MH5 Robot

Release D.1

Alex Sonea

May 30, 2021

GETTING STARTED

This is the consolidated documentation for the MH5 Humanoid Robot (Miha).

The section GETTING STARTED will introduce the key *Design Principles* and the robot *Specifications*. This is a good start to have an understanding about the technical capabilities of the robot.

The section HARDWARE includes details about the the standard and customized hardware elements included in the construction of the robot. The information provided relates to:

Table 1: Hardware

Section	Purpose
Actuators	Information about the actuators used in the robot Information about the configuration of the actuators A selection of features important for the usage of the robot
Frames	Details about the frames used in its construction
Raspberry Pi HAT	Details about the Raspberry Pi HAT with more in-depth information about: - the Dynamixel Interface - the TFT Display - the Sound Interface - the IMU and - the ADC used to monitor the voltages
Hot-Swap battery circuits	used to manage the power supply from the two batteries in the feet
FSR Feet	circuits used to provide: - pressure information - voltage and current information related to each of the batteries

The section ROS PACKAGES describes the setup of the ROS Noetic version onto the MH5 main controller, including all dependencies needed for its correct functioning. After this it presents is detail the way the custom MH5 packages are designed and are supposed to be used. The packages are grouped in several repos to support better control over the installation (for example the `mh5_hardware` package is dependent on platform specific drivers and libraries like I2C, Serial, etc. that might not be available on a desktop platform, while the `mh5_monitoring` package makes extensive use of `rqt` plug-ins that are not installed on the robot by choice, instead being intended to be used on a remote desktop that has such support enabled):

Table 2: ROS Packages

Section	Purpose
`mh5_robot`	Meta-package containing all the packages intended for deployment on the robot: - Usage of `mh5_hardware` *package* - Usage of `mh5_controllers` package - Usage of `mh5_ui` package - Usage of `mh5_vision` package
`mh5_common`	Meta-package containing all the packages that can be deployed both on the robot as well as on a remote desktop: - Usage of `mh5_description` package - Usage of `mh5_msgs` package
`mh5_remote`	Meta-package containing all the packages that can be deployed on a remote desktop: - Usage of `mh5_monitor` package

Finally the section REFERENCE contains detail API reference for all packages and classes used in the MH5 ROS packages and is intended to help developers understand in more detail these packages.

CHAPTER

ONE

DESIGN PRINCIPLES

When designing Miha we have considered the following principles:

1. **Affordable**: Traditionally, complex humanoid robots are expensive and difficult to afford. Costs tend to grow exponentially with the size of the robot and, while the costs of electronics follows the same downward trajectory that applies to other consumer electronic products, the mechanical parts do no not exhibit such a trend. We therefore have considered a design that offers enough volume to permit high performance computing at the edge while still minimizing the requirements for the actuators.
2. **Complex but not complicated**: Toy robots are fun but useless when designing complex ML models or robotics frameworks. To provide utility, the robots need to have a degree of complexity that will warrant innovative ML models and robotics frameworks. We believe 22 DoF is a minimum that reflects the need for studying bipedal locomotion and interaction. A good array of sensors (position, effort, vision, sound, etc.) and processing abilities need to complement its high-performing actuators.
3. **Easily serviceable and expandable**: Humanoid robots have a very tough life. Because a lot of the research is still in infancy, accidents happen and robots break quite often. Many of the platforms available on the market are not user serviceable which means significant downtime in research until the robots are returned from service. Miha is designed to be be easily serviced by any user with some minimum technical skills: parts are easily available and standard (Raspberry Pi, Dynamixel Servos) and frames are 3D printable or available as spares. Due to the modular nature of the frames, it is also very easy for users to design and 3D print custom parts that would provide the required functionally for a project at hand.
4. **Open and based on standardized framework**: We aim to release as much as possible from the design of the robot as open-source (hardware and software) and we will encourage the community to contribute and expand this base. We are integrating standard frameworks (ROS, TFLite, etc.) with the robot.

