Real-time Control of Robots with ROS

Simon Rasmussen

Kongens Lyngby 2017

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

This thesis explores the various facets in integrating new hardware with existing robot platforms such as ROS and bridging the gap to other robot platforms such Mobotware.

Summary (Danish)

Målet for denne afhandling er at udforske de forskellige problemer der opstår når nyt hardware skal integreres med eksisterende robot platforme så som ROS og hvordan forskellige robot platforme bedre kan integreres med hinanden.

Preface

In today's world an ever growing amount of industrial automation is fueling the need for evermore sophisticated robots. However building and controlling these robots without large abstractions has proven infeasible from an engineering standpoint, much like modern software systems such as Windows, Facebook and Google Search would be practically impossible to implement using only assembly. Abstractions in embedded systems however must carefully consider the real-time aspects and constraints of the system which has proven to be a difficult endeavor. Fortunately most systems can settle for soft real-time constraints by isolating the hard real-time parts to separate systems. This allows developers to take advantage of Linux and all of the abstraction benefits it provides freeing developers to focus on the automation problem they're trying to solve rather.

To facilitate robot programming even larger abstractions exists in the form of robotics middleware that enables reuse of common components and subsystems and sharing these with the community. Notable examples of these includes Robot Operating System (ROS), Player Project, Mobotware and MIRO. The platforms used in this project is described in chapter 1.

When interfacing with middleware special care should still be taken to minimize deadline misses to avoid unacceptable system performance degradation. The overarching goal of this project is therefore to explore how existing systems can be integrated with the ROS middleware while keeping performance and real-time constraints in mind.

This project has taken place in two parts that are only slightly related in that they both interact with ROS.

The first part is improving upon an existing ROS hardware driver for the Universal Robot family of robot arms and is described in chapter 2.

The second part consists of building a software bridge between Mobotware and ROS such the simple scripting language SMR-CL from Mobotware can be used to control robots supported by ROS as described in chapter 3.

Lyngby, 09-June-2017

Simon Rasmussen

Acknowledgements

I would like to thank all of my advisors for their insightful comments:

- Ole Ravn
- Thomas Timm Andersen
- Søren Hansen
- Nils Alex Andersen

Contents

Chapter 1
Platforms

This chapter gives a detailed overview of the various robot platforms and related software used in this project.

1.1 Robot Operating System

Robot Operating System (ROS) is a middleware built on top of the Linux kernel and contains many community provided packages facilitating interaction with various robot platforms, sensors and actuators.

At its core ROS is a distributed message passing system with many predefined message types where a central master service called `roscore` coordinates data streams and logging.
ROS calls message channels for topics and each topic is strongly typed to only accept one kind do message. These topics are published and subscribed to by what ROS defines as nodes where a OS process can run one or many nodes. Nodes connect to `roscore` at startup and declare which topics they publish and/or subscribe to.
For example an odometry node can declare it will publish odometery messages of type "`nav_msgs/Odometry`" to the "`/odom`" topic. A controller node would then subscribe to the odometry and goal topics and from these calculate steering commands of type "`geometry_msgs/Twist`" to be published to the "`/cmd_vel`" topic. A motor node would subscribe to this topic and convert it into native motor commands that gets sent over I2C. A graph of the nodes and topics can be seen in Figure 1.1 where rounded squares represents nodes and edges represent topics.

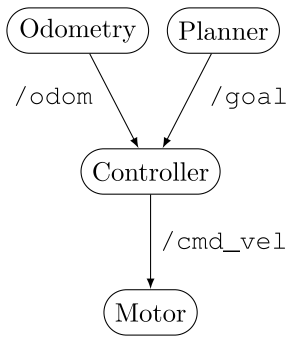


Figure 1.1: Graph of nodes and topics.

1.1.1 Topics, Messages and Services

Each topic published by ROS nodes have a specific message type associated with them, for example "`geometry_msgs/Twist`" for "`/cmd_vel`" as in the example described above. These message types are described with a Domain Specific Language (DSL) called ROS Message Description Language (MDL) to facilitate interoperability between different programming languages with Python and C++ supported by default. MDL is very straightforward and each line simply states a field and its type, furthermore the language contains some simple base types such as `bool`, `intX`, `uintX`, `floatX`, `string`, `time` and `duration` where `X` denotes the type size in bits. Appending `[]` to a field type turns it into an array. Field types can be message types defined by other packages which in turn also can have nested messages, for example the "geometry_msgs/Twist" description can be seen in Listing 1.1 which references "geometry_msgs/Vector3" as seen in Listing 1.2.

```
Vector3  linear
Vector3  angular
```

Listing 1.1: "geometry_msgs/Twist".

```
float64 x
float64 y
float64 z
```

Listing 1.2: "geometry_msgs/Vector3".

Naturally these message descriptions aren't useful on their own and ROS therefore has a tool to generate serialization and deserialization code for C++ and Python and community tools exists for other languages.

Furthermore ROS has a notion of services that can be interacted with from other nodes. These services are essentially Remote Procedure Calls (RPC) and use the same strongly typed message system described above by taking a message of type X as argument and returning another message of type Y. Service calls are for example useful when toggling GPIO pins on the robot, changing runtime parameters or switching control modes.

1.2 Mobotware

Mobotware[BAAR10] is developed by the Automation and Control (AUT) department at DTU and is used in both educational and research contexts. The design is primarily centered around separating the hard and soft real-time parts of robot control while facilitating an easy to extend plugin system. As shown in Figure 1.2 the system is split into 3 parts: Robot Hardware Daemon, Mobile Robot Controller and Automation Robot Servers.

The backbone of Mobotware is the Robot Hardware Daemon (RHD) which is responsible for running hard real-time scheduling and communicating with both hardware sensors and actuators.

To facilitate access of sensor and actuators from the soft real-time side of the system RHD also maintains a variable database that is synchronized with a single master writer and multiple reading clients through sockets. To minimize network communication the whole variable database is only transmitted in its entirety to new clients and from then only kept in sync by sending variable changes. Variables in the database have a direction where "read" direction describes variables provided by sensors and therefore only can be read from out-

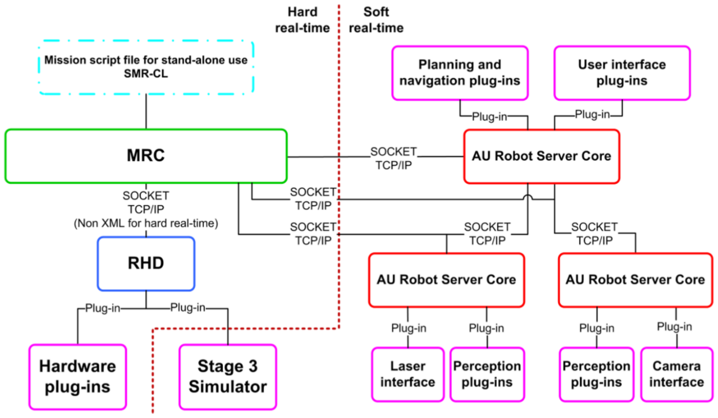


Figure 1.2: Mobotware Design as depicted in[BAAR10].

side of RHD while "write" direction variables can be written to by the master client.

In reality the RHD does not directly interact with any hardware but rather delegates it to plugins. RHD plugins vary in how specialized they are, for example the plugin AUSerial is a generic plugin for communicating with hardware peripherals that use a serial interface and can be fully configured using XML whereas the RFlex plugin is specialized to only work on one robot platform namely the iRobot ATRV-Jr. The RHD is configured using a simple XML file that describes which plugins to load and setting configuration values for these plugins.

