

The request script should be in the form: <portfolio.script>. Will log errors if the requested portfolio or script in that portfolio doesn't exist.

The information in the script is converted into a JointTrajectoryGoal and passed to the action server. If the `feedback` attribute in the message is True, the script_feedback_callback() will be also submitted to the send_goal() method of the action client. If the `wait` attribute in the message is True the method will wait for the action server to finish before completing.

Parameters **msg** (*RunScript*) – message received

script_feedback_callback (*feedback*)
Provides feedback while running the script.

Parameters **feedback** (*FollowJointTrajectoryFeedback*) – the feedback provided by the action server

2.3.2 class Portfolio

class Portfolio
A portfolio of scripts.

A Portfolio is a collection of scripts that can share certain elements like scenes and poses.

A portfolio is composed of the following elements:

name
the name of the Portfolio

units
Unit of measurements for positions angles. Only `rad` and `deg` allowed and by default it will be `rad` if no attribute is specified in the souce XACRO. :type str

joints
The default list of joints to be used across the scripts in this portfolio. The order of joints is important as all the `positions` sppecifications will assume the same order. A portfolio can use a subset of all the joints of the robot and only the joints specified here will be passed when constructing the communcation messages with the robots' controllers. :type list(str)

duration
The default duration (in seconds) for poses in scenes. If scenes do not specify a `duration=` attribute, they will inherit automatically this duration from the portfolio. :type float

poses
A dictionary of Pose() defined in this portfolio. type dict{name: Pose}

scenes
A dictionary of Scene() defined in this portfolio. type dict{name: Scene}

scripts
A dictionary of Script() defined in this portfolio. type dict{name: Script}

classmethod from_xml (*xml_elem*)
Constructs a `Portfolio` object by reading an XML defintion and parsing it.

Parameters **xml_elem** (*xml.etree.ElementTree.ElementTree*) – The XML element with the structure of the portfolio.

Raises

- **ValueError:** – If the data is incorrect. Additional details are provided in the exception text.

- **AssertionError:** – If attributes are missing or missmatched with the expected ones.

get_script_names()

to_joint_trajectory_goal(*script_name*, *speed*)

2.3.3 class Script

class Script

classmethod from_xml(*xml_elem*, *portfolio*)

2.3.4 class Scene

class Scene

classmethod from_xml(*xml_elem*, *portfolio*)

2.3.5 class Pose

class Pose

A *Pose* groups together the joints and the positions needed to produce a specific robot pose.

name

Pose name

Type str

joints

Joints used by the *Pose*. If the source XACRO did not use `joints=` the *Pose* will inherit by default all the joints defined in the portfolio

Type list of str

positions

Positions for each of the joints associated with this *Pose*. Each position matches the order of joints and is expressed in the unit of measures defined by the portfolio.

Type list of float

classmethod from_xml(*xml_elem*, *portfolio*)

Initializes the *Pose* from an XML element tree structure.

Parameters

- **xml_elem** (*xml.etree.ElementTree.ElementTree*) – The XML element tree that contains the structure of the *Pose*
- **portfolio** (*Portfolio*) – The top *Portfolio* object that ownes this *Pose*

Returns The initiazed *Pose* object.

Return type *Pose*

Raises

- **ValueError** – If values included in the XML are incorrect. More details are provided in the exception text.

- **AssertError** – If certain attributes are not included in the XML. More details are provided in the exception text.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

N

P

R

S

T

U