CHAPTER

TWO

SPECIFICATIONS

Rev D.1 (March 2021)

2.1 Dimensions

Table 1: Dimensions

Parameter	Value	Comments
Height	48cm	
Width	51cm	with arms stretched
Depth	20cm	
Weight	2.15Kg	Including batteries
Weight	1.84Kg	Excluding the batteries

2.2 Actuators

Table 2: Actuators

Parameter	Value	Comments
Total DoF	22	
Legs DoF (each)	6	ankle pitch and roll, knee yaw and pitch, hip pitch and roll
Legs Actuators	6 x 2XL430	each leg contains 3 2XL430-W250
Arms Dof (each)	4	shoulder pitch and roll, elbow yaw and pitch
Arms Actuators	8 x XL430	each arm contains 4 XL430-W250
Head DoF	2	pitch and yaw
Head Actuators	1 x 2XL430	one 2XL430-W250

2.3 Power

Table 3: Power

Parameter	Value	Comments
Batteries	2 x 2500mAh	3S LiPo batteries, Batteries are located in the feet and are hot-swap; there is no need to turn off the main controller to change the batteries
External power	2.5mm power jack 12V	Optionally the robot can be powered with a 12V power adapter using a standard 2.5mm barrel jack
Autonomy	3 hours	(preliminary estimates)
Monitoring	voltage ADC	Dynamixel voltage, 5V railing, 3.3V railing

2.4 Electronics

Table 4: Electronics

Parameter	Value	Comments
Main controller	Raspberry Pi	Model 4 4GB RAM
Add on board	Robotics HAT	The board includes: 1. dual high speed dual Dynamixel bus 2. IMU (Gyroscope and Accelerometer) 3. 5V 3A power switch for RPi 4. ADC for monitoring power 5. stereo codec with mics and 2 x 1W output 6. PWM fan control 7. USB to UART converter for console access
Hot-swap circuits	2	Each foot includes a circuit that implements: 1. an ideal diode and allows hot-swap 2. low-voltage alarm 3. emergency shutdown for ultra-low voltage
Display	Adafruit 2.0” IPS display	A 2.0” 320x 240 IPS TFT display connected on SPI with console support
Camera	2	Model HBV-1716HD Max resolution 1920 x 1080 USB connected directly to Raspberry Pi field of view 60 degrees
WiFi	2	Built-in 5Ghz frequency WiFi Second USB dongle Low-latency (5GHz band) Access Point (AP) The second WiFi can connect to an exiting infrastructure DHCP and ip routing
Bluetooth	Builtin	Bluetooth 5.0 BLE Bluetooth keyboard for remote control and interface navigation

2.5 Software

Table 5: Software

Parameter	Value	Comments
OS	Raspbian (Debian Buster)	Using Linux kernel 5.10 Kernel drivers added for: - SC16IS762 (SPI to UART) - ST7789V (TFT display) - WM8960 (sound) - ADS1015 (for voltage monitoring ADC) - fan_control
Software	ROS Noetic	ROS Noetic is installed from source
Custom ROS packages		The following packages are included: - hardware interface - controllers - UI for robot TFT - URDF with support for RViz and Gazebo - "director" package for scripted moves - vision (in progress)

2.6 Future plans

There are a number of exciting upgrades to the platform that we expect to deliver soon:

Table 6: Planed improvements

Area	Improvement
Vision	Updated cameras with 100 degrees FoV and more fps options
Foot Sensor	Soles with 4 force sensing resistors (FSR) Information is exchanged over the Dynamixel bus.
Display	increase size of display to 2.8 inch to improve readability

CHAPTER

THREE

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

FOUR

MH5_HARDWARE REFERENCE

4.1 Main classes

4.1.1 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

bool **initSensors**()

Initializes the sensors.

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**

mh5_hardware::TempVoltInterface **joint_temp_volt_interface**

mh5_hardware::VoltCurrInterface **sensor_volt_curr_interface**

int **num_joints_**

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

int **num_sensors_**

std::vector<*FootSensor**> **foot_sensors_**

4.1.2 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_**

4.2 Supporting classes

4.2.1 class MH5PortHandler

class `mh5_port_handler::PortHandlerMH5` : public PARENT

Public Functions

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

inline bool **setRS485**()

4.2.2 class DynamixelDevice

class `mh5_hardware::DynamixelDevice`

Represents a generic Dyanmixel device.

Subclassed by *FootSensor*, *Joint*

Public Functions

inline **DynamixelDevice**()

Default constructor.

virtual 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 Device.