The master client mentioned above is in most cases the Mobile Robot Controller (MRC) which has a host of functionality, the primary being motion control and interpreting SMR-CL commands sent to it. SMR-CL is a very simplistic scripting language that is primarily used in a teaching context to program the Small Mobile Robot (SMR) to go through obstacle courses.

However SMR-CL is also used in more advanced use-cases involving Automation Robot Servers (ARS) which primary purpose is motion planning and submits SMR-CL commands to MRC.

1.3 Universal Robots

Universal Robots is a Danish company that produces small industrial robot arms like the UR5 model shown in Figure 1.3. The main advantage of these robots is their safety stop measures allowing them to work in the same space as humans instead of requiring large clearances and cages to prevent bodily harm to humans. Furthermore a similar protective stop measure prevents the arm from colliding with itself. The arm has six degrees of freedom (DOF) while the three different models UR3, UR5 and UR10 have different carrying capacities and reach with higher numbered models being better.

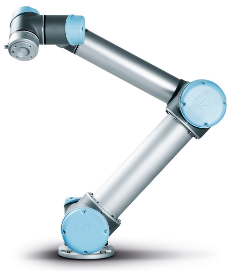


Figure 1.3: UR5 robot arm.

	Primary	Secondary	Real-time
Description	State and Messages	State and Version	Real-time state
Port	30001	30002	30003
Frequency	10 Hz	10 Hz	125 Hz

Figure 1.4: UR Network Interfaces.

The robot can be communicated with over TCP on three different ports as shown in Figure 1.4. The primary and secondary ports share the package format shown in **??** where the first field of the package is a 4 byte unsigned integer describes the entire message size including the size field.

Control of the robot is done by sending URScript commands to the robot via the real-time port. URScript is a python inspired programming language with various built in methods to control the movement, run threads and do socket communication.

1.4 Docker

Docker is a software suite that "containerizes" applications and give them a consistent runtime environment. The basic premise is that software relies on different versions of different libraries which can vary depending on the operating system the software is run and/or compiled on. These libraries in turn have their own dependencies as well which makes it very hard to ensure a consistent operating environment across different systems.

Docker's solution to this problem is to create lightweight images/packages that contain a snapshot of the environment and its dependencies. Images can be built from other images such that only a minimal set of changes made to the parent image must be stored. For example ROS provides docker images for the 4 most recent versions of ROS[ROS] which in this project has been used extensively as a development environment because ROS is only supported on Ubuntu.

Furthermore using docker in this project has been very beneficial because all the library dependencies are explicitly listed in the Dockerfile.

Last but not least having a docker image with all build dependencies of a project has the added benefit of making Continuous Integration (CI) much simpler to setup because most CI services support docker.

Chapter 2

Improving ur_modern_driver

This chapter describes the work done t enao improve the `ur_modern_driver`[Tim] originally written by Thomas Timm as described in [And15].

2.1 Goals

During the beginning of this project a set of improvements for `ur_modern_driver` was discussed with Thomas. Most of the improvements are based on feedback from various people and organizations posted on the project's Github issue page. The final list of desirable improvements can be seen below:

- Improve support for newer versions of UR (Github issue #75)
- Publish joint temperatures (Github issue #81)
- Publish analog tool inputs (Github issue #83)
- Improve safety by implementing a ROS service that must be called to enable the robot and recover from safety stop.
- Improve security by hardening the driver against malicious attacks.
- Improve documentation of the code.
- Improve the usability of the driver as a library that does not depend on ROS.
- Prepare the code as much as possible to allow maintenance and ownership to be transferred to the ROS-Industrial project. This includes various code style and formatting changes, porting to ROS Kinetic, adding unit testing and preparing for continuous integration.

Initially it was thought these changes could be implemented in the existing code, however during the first week of getting familiar with the code it quickly became apparent that the existing code was written in an ad-hoc manner that does not follow modern software development standards. This meant that unit testing would require extensive and very time consuming refactoring and it was therefore decided to rewrite the driver almost entirely, despite the rule of thumb that rewriting software systems from scratch should be avoided.

This is generally considered a good rule of thumb because it can be very difficult to catch all the edge cases that the existing code already has run into and dealt with, especially if the rewrite is done by a different set of people than the original implementation. However, due to the architectural deficits of the existing code as described in section 2.2 a radical shift in architecture was required which justifies a complete rewrite. Furthermore Thomas Timm, the original implementor, overseeing the rewrite helps catching potential edge cases that must be handled properly.

2.2 Existing Solution and Problems

This section will only give a small overview of the existing solution and not go deep into the details of the design because from a software engineering perspective the design is non-existent. A small description of the existing solution can be found in [And15, p. 24-27] but it does not go into much detail either. Instead this section will try to focus on the problems of the existing solution from a software engineering perspective.

The existing `ur_modern_driver` consists of roughly 2700 lines of C++ code, according to the `cloc` CLI tool[AlD], spread across eight source files and seven header files.

As described in [And15, p. 24-27] the driver is split into 4 parts:

- UR state/message parsing handled by `RobotStateRT` and `RobotState`.
  Represented by 🟩 in Figure 2.1.
- Communication with UR in both directions handled by `UrCommunication` and `UrRealtimeCommunication`.
  Represented by 🟨 in Figure 2.1.
- Trajectory execution handled by `UrDriver`.
  Represented by 🟥 in Figure 2.1.
- ROS integration handled by `RosWrapper` and `UrHardwareInterface`.
  Represented by 🟦 in Figure 2.1.

A graph of the relation between these classes and areas can be seen in Figure 2.1 where nodes represent classes and edges represent calls to functions, getters or setters on the class pointed to.

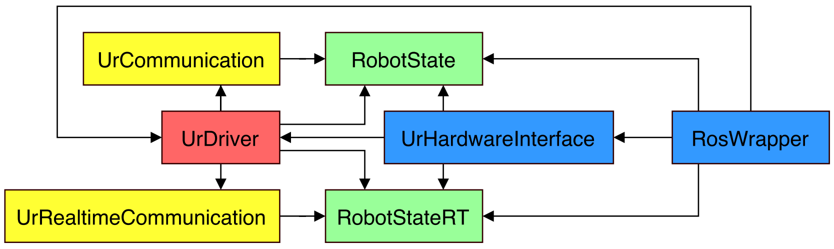


Figure 2.1: `ur_modern_driver` classes and their relations.

As Figure 2.1 shows the existing driver is very complex and does not separate concerns in a good way because every single class accesses at least one function

from the two parsing and state classes RobotState and RobotStateRT. The complexity is further increased by multiple threads accessing the various components making synchronization necessary.
Another increase in complexity not shown in Figure 2.1 comes from the fact some function calls from UrHardwareInterface to UrDriver are directly forwarded to UrCommunication or UrRealtimeCommunication while calls from UrHardwareInterface to RobotState goes through public field accesses of UrDriver to UrCommunication and then to UrHardwareInterface and vice versa for RobotStateRT.

