ECE 544 Final Project
ARCL-ROS Driver

Samuel Bellaire

April 2022

Table of Contents

1 Introduction to ARCL

Advanced Robotics Command Language (ARCL) is a simple text-based protocol that can be used to communicate with Omron's LD-series mobile robots. The user must connect to the ARCL server hosted on the robot using a Telnet client like PuTTY, or with a raw TCP connection. Once the user has connected to the ARCL server and has logged in, they can send commands to the robot through the client.

Using ARCL, it is possible to issue simple commands to the robot, such as navigating to goals or waypoints, moving forward or turning by a specified amount, and reading the robot's sensor data. It should be noted that ARCL is not a replacement to Omron MobilePlanner, which is still useful for more complex tasks like mapping the environment and configuring parameters on the robot. Refer to the *Mobile Robot Software Suite User's Guide* for more information on MobilePlanner.

1.1 Connecting to the ARCL Server

To connect to the robot's ARCL server, it must first be configured via MobilePlanner. The steps below summarize this process; see Fig. 1 for a visual reference.

1. Connect to the robot using MobilePlanner. The IP address of the robot is displayed on its touchscreen.

2. Open the robot's configuration and select the *Robot