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

- `id`: the Dynamixel ID of the device; if missing the device will be marked as not present (ex. present_ = false) and this will exclude it from all communication

Parameters

- **hw_nh** – node handle to the hardware interface
- **name** – name given to this device
- **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 device.

Returns uint8_t the ID of the device.

inline std::string **name**()
Returns the name of the device.

Returns std::string the name of the device.

inline bool **present**()
Returns if the device is present (all settings are ok and communication with it was successfull).

Returns true if the device is physically present

Returns false if the device could not be detected

inline void **setPresent**(bool state)
Updates the present flag of the device.

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

bool **ping**(const int num_tries)
Performs a Dynamixel ping to the device. 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 device has responded

Returns false if the device failed to respond after num_tries times

virtual void **initRegisters**() = 0
Hard-codes the initialization of the device. Subclasses must override the method.

bool **writeRegister**(const uint16_t address, const int size, const long value, const int num_tries)
Convenience method for writing a register to the device. 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 from the device. 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

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

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.

Protected Attributes

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

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_**
Device ID.

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

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

4.2.3 class Joint

class `mh5_hardware::Joint` : public *DynamixelDevice*

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.

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

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

virtual void **`initRegisters`**`()` override

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 **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 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 10 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*&

inline const mh5_hardware::TempVoltHandle &**getTempVoltHandle**()

Protected Attributes

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.

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.

mh5_hardware::TempVoltHandle **jointTempVoltHandle_**

4.2.4 class FootSensor

class `mh5_hardware::FootSensor` : public *DynamixelDevice*
Represents a Dynamixel Foot sensor.

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

Public Functions

inline **FootSensor**()
Default constructor.

virtual void **fromParam**(ros::NodeHandle &hw_nh, std::string &name, mh5_port_handler::*PortHandlerMH5* *port, dynamixel::PacketHandler *ph) override
Uses information from the paramter server to initialize the Device.

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

- `id`: the Dynamixel ID of the device; if missing the device will be marked as not present (ex. present_ = false) and this will exclude it from all communication

Parameters

- **hw_nh** – node handle to the hardware interface
- **name** – name given to this device
- **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

virtual void **initRegisters**() override
Hard-codes the initialization of the registers in the foot (see <link to="" the="" documentation>="">).

The registers are initialized as follows:

Register | Address | Value | Comments ——————— | —- | – | ——————————-

Other registers might be added in the future.

inline bool **readRawSensors**()

inline bool **readLPFSensors**()

inline bool **readCalibratedSensors**()

bool **readCalibrationFactors**()

bool **updateCalibrationFactors**()

bool **readPower**()

inline const mh5_hardware::VoltCurrHandle &**getVoltCurrHandle**()

Protected Functions

bool **read4Sensors**(u_int16_t address, FootReading &readings)

Protected Attributes

FootReading **foot_readings_**

Returns the handle to the joint position interface object for this joint.

Returns const hardware_interface::JointStateHandle& Returns the handle to the joint position / velocity command interface object for this joint

Returns const hardware_interface::PosVelJointHandle& Returns the handle to the joint activation command interface object for this joint

Returns const *mh5_hardware::JointTorqueAndReboot*&

FootReading **raw_readings_**

FootReading **lpf_readings_**

CalibrationFactors **calibration_factors_**

double **voltage_**

double **current_**

mh5_hardware::VoltCurrHandle **volt_curr_handle_**

4.2.5 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**

4.3 Syncronization Loops

4.3.1 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 *GroupSyncRead*, *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.

4.3.2 class GroupSyncRead

class `mh5_hardware::GroupSyncRead` : public *GroupSyncRead*, public *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 *PVLReader*, 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

4.3.3 class GroupSyncWrite

class mh5_hardware::**GroupSyncWrite** : public *GroupSyncWrite*, public *LoopWithCommunicationStats*

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 *PVWriter*, 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

4.3.4 class PVLReader

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

4.3.5 class PVWriter

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

4.4 `ros_control` Hardware Interface

4.4.1 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 *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 *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}

4.4.2 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.