Another issue with the existing driver is code duplication and copy pasting, an example of this can be seen in Listing 2.1 where the code to parse a new version message received from the robot inside the buffer *buf is essentially the same for each field. Practically all parsing in the existing driver functions like this and is not only very verbose but could easily be abstracted.

```
void RobotState::unpackRobotMessageVersion(uint8_t* buf, unsigned int offset, uint32_t len)
{
  memcpy(&version_msg_.project_name_size, &buf[offset], sizeof(version_msg_.project_name_size));
  offset += sizeof(version_msg_.project_name_size);
  memcpy(&version_msg_.project_name, &buf[offset], sizeof(char) * version_msg_.project_name_size);
  offset += version_msg_.project_name_size;
  version_msg_.project_name[version_msg_.project_name_size] = '\0';
  memcpy(&version_msg_.major_version, &buf[offset], sizeof(version_msg_.major_version));
  offset += sizeof(version_msg_.major_version);
  memcpy(&version_msg_.minor_version, &buf[offset], sizeof(version_msg_.minor_version));
  offset += sizeof(version_msg_.minor_version);
  ...
}
```

Listing 2.1: Parsing of RobotState.

Direct code duplication across class can also be spotted as seen in Listing 2.2 and Listing 2.2 which handles converting data of network order to host order.

```
double RobotStateRT::ntohd(uint64_t nf)
{
  double x;
  nf = be64toh(nf);
  memcpy(&x, &nf, sizeof(x));
  return x;
}

```

```
double RobotState::ntohd(uint64_t nf)
{
  double x;
  nf = be64toh(nf);
  memcpy(&x, &nf, sizeof(x));
  return x;
}

```

Many more examples of both direct and indirect duplication exists which makes maintenance of the code a lot more difficult because potential bugs must be fixed in multiple places.

Due to the complicated usage of threads there are also multiple race conditions in the code that can lead to invalid data being published to ROS. An example of such a race condition can be found in RosWrapper::rosControlLoop which calls getToolVectorActual on RobotStateRT and publish the data and only then calls getTcpSpeedActual to publish it. The race condition arises from the fact that while both `getToolVectorActual` and `getTcpSpeedActual` internally locks the whole `RobotStateRT` from being updated during access, `RobotStateRT` is unlocked in between the two calls which means it may be updated in between resulting in data from two different moments being published with the same timestamp. While it's unlikely to happen in practice due to the relatively long delay of 8 milliseconds between state updates the probability increases rapidly with CPU load. Another related threading issue exists in the same loop due to the same lack of a critical region where updates to `RobotStateRT` can entirely be missed if the scheduling of the `rosControlLoop` thread is less than ideal.

All in all the synchronization resembles a poorly implemented producer/consumer queue intermingled with lots of other concerns.

2.3 Design

Before jumping into writing code it's important to explore different architectural designs and consider their pros and cons to arrive at a design that fulfills the goals described above in the best possible manner.

The primary take away from the existing driver is that the majority of the complexity can be abstracted away into multiple producer/consumer queues best described as pipelines, one for the real-time state and another for the regular state updates. Each of these pipelines follow the same overall process where raw data is received from a TCP socket which is then parsed and then published. An example of this real-time pipeline design can be seen in Figure 2.2 where data flows from the robot through the producer resulting in new messages being put into the queue while the consumer awaits new messages and runs them through multiple sub-consumers with varying functionality.

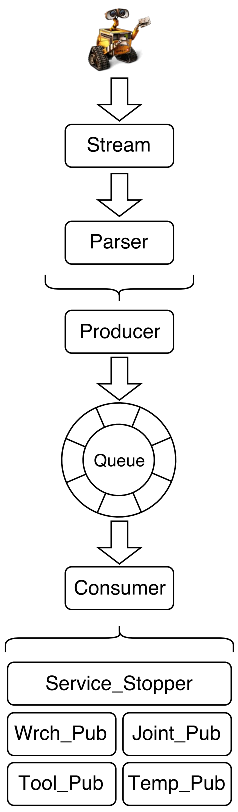


Figure 2.2: RT Pipeline Design.

Ideally to prevent the non-deterministic behavior of heap memory allocation the queue should function as a fixed buffer of memory that parsed messages could be written directly to. However the non real-time pipeline is complicated by the fact that a single data packet from the robot may contain a variable amount of messages that each have different sizes and therefore can't be pre-allocated in a single queue. One solution would be to implement multiple queues and have one for each message type but this complicates the consumer side significantly because it must either balance dequeuing from multiple queues or run one thread per message type.

Instead it was decided to allow a single heap allocation per parsed message which simplifies the pipeline to a single producer and single consumer which operate on an abstract base type representing message. Ownership of the allocated data is handled using C++11 smart pointer `unique_ptr<T>` which ensures memory is freed whenever it goes out of scope. The ownership starts at the producer which explicitly transfer it to the queue and from there is transfered to the consumer at which point it's converted to a `shared_ptr<T>` because multiple sub-consumers borrow access to it and potentially would like to keep it around for later usage, for example until next message to compute a delta between the two.

However the actual data parsing is complicated by the fact multiple versions of each message type exists and each newer version is not strictly backwards compatible in the sense that fields sometimes change ordering, length or data size.

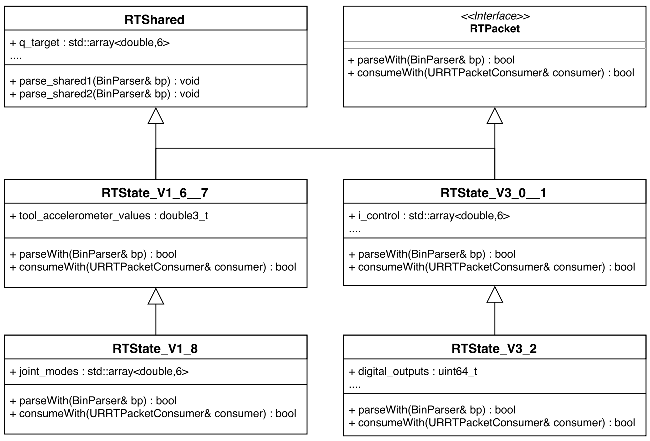

RTShared
+ q_target : std::array<double,6>
....
+ parse_shared1(BinParser& bp) : void
+ parse_shared2(BinParser& bp) : void
<<Interface>>
RTPacket
+ parseWith(BinParser& bp) : bool
+ consumeWith(URRTPacketConsumer& consumer) : bool
RTState_V1_6__7
+ tool_accelerometer_values : double3_t
+ parseWith(BinParser& bp) : bool
+ consumeWith(URRTPacketConsumer& consumer) : bool
RTState_V3_0__1
+ i_control : std::array<double,6>
....
+ parseWith(BinParser& bp) : bool
+ consumeWith(URRTPacketConsumer& consumer) : bool
RTState_V1_8
+ joint_modes : std::array<double,6>
+ parseWith(BinParser& bp) : bool
+ consumeWith(URRTPacketConsumer& consumer) : bool
RTState_V3_2
+ digital_outputs : uint64_t
....
+ parseWith(BinParser& bp) : bool
+ consumeWith(URRTPacketConsumer& consumer) : bool

Figure 2.3: Simplified UML for real-time messages.

One approach to parsing the data is to have one resulting message object that contains all the possible fields for all versions and only assign values to the fields for the version being parsed. This is the approach taken in the existing solution however this becomes quite messy because fields sometimes are reordered between versions and therefore requires multiple if-else chains for each version spread out through the parsing code. Furthermore it complicates the publishing site because it explicitly need to be aware of which message version it is publishing. Over all it complicates maintainability because adding support for a new firmware version touches most parts of the system.