4.4.3 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}

4.4.4 class CommunicationStatsInterface

class **CommunicationStatsInterface** : public
hardware_interface::HardwareResourceManager<*CommunicationStatsHandle*>

CHAPTER

FIVE

MH5_CONTROLLERS REFERENCE

5.1 class ActiveJointController

class mh5_controllers::**ActiveJointController** : public controller_interface::Controller<mh5_hardware::*ActiveJointInterface*>

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/switch_torque "{name: "head_p", state: true}"
```

of for a group:

```
rosservice call /torque_control/switch_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_msgs::ActivateJoint::Request &req, mh5_msgs::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_msgs::ActivateJoint::Request &req, mh5_msgs::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<mh5_msgs::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<mh5_msgs::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.

5.2 class ExtendedJointTrajectoryController

class mh5_controllers::**ExtendedJointTrajectoryController** : public controller_interface::MultiInterfaceController<hardware_interface::PosVelJointInterface, mh5_hardware::*ActiveJointInterface*>

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_**

5.3 class CommunicationStatsController

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

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.

CHAPTER

SIX

MH5_UI REFERENCE

6.1 Main classes

6.1.1 class MainUI

class main_ui.**MainUI**

Main UI class that handles the views and switches between them.

The MainUI setups a SnackScreen, determines the size of the available screen and handles the main display loop that processes the hotkeys. An additional hot-key 'q' is provided to quit the loop and close the display.

__init__() → None

Initializes the UI. Allocates the SnackScreen, determines the width and height of the screen and initializes the views.

screen: snack.SnackScreen

The main screen of the UI. It is a SnackScreen.

w: int

The width of the screen.

h: int

The height of the screen.

views: Dict[str, snack.Widget]

The views in the UI. You can use *add_view()* to add them to this dictionary.

current_view: snack.Widget

The current view being shown.

done: bool

Controls the display loop. Will be initialized to *False* and will only be set to *True* by pressing the q hot-key.

add_view(*view: snack.Widget*, *hot_key: str*, *default_view: bool = False*) → None

Adds a view (page) to the dictionay of views. Views are held by their hotkey.

Parameters

- **view** (*Widget or subclass*) – The view (page) to be added. The view must be fully constructed and *view_ui.View.run()* must be possible to be executed on that object.
- **hot_key** (*str*) – The key associated with the view. The main loop will process keys and if they match one of these it will handle the switch to that particular view.
- **default_view** (*bool, optional*) – Marks this view as the default view which means the MainUI will use this to start displaying the interface when executing *run()* for the first time. When you add views to the MainUI the last one that uses the default_view

will overwrite the other ones and that will be the one to be used. If no view is defined as `default_view` the MainUI will use the first item in the list of hot-keys. Because of the way the dictionaries work in Python this might not be the first view added. By default *False*

change_view(*hotkey: str*) → None
Changes a view to the one specified by the hot-key provided.

The method will ask the present view to *view_ui.View.finish()* then will *popWindow()* from the screen. It will assign the view represented in the dictionary by the `hotkey` to the `current_view`, it will ask to *view_ui.View.setup()*, and will setup the hot-keys from that view.

Parameters **hotkey** (*str*) – The hot-key identifying that view.

run() → None
Runs the main loop of the UI. It will activate the `default_view` and then will execute a *view_ui.View.run()* for that view (which for shack means to wait for a key press) then handle the hotkeys by switching the views if they match the ones associated with the views or finish the loop if 'q' was pressed.

6.2 Supporting classes

6.2.1 class View

class view_ui.**View**(*screen: snack.SnackScreen*, *timer: int*, *title: str*)
Base class for a view.

grid: snack.GridForm
The view places all the elements into a *snack.GridForm* object of size 1 x 1. We use a GridForm because this is handling hotkeys and allows to define an automated timer to trigger the refresh of the content.

content: snack.Widget
Is the actual content of the view that is normally produced by invoking *create_content()*.

__init__(*screen: snack.SnackScreen*, *timer: int*, *title: str*) → None
Initializes a new view.

A view uses a *snack.GridForm* as a canvas, that is pinned on the screen provided and displays a title.

The constructor only stores the *screen*, *timer* and *title* in the internal variables. You need to specifically call *setup()* to construct the view. *setup()* will call *create_content()* that normally needs to be overridden by subclasses to present a specific content.

Parameters

- **screen** (*snack.SnackScreen*) – The screen where the view will be positioned.
- **timer** (*int*) – Refresh time for view in milliseconds. This will trigger the *update_content()*.
- **title** (*str*) – Title to be presented on the top of the view.

screen: snack.SnackScreen
The main screen where the view will be posted.

timer: int
The refresh timer (in milliseconds) that the view will use to update the content displayed.

title: str
The title of the view. It is diplayed at the top of the screen.

setup() → None
Builds the view content.

Must be called by the MainUi before starting the view. This creates all the objects of the UI and initializes them. Sets-up a `GridForm` of size 1x1 and calls create_content() to fill the specific content of the view. It also registers the hot keys as are reported by the `hotkeys` property that must be subclassed if the view needs to handle keys.

create_content() → snack.Widget
Should be implemented in subclasses to produce the desired view output.

Returns A snack.Widget that will be included in the grid. Note that it should be one element only and if you need a more complex structure you need to use a GridForm or other classes to contain and structure the elements. Have a look at the implementation of RobotStatusView, CommsStatusView and JointView.

Return type Widget

property hotkeys: List[str]
Returns the keys this view handles. If implemented by subclasses then also `process_hotkey` should be implemented.

update_content() → None
Handles updates to the content of the view. Normally these are triggered by the elapsed timer set up by the *timer* property. Should be implemented in the subclass according to the desired behavior.

process_hotkey(*key: str*) → None
Processes the declared hotkeys. Should be implemented in subclass.

Parameters **key** (*str*) – The key to be processed.

run() → str
Performs a *run()* of the grid.

First calls the *update_content()* to trigger updates to the interface and *refresh()* on the screen object. After that it runs the *run()* of the grid object followed by `process_key()` method to process the hotkey pressed (if any) after which it returns the hot key to the caller program (typically the MainUI) so that the loop there can process it's own hot keys.

Returns The key pressed for the caller program to handle if necessary

Return type str

finish() → None
Provides a way for the view to clear resources before being switched from. For instance views that are displaying information from ROS topics have the chance to unsubscribe from the topics here to save resources.

6.2.2 class NameValueScale

class view_ui.**NameValueScale**(*name: str*, *unit: str*, *grid: snack.GridForm*, *row: int*, *widths: List[int]*, *min_val: float*, *max_val: float*)
A display element that includes a name for the object, a value (+ unit of measure if provided) and a Scale (a horizontal bar graph).

__init__(*name: str*, *unit: str*, *grid: snack.GridForm*, *row: int*, *widths: List[int]*, *min_val: float*, *max_val: float*)
Creates a combined display element in one line with a name, a value and a horizontal bar graph.

Parameters

- **name** (*str*) – The name to be shown on the left side of the display.

- **unit** (*str*) – A string to be shown after the value to denote the unit of measure.
- **grid** (*GridForm*) – The form where the elements are added to
- **row** (*int*) – The row number in the form where the elements will be positioned. All elements are on the same row.
- **widths** (*List[int]*) – A list of width for the elements (name, value, scale)
- **min_val** (*float*) – The minium value that the element will display. Needed to calibrate the bar graph.
- **max_val** (*float*) – The maximum value that the element will display. Needed to calibrate the bar graph.

unit: str
String for units of measure.

name: snack.Textbox
A *snack.TextBox* that will display the name part of the element.

value: snack.Textbox
A *snack.TextBox* that will display the value part of the element.

min_val: float
The minimum value expected for the element to display.

max_val: float
The maximum value expected for the element to display.

range_val: float
The range of the value expected to be displayed. Calculated as *max-val - min_val*.

scale: snack.Scale
The *snack.Scale* that will display the bar graph of the item.

update_value(*value: float*, *format: str = '4.1f'*) → None
Updates the content of the elements based on the provided value.

Parameters

- **value** (*float*) – The new value to be displayed. This will be reflected in the value field as well as in the bar graph.
- **format** (*str, optional*) – The format to display the value in the value field, by default '4.1f'

6.2.3 class NameStatValue

class view_ui.**NameStatValue**(*name: str*, *unit: str*, *grid: snack.GridForm*, *row: int*, *widths: List[int]*)
A display element that includes a name for the object, a status and an additional (optional can be '') text.