However since all message versions have a common set of fields for which parsing can be shared the problem lends itself well to using Object Oriented Programming (OOP) inheritance. This leads to a design where newer message versions inherit from a base type containing shared fields and parsing functionality. The parser therefore produces the base type and the consumer side then use the visitor pattern to access the non-shared fields of the message version. A simplified UML diagram for the real-time message versions can be seen in Figure 2.3.

The visitor pattern helps to ensure all consumers explicitly handles all versions of messages as this improves maintainability when adding support for new message versions because the compile will complain about unimplemented functions.

2.4 General Improvements

The first and simplest improvement was adopting `clang-format` which is an automated formatting tool for C and C++ code. ROS Industrial suggests[Ind] using the ROS C++ Style Guide which Dave Coleman has kindly implemented as a `.clang-format` file[Col] making the whole process automated. Similarly the ROS naming convention for C++ was also adopted for all new code.

The next improvement made was primarily focused on improving the development process and keeping code quality in check by enabling the GCC flags `"-Wall -Wextra -Wno-unused-parameter"`. These compile flags enable some basic static analysis to avoid undefined behavior and other questionable code constructs.

2.5 Implementation

To highlight the improved maintainability that sensible abstractions can provide compare the version message parsing code from before in Listing 2.1 to Listing 2.2. The new version of the code not only has the same functionality as the previous version but also improves security by preventing buffer overflows which together with the improved readability and maintainability can only be considered an improvement.

```cpp
bool VersionMessage::parseWith(BinParser& bp)
{
  bp.parse(project_name);
  bp.parse(major_version);
  bp.parse(minor_version);
  bp.parse(svn_version);
  bp.consume(sizeof(uint32_t));  // undocumented field??
  bp.parse_remainder(build_date);
  return true;  // not possible to check dynamic size packets
}
```

Listing 2.2: Parsing of `VersionMessage` in new version.

In Listing 2.3 it can be seen how the implementation of `URRTStateParser` is templated and then specialized using `typedef` to provide different parsers depending on the version which is beneficial because adding support for a new version is as simple as adding another `typdef`. Naturally this requires the corresponding `RTState_X` class to be declared for the new version.

```cpp
template <typename T>
class URRTStateParser : public URParser<RTPacket>
{
public:
  bool parse(BinParser& bp, std::vector<std::unique_ptr<RTPacket>>& results)
  {
    int32_t packet_size = bp.peek<int32_t>();
    if (!bp.checkSize(packet_size)) { LOG_ERROR("Buffer len shorter than expected packet length"); return false; }
    bp.parse(packet_size);  // consumes the peeked data
    std::unique_ptr<RTPacket> packet(new T);
    if (!packet->parseWith(bp))
      return false;
    results.push_back(std::move(packet));
    return true;
  }
};

typedef URRTStateParser<RTState_V1_6__7> URRTStateParser_V1_6__7;
typedef URRTStateParser<RTState_V1_8>    URRTStateParser_V1_8;
typedef URRTStateParser<RTState_V3_0__1> URRTStateParser_V3_0__1;
typedef URRTStateParser<RTState_V3_2__3> URRTStateParser_V3_2__3;
```

Listing 2.3: Templated URRTStateParser.

In order to make the pipeline creation as simple as possible a factory/utility class was implemented to synchronously connect to the robot, fetch the first message which is always a VersionMessage, save the result and then use the retrieved version to construct the right parser version as shown in Listing 2.4

```cpp
class URFactory : private URMessagePacketConsumer
{
private:
  bool consume(VersionMessage& vm) { major_version_ = vm.major_version; minor_version_ = vm.minor_version; return
       true; }
public:
  URFactory(std::string& host) : stream_(host, UR_PRIMARY_PORT)
  {
    URProducer<MessagePacket> prod(stream_, parser_);
    std::vector<unique_ptr<MessagePacket>> results;
    prod.setupProducer();
    if (!prod.tryGet(results) || results.size() == 0)
    {
      LOG_FATAL("No version message received, init failed!"); std::exit(EXIT_FAILURE);
    }

    for (auto const& p : results)
      p->consumeWith(*this);

    if (major_version_ == 0 && minor_version_ == 0)
    {
      LOG_FATAL("No version message received, init failed!"); std::exit(EXIT_FAILURE);
    }

    prod.teardownProducer();
  }

  std::unique_ptr<URParser<RTPacket>> getRTParser()
  {
    if (major_version_ == 1)
    {
      if (minor_version_ < 8)
        return std::unique_ptr<URParser<RTPacket>>(new URRTStateParser_V1_6__7);
      else
        return std::unique_ptr<URParser<RTPacket>>(new URRTStateParser_V1_8);
    }
    else
    {
      if (minor_version_ < 3)
        return std::unique_ptr<URParser<RTPacket>>(new URRTStateParser_V3_0__1);
      else
        return std::unique_ptr<URParser<RTPacket>>(new URRTStateParser_V3_2__3);
    }
  }
};
```

Listing 2.4: Partial code of URFactory.

2.6 Unit Testing

Unit testing in ROS is implemented using gtest. Basic unit testing of the message parsing for all versions of RobotModeData, RTState and MasterBoardData was implemented.

Each version of each message has a testRandomDataParsing test case which generates random bytes that are parsed and then checked to ensure the parsing order is correct, an example of the code to do this is shown in Listing 2.5.

```
TEST(RTState_V1_8, testRandomDataParsing)
{
  RandomDataTest rdt(812);
  BinParser bp = rdt.getParser(true);
  RTState_V1_8 state;
  EXPECT_TRUE(state.parseWith(bp)) << "parse() returned false";

  ASSERT_EQ(rdt.getNext<double>(), state.time);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.q_target);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.qd_target);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.qdd_target);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.i_target);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.m_target);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.q_actual);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.qd_actual);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.i_actual);
  ASSERT_EQ(rdt.getNext<double3_t>(), state.tool_accelerometer_values);
  rdt.skip(sizeof(double) * 15);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.tcp_force);
  ASSERT_EQ(rdt.getNext<cartesian_coord_t>(), state.tool_vector_actual);
  ASSERT_EQ(rdt.getNext<cartesian_coord_t>(), state.tcp_speed_actual);
  ASSERT_EQ(rdt.getNext<uint64_t>(), state.digital_inputs);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.motor_temperatures);
  ASSERT_EQ(rdt.getNext<double>(), state.controller_time);
  rdt.skip(sizeof(double));  // skip unused value
  ASSERT_EQ(rdt.getNext<double>(), state.robot_mode);
  ASSERT_DOUBLE_ARRAY_EQ(rdt.getNext<double>(), state.joint_modes);

  EXPECT_TRUE(bp.empty()) << "Did not consume all data";
}
```

Listing 2.5: Unit testing of RTState_V1_8.

The RandomDataTest class uses the same BinParser class as the various message parsers to parse the random data which means the testRandomDataParsing test cases only ensure correct parsing order and that the whole buffer is consumed.

```
TEST(MasterBoardData_V1_X, testTooSmallBuffer)
{
  RandomDataTest rdt(10);
  BinParser bp = rdt.getParser();
  MasterBoardData_V1_X state;
  EXPECT_FALSE(state.parseWith(bp)) << "parse() should fail when buffer not big enough";
}
```

Listing 2.6: Unit testing of MasterBoardData_V1_X.

Furthermore each version of each message has another test case called testTooSmallBuffer that ensures all parsing checks buffer sizes to prevent buffer overflows, an example of such a test can be seen in Listing 2.6.

Testing of the publishing subsystem was out of the scope of this project because time constraints prohibited further investigation into the advanced area of unit testing ROS nodes and topics.

2.7 System Testing

Reaching 100% code coverage with unit testing was out of the scope for this project however to ensure the driver works as expected, system testing was performed against version 1.8 and 3.3.4 of URSim by moving the robot arm in the simulator and manually comparing that the published joint states matches what the simulator numbers.

Unfortunately URSim did not perform exactly like the real robot and varied significantly in message publishing rates ranging from 108 to 120 Hz and oscillating over time. For this reason direct comparison of how the two drivers perform during trajectory execution was only done using a physical robot as described in section 2.8.

2.8 Performance

Naturally it's important to show that the extensive rewrite and introduction of abstractions hasn't impacted the performance of the driver. This can be shown with a full system test that touches as much of the system as possible by submitting a trajectory to follow through the ROS ActionServer interface and measuring the position updates published to the `/joint_states` topic. This kind of test was also conducted in [And15][p. 28] to compare with the even older python driver, where it's also suggested to minimize external factors by conducting the test of the end (tool) joint. All the tests were conducted on an UR5 with firmware version 1.8.14035 and the measured network latency between the computer running the driver and the robot controller was 0.5 to 1.0 millisecond and on average around 0.8 millisecond. The network factor was not entirely isolated as was only measured before and not continuously during testing so external equipment connected to the same network may have caused interference.

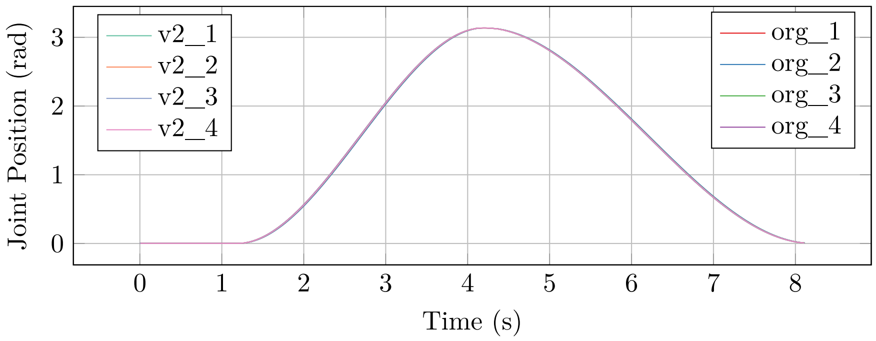


Figure 2.4: Tool joint rotation over time.

The test conducted here therefore consists of first rotating the tool joint to a neutral 0 radian position over 1 second, then 1 radian forward over 3 seconds and then 1 radian backwards over 4 seconds. The initial 1 second zeroing is there minimize any variance in starting position across drivers. The described trajectory is submitted to the driver via a small python script that sets up logging of the joint states and starts logging as soon as the trajectory has been submitted and stops logging when the trajectory finished signal is received. To minimize variance 4 tests per driver were conducted and the linux performance governor of the machine running the driver was set to performance mode ensuring a consistent CPU clock speed.

The combined 8 tests are graphed in Figure 2.4 where it can be seen that both drivers follow the same path very closely. Looking at a 100ms slice of the same graph in Figure 2.5 it can be seen that of all 8 tests the two lines furthest apart are both from the original driver and roughly has a 0.03 radian difference. One difference to note is that the old driver appears to be slightly more consistent in timing between multiple runs in the sense that the timestamp of sample N in one test is within a few milliseconds of sample N of the other tests.

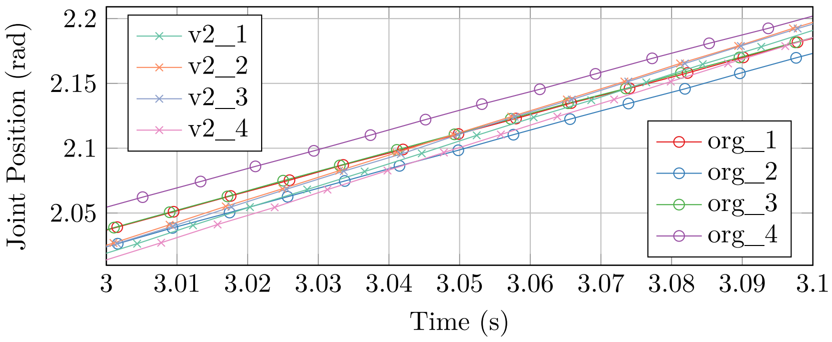


Figure 2.5: Tool joint rotation over time (zoomed).

To ensure this isn't a problem with the new driver introducing more jitter in the publishing of joint states a new test was devised. This test simply logs the timestamp of published joint states and the time since the previous update over a 60 second period. The timestamp of joint state messages are assigned by the driver right before publishing and therefore works as easy measuring points without having to consider the variable delay of the ROS message publishing stack. Again to minimize variance and 4 runs per driver were conducted using the same circumstances described above.

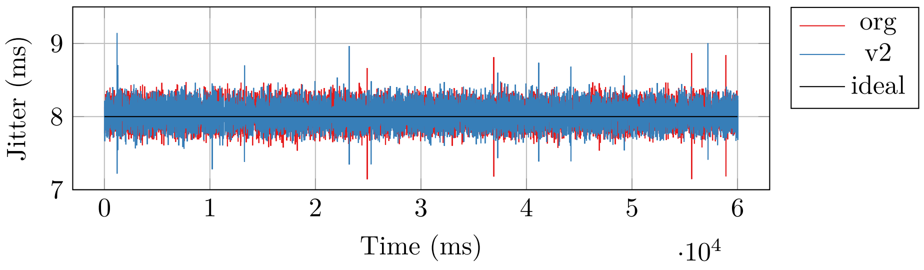


Figure 2.6: Time between joint state updates.

One run from each driver can be seen in Figure 2.6 which shows both drivers having for the most part having ±0.5 millisecond jitter with few spikes exceeding 1 millisecond. Looking at the histogram of the jitter in Figure 2.7 performance similarity of the two drivers is further cemented by both having 99% of joint state updates happen between 7.5 to 8.5 milliseconds.

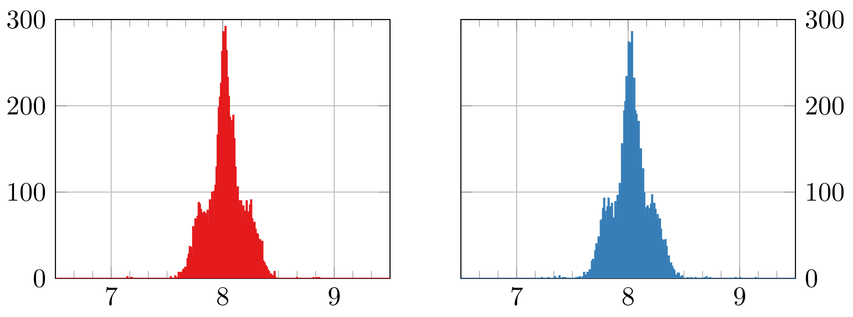


Figure 2.7: Histograms of Figure 2.6.

The 3 remaining tests are plotted in Appendix B which shows the same pattern but with some slightly larger spikes.

2.9 Discussion

Improved support for newer versions of UR firmwares have been achieved because the new driver supports version 3.3.4 which the old driver had some issues with. Furthermore adding support for new versions have been eased thanks to the improved parsing design.