__init__(*name: str*, *unit: str*, *grid: snack.GridForm*, *row: int*, *widths: List[int]*) → None
Creates a combined display element in one line with a name, a status (+unit of measure if provided) and a value.

Parameters

- **name** (*str*) – The name to be shown on the left side of the display.
- **unit** (*str*) – [A string to be shown after the value to denote the unit of measure.
- **grid** (*GridForm*) – [description]

- **row** (*int*) – The row number in the form where the elements will be positioned. All elements are on the same row.
- **widths** (*List[int]*) – A list of width for the elements (name, status, value)

unit: str
String for units of measure.

name: snack.Textbox
A *snack.TextBox* that will display the name part of the element.

stat: snack.Textbox
A *snack.TextBox* that will display the status part of the element.

value: snack.Textbox
A *snack.TextBox* that will display the value part of the element.

update_value(*stat: str*, *value: str = ''*) → None
Updates the content of the elements based on the provided value.

Parameters

- **stat** (*str*) – A string showing the status of the element.
- **value** (*str, optional*) – An additional value to be shown after the status, by default ‘’.

6.3 Views

6.3.1 class RobotStatusView

class status_view.**RobotStatusView**(*screen: snack.SnackScreen*, *timer: int*, *title: str = 'Robot Status'*)
View that presents the overview of robot’s hardware (excluding servos).

This version includes the following information: - battery voltage - voltage for 5V railing - voltage for 3.3V railing - battery statistics (on battery for…, battery remaining…); these are claculated

in the code here and are based on monitoring the discharge of the battery and the last time the battery was changed

- processor temperature
- fan status (on, off)
- CPU frequency
- CPU governor
- CPU load (1 minute, 5 minutes, 15 minutes average)
- memory used (in %)
- WiFi AP status (IP address if on)
- WiFi dongle status (IP address is connected to infrastructure)
- LAN status (IP address is connected to infrastructure)

__init__(*screen: snack.SnackScreen*, *timer: int*, *title: str = 'Robot Status'*)
Constructor for the status view.

Initializes the battery statistics.

Parameters

- **screen** (*snack.SnackScreen*) – The screen where the display will be made.
- **timer** (*int*) – [description]
- **title** (*str, optional*) – Title to be printed for the view, by default 'Robot Status'

batt_last_change: float
Keeps the time that the battery was changed last.

batt_last_change_value: float
The last value for the battery voltage when battery was replaced.

batt_last_value: float
Last read battery voltage.

batt_last_estimate: float
Time when the latest estimate about battery life was done.

on_batt_str: str
mm time on battery from last change (or start).

Type String showing hh

rem_batt_str: str
mm time remaining on battery based on last estimate.

Type String showing hh

create_content() → snack.Grid
Creates a snack.Grid that contains the items to be displayed and initializes the values for these elements.

Returns The initialized Grid to be used by MainUI.

Return type snack.Grid

shell_cmd(*command: str*) → str
Convenience function for running a Shell command an returning the result.

Parameters **command** (*str*) – Command to be execcuted (ex. `ifconfig wlan0 | grep "inet "`).

Returns The result of running the command or empty string is errors occurred.

Return type str

read_sysfs(*file: str*) → str
Reads the content of a `sysfs` parameter and returns the value stripped.

The method is provided as a faster alternative to using the `shell_cmd`` ()` because no shell will need to be spun.

Parameters **file** (*str*) – The `sysfs` access (ex. `/sys/class/thermal/thermal_zone0/temp`). Please note that the function does not handle any exceptions, so if the file does not exist or the user does not have authorization to read the value an exception will be raised and needs to be handled by the calling program.

Returns The result of reading that `sysfs` parameter.

Return type str

get_interf_status(*interf: str*) → Tuple[str, str]
Convenience function for getting the status and the IP address of an interface.

Parameters **interf** (*str*) – The name of the interface (ex. `wlan0`)

Returns Returns the status of the interface in the first string as "On" or "Off" and the IP address in the second string if connected or empty string if not connected.

Return type tuple(str, str)

`update_content()` → None
Reads the information for each of the elements in the screen and updates their content.

This is triggered by the timer that is setup by the *view_ui.View* class.

6.3.2 class JointView

INDEX

Symbols

A

B

C

D

F

G

H

L

M

N

O

P

R

S

T

U

V

W