Both joint temperatures and analog tool inputs are parsed by the new driver however currently only the joint temperatures are published.

The service stopper has been implemented and prevents both ActionServer and ROS Control from submitting new goals whenever the robot enters a fault state until this state has been acknowledged.

The driver has been hardened against malicious attacks by adding both explicit and implicit buffer size checks throughout the dangerous parsing code.

Documentation of the code has not improved substantially due to time constraints and a focus on stabilizing design and functionality before documenting code that might change.

Using the driver as a standalone library has not seen any progress either because the code cannot compile without ROS however the pipeline design has been built with this usecase in mind and should make isolating the ROS specific parts fairly simple.

There has been significant progress in adopting code style changes and unit testing, however there are still significant improvements to be made in the unit testing department. No specific communication with ROS-Industrial has taken place yet and should sought to ensure there are no major blocking issues preventing them from adopting code ownership.

2.10 Conclusion

Looking at the design, achieved goals and the performance of the rewritten driver it can be seen that a significant contribution has been made, in the form of improvements and additions, despite taking significantly longer than initially anticipated.

Besides the unfulfilled goals described above the primary improvement that could have been made was better project management both in terms of managing the available time and maintaining continuous contact with ROS-Industrial to have a well defined goal that would result in transferring of ownership.

2.11 Future Work

Further testing should be the primary goal of any future work done because the implemented testing is limited to only parsing. The first step would be to implement unit testing of the publishing sub-system and then integration tests of the pipelines using packet captures from the real robot.

Porting the code to ROS Kinetic is another top priority task. however the majority of changes should be isolated to `RosController` and implementation of `hardware_interface::RobotHW` which has some breaking changes.

Concrete contact should be established with ROS-Industrial to investigate the precise procedure for transferring ownership, specifically regarding licenses, git access and such. Documentation of the code also leaves a lot to be desired and Doxygen should be integrated into the build pipeline. Last but not least a special compile target should be introduced that builds the driver as a library that can be used without any ROS code.

Chapter 3

Bridging Mobotware and ROS

This chapter describes the criteria, design, code and results of the SMR/ROS software bridge.

In order to effectively come up with a solution the exact problem being solved must first be defined. As it stands currently Mobotware only supports a very limited set of robot platforms and as such limit's its impact and reach[BAAR10]. It's therefore desirable to support more platforms but with the limited funding and manpower of the DTU's Robotics department it's simply infeasible to do so on any impactful scale. As such the next best solution is to collaborate with external entities to leverage their code and since ROS has a large amount of pre-written hardware interfaces it seems like a good choice.

The problem therefore becomes: *"How can Mobotware be integrated with the ROS ecosystem to leverage existing ROS hardware interfaces/drivers?"*

3.1 Solution Criteria

Before answering that question a set of criteria for a possible solution must be defined:

- The implementation must incorporate well with the Mobotware hard/soft real-time boundary.
- Basic movement should be supported "out of the box" for wheeled robots.
- Easy to extend with support for new types of hardware interfaces.

Interfacing with advanced sensors like laser and cameras to/from ROS is however outside the scope of this project.

3.2 Solution Space

Before settling for a solution design it's important to explore different options and their pros & cons. The primary design decision to be made when integrating Mobotware and ROS is where to slice open Mobotware and inserting ROS, three of these possible incision points are described below.

A) SMR-CL Interpreter

One solution would be to throw away much of the existing Mobotware code and instead implement a new interpreter that interfaces directly with ROS. Effectively this would be equal to replacing This would provide a more integrated experience for the SMR-CL users/programmers in terms of having access data published from ROS topics. To minimize the amount of new code existing ROS packages such as `move_base` could be used to implement the motion control and would make many of the SMR-CL commands trivial to interpret. However it would require a substantial amount of work to provide compatibility with the robot platforms Mobotware currently supports as many of these do not have corresponding ROS hardware interfaces.

In summary this solution is fairly good if the primary goal is to bring the simplicity of robot programming to ROS but cannot support the current more advanced usages of Mobotware.

B) RHD Plugin

By far the simplest and fastest solution to implement is an RHD plugin that synchronize the ROS odometry topic with the two RHD database variables "transpos" and "rotpos". RHD has multiple controller modes most of which are not well suited to control ROS because the "cmd_vel" topic format is centered around controlling the robot using linear and angular velocities which are hard to translate to from a differential drive controller. Fortunately MRC has a velocity & omega control mode that with simple trigonometry can be converted to the linear and angular velocity format ROS uses. In this mode MRC writes velocities to "cmdtransvel" and "cmdrotvel".

This approach enables SMR-CL control of most wheeled ROS robot that support the "cmd_vel" but does not make it feasible to add support for infrared and line sensors.

In the end this approach was selected because of time constraints and doubts about the feasibility of integrating ROS and Mobotware that could be eliminated as fast as possible with a proof of concept.

3.3 Implementation

Implementing an RHDPlugin is fairly straightforward because it's simply a shared library (.so file) that exports 3 C functions:

- `int initXML(char *filename)` which initializes the plugin and provides a path to the RHD configuration file.
- `int periodic(int RHDtick)` which is called once at every RHD tick.
- `int terminate (void)` which is called when RHD shuts down.

However since the plugin integrates with ROS which requires C++ the functions must be wrapped in `extern "C"` to prevent the functions names from being mangled by the C++ compiler.

Plugins access the variable database through the interface defined in `database.h` from RHD which is pure C and not very convenient to use so templated C++ wrappers was created to represent read and write database variables. The base class can be seen in Listing 3.1 which takes a direction and name to create either a read or write variable and saves the id. Implicit casting to the templated `T` parameter is implemented to make accessing the variable seamless.

```
template <typename T>
class DBVal
{
protected:
  int id_;
  T value_;

  DBVal(const std::string& name, const char dir)
  : id_(createVariable(dir, IPT, name.c_str()))
  { }
  DBVal(const DBVal& o) = default;
  DBVal(DBVal&& o) = default;
  DBVal& operator=(const DBVal& o) = default;
  DBVal& operator=(DBVal&& o) = default;

public:
  static const auto IPT = 1 + ((sizeof(T) - 1) / sizeof(int));
  operator T() const { return value_; }
};
```

Listing 3.1: Database value wrapper.

Using the base wrapper a wrapper for write variables is implemented as seen in Listing 3.2 where the `update` function should be called on each call to the `periodic(int rhdTick)` function to ensure the value is kept in sync. Because the RHD database internally stores values in ints the `update` function work allocating an int buffer that can contain the type `T` which the value from the database is copied into and then from there assigned to the `value_` field.

```cpp
template <typename T>
class DBWriteVal : public DBVal<T>
{
public:
  DBWriteVal(const std::string& name)
  : DBVal<T>(name, 'w')
  { }
  DBWriteVal(const DBWriteVal& o) = default;
  DBWriteVal(DBWriteVal&& o) = default;
  DBWriteVal& operator=(const DBWriteVal& o) = default;
  DBWriteVal& operator=(DBWriteVal&& o) = default;

  bool update()
  {
    if(!isUpdated('w', this->id_))
      return false;

    int buf[DBVal<T>::IPT];
    int res = getWriteArray(this->id_, DBVal<T>::IPT, buf);
    if(res)
      std::memcpy(&(this->value_), buf, sizeof(T));

    return true;
  }
};
```

Listing 3.2: Database write wrapper.

Similarly a read variable wrapper shown in Listing 3.3 is implemented with overridden operators which ensures writes to the variable is always synchronized to the variable database.

```cpp
template <typename T>
class DBReadVal : public DBVal<T>
{
private:
  inline void update(const T& value)
  {
    int buf[DBVal<T>::IPT];
    std::memcpy(buf, &value, sizeof(T));
    //todo check error
    setArray(this->id_, DBVal<T>::IPT, buf);
  }

public:
  DBReadVal(const std::string& name)
  : DBVal<T>(name, 'r')
  { }
  //Delete copy ctr/assign to avoid assigned value_
  //going out of sync between copies
  DBReadVal(const DBReadVal& o) = delete;
  DBReadVal(DBReadVal&& o) = default;
  DBReadVal& operator=(const DBReadVal& o) = delete;
  DBReadVal& operator=(DBReadVal&& o) = default;

  inline DBReadVal<T>& operator=(const T& value)
  {
    update(value);
    this->value_ = value;
    return *this;
  }
  inline DBReadVal<T>& operator+=(const T& value) { return this->operator=(this->value_ + value); }
  inline DBReadVal<T>& operator-=(const T& value) { return this->operator=(this->value_ - value); }
  inline DBReadVal<T>& operator*=(const T& value) { return this->operator=(this->value_ * value); }
};
```

Listing 3.3: Database read wrapper.

Furthermore a small helper class for working with 2D vectors was implemented. Combining the database wrappers and vector helper it only takes a few linens of code to keep the database up to date with odometry from ROS as seen in Listing 3.4 where `transpos_` and `rotpos_` are fields of type `DBReadVal<int32_t>`.

```cpp
void SMROS::onPosUpdate(Vec2D pos, double theta)
{
  auto delta = pos - prev_pos_;
  auto frame = delta * Vec2D(std::cos(theta), std::sin(theta));

  transpos_ += static_cast<int32_t>(frame.sum() * LINEAR_VEL_SCALE);
  rotpos_ = static_cast<int32_t>(theta * ANGULAR_VEL_SCALE);

  prev_pos_ = pos;
}
```

Listing 3.4: Updating "`transpos`" and "`rotpos`".

This `onPosUpdate` function is called every time a new odometry message is received from ROS.

The final piece of the puzzle is publishing velocity commands to "`cmd_vel`" whenever the "`cmdrotvel`" or "`cmdtransvel`" database variables update as seen in Listing 3.5. The primary thing to notice here is that the linear velocities have a cutoff of $0.005m/s$ because otherwise the simulation described in section 3.4 would slowly drift out of place over time, it's however uncertain if this would happen or not with a real robot due to friction. Another important implementation detail here is that `cmd_vel_publisher_` is not a regular ROS publisher but rather a `realtime_tools::RealtimePublisher` which runs the actual publishing in a separate thread. This is important because the code runs in the hard real-time section of Mobotware and should under no circumstances block the thread.

```cpp
bool SMROS::tick(int rhdTick)
{
  static const double epsilon = 0.005;
  bool updated = cmdtransvel_.update();
  updated |= cmdrotvel_.update();

  //only publish when either rot or vel changed
  if(updated && cmd_vel_publisher_.trylock())
  {
    double rot = (cmdrotvel_ / ANGULAR_VEL_SCALE);
    double vel = cmdtransvel_ / LINEAR_VEL_SCALE;
    auto x = std::cos(rot) * vel;
    auto y = std::sin(rot) * vel;
    //ignore very small values to remove instability that would
    //result in drifting
    x = x > epsilon ? x : (x < -epsilon ? x : 0.);
    y = y > epsilon ? y : (y < -epsilon ? y : 0.);

    //MRC rotvel to ROS geometry_msgs/Twist angular.z
    //MRC transvel to ROS geometry_msgs/Twist linear x & y
    cmd_vel_publisher_.msg_.linear.x = x;
    cmd_vel_publisher_.msg_.linear.y = y;
    cmd_vel_publisher_.msg_.linear.z = 0.;
    cmd_vel_publisher_.msg_.angular.z = rot;
    cmd_vel_publisher_.unlockAndPublish();
  }

  return ros::ok();
}
```

Listing 3.5: Publishing to "`cmd_vel`".

In total this proof of concept implementation is roughly 400 lines of code.

3.4 Results

Due to time constraints it was not possible to test the implementation on a robot however ROS provides a convenient 2D simulator called `turtlesim` which testing could be done with. The simulator spawns a turtle in the center of an 11 × 11 meter room and moves the turtle according to messages received on the `cmd_vel` topic and publishes position information to the `turtle1/pose` topic. To demonstrate the implementation correctness two SMR-CL programs were written. The first one shown in page 29 works as expected and draws a square spiral with a diagonal crossing the entire width.

```
fwd 1
turn 90
fwd 1
turn 90
fwd 2
turn 90
fwd 2
turn 90
fwd 3
turn 90
fwd 3
turn 90
fwd 4
turn 90
fwd 4
turn 135
l=sqrt(5*5+5*5)
fwd l
```

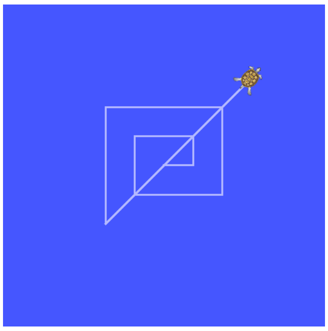


Figure 3.1: SMR-CL code and resulting turtlesim path.

However when testing the `turnr` command which is described as "turn b degrees with turning radius r in meters", something goes wrong and the turtle keeps following the same circle and never stopping as seen in Figure 3.2.

```
turnr 1 90
```

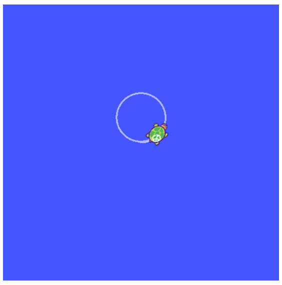


Figure 3.2: SMR-CL code and resulting turtlesim path.

The problem however does not seem to originate from the plugin because executing the same command in the Mobotware provided Stage simulation of the SMR platform results in the same never ending spinning. Unfortunately due to time constraints further investigation of the problem was not possible.

3.5 Discussion

This chapter has shown that both Mobotware and ROS provide powerful abstractions that enables both systems to be integrated in a relatively short time period. However the internal structure of Mobotware is poorly documented with none to few comments throughout the very terse C codebase which meant the only way to discover which database variables to read and write was to look through the source of existing plugins. Unfortunately only a single plugin, namely `rflex`, implemented the velocity omega control interface and to complicate matters further the data written to the `"transpos"` and `"rotpos"` database variables came directly a hardware sensor of a poorly documented robot. All of the above combined with the authors lack of experience with robot odometry meant that it took unnecessarily long to reverse engineer the simple trigonometric function to calculate `"transpos"` shown in Listing 3.4.

The first solution criteria to incorporate well with the hard/soft real-time boundaries of Mobotware has been fulfilled as shown in Listing 3.5 by taking special care to only interface with ROS in a way that does not interfere with the hard real-time thread running the `tick` function. The second solution criteria to support wheeled robots has also been fulfilled as the `cmd_vel` interface is the defacto standard in ROS to control wheeled robots. The third and last solution criteria to make the implementation easy to extend has also been fulfilled by providing wrappers for database variables that makes them significantly more natural to interact with.

3.6 Conclusion

This chapter has proposed 2 different solutions and implemented one as a proof of concept to answer the question: *"How can Mobotware be integrated with the ROS ecosystem to leverage existing ROS hardware interfaces/drivers?"*.

While the implemented solution by no means is perfect it provides a basic and clean implementation that in the future can be used to further integrate Mobotware and ROS.

3.7 Future Work

In the future it could be interesting to extend the implemented RHD plugin to support the remaining control modes of MRC as there most likely exists robots supported by ROS that are controlled in similar fashions. Possible future work also includes implementing one or more Automation Robot Server plugins that interface with the more advanced ROS sensors such as laser scanners and cameras.

Chapter 4

Conclusion

Finishing this project has been a long and grueling 6 months with many bumps along the road however in the end it succeeded and many lessons has been learned.

Despite initial skepticism ROS has turned out to be a pleasant platform to work with due to the large ecosystem of existing packages.

Over all the the primary objects have been fulfilled however the meta aspects of the project such as time management, planning and communication could have been done better.

Appendix A

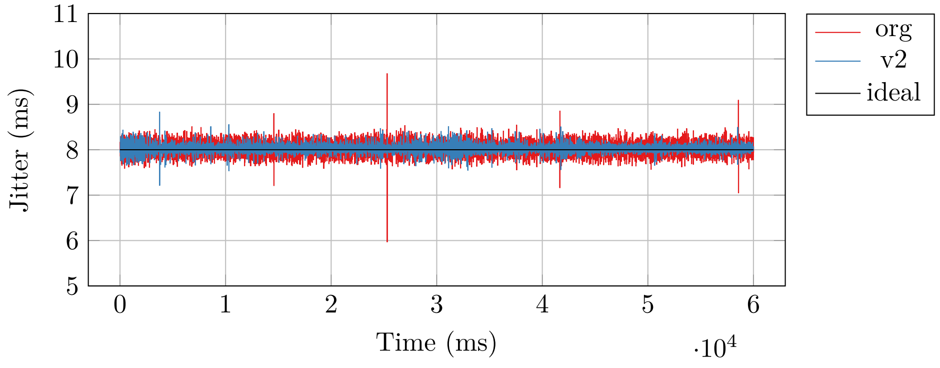
UR Protocol

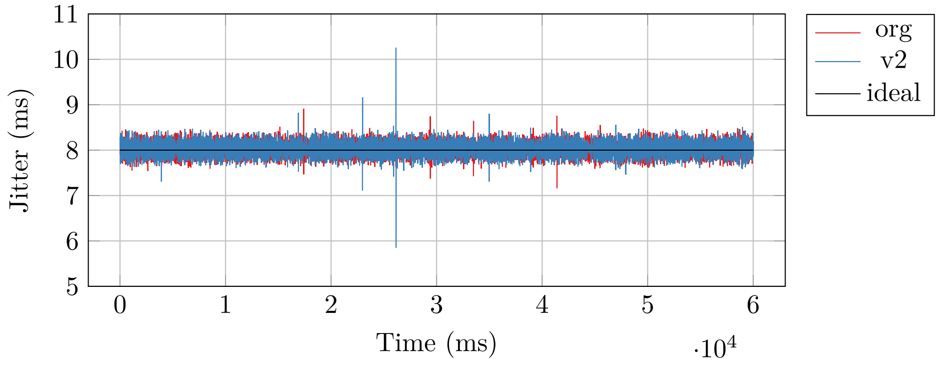
Meaning	Type	Num. values	Num. bytes	Notes
Message Size	uint32_t	1	4	Total message length in bytes
Time	double	1	8	Time elapsed since the controller was started
q target	double	6	48	Target joint positions
qd target	double	6	48	Target joint velocities
qdd target	double	6	48	Target joint accelerations
I target	double	6	48	Target joint currents
M target	double	6	48	Target joint moments (torques)
q actual	double	6	48	Actual joint positions
qd actual	double	6	48	Actual joint velocities
I actual	double	6	48	Actual joint currents
Tool Accelerometer values	double	3	24	Tool x,y and z accelerometer values
Unused	double	15	120	Unused
TCP force	double	6	48	Generalised forces in the TCP
Tool vector	double	6	48	Cartesian coordinates of the tool
TCP speed	double	6	48	Speed of the tool given in cartesian coordinates
Digital input bits	uint64_t	1	8	Current state of the digital inputs.
Motor temperatures	double	6	48	Temperature of each joint in degrees celcius
Controller Timer	double	1	8	Controller realtime thread execution time
Test value	double	1	8	A value used by Universal Robots software only
Robot Mode	double	1	8	Robot control mode
Joint Modes	double	6	48	Joint control modes

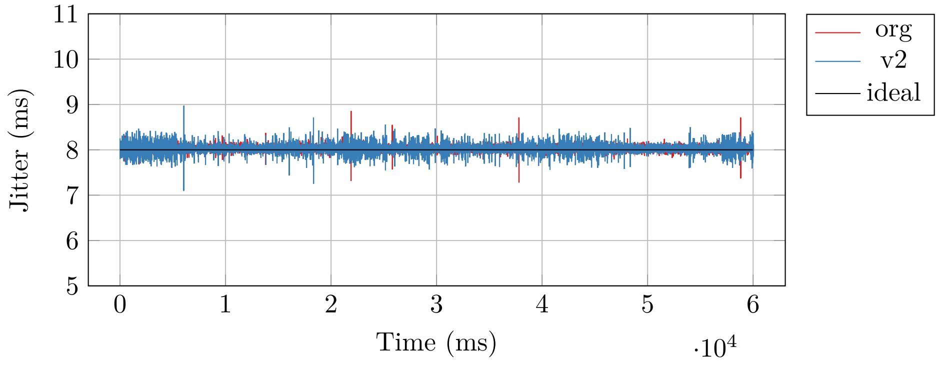
Table A.1: Real-time message format for v1.8

Appendix B

Driver Plots


11
10
9
8
7
6
5
Jitter (ms)
0
1
2
3
4
5
6
Time (ms)
·10^4
org
v2
ideal


11
10
9
8
7
6
5
Jitter (ms)
0
1
2
3
4
5
6
Time (ms)
·10^4
org
v2
ideal


11
10
9
8
7
6
5
Jitter (ms)
0
1
2
3
4
5
6
Time (ms)
·10^4
org
v2
ideal

Bibliography

[AlD] AlDanial. cloc. `https://github.com/AlDanial/cloc`. Accessed: 2017-07-9.

[And15] Thomas Timm Andersen. *Optimizing the Universal Robots ROS driver.* Technical University of Denmark, Department of Electrical Engineering, 2015.

[BAAR10] Anders B. Beck, Nils Axel Andersen, Jens Christian Andersen, and Ole Ravn. Mobotware a plug-in based framework for mobile robots. *IFAC Proceedings Volumes*, 43(16):127 – 132, 2010. 7th IFAC Symposium on Intelligent Autonomous Vehicles.

[Col] Dave Coleman. Roscpp code format. `https://github.com/davetcoleman/roscpp_code_format`. Accessed: 2017-07-9.

[Ind] ROS Industrial. Ros industrial development process. `http://wiki.ros.org/Industrial/DevProcess`. Accessed: 2017-07-9.

[ROS] ROS. Ros - docker hub. `https://hub.docker.com/_/ros/`. Accessed: 2017-07-9.

[Tim] Thomas Timm. ur_modern_driver. `https://github.com/ThomasTimm/ur_modern_driver`. Accessed: 2017-07-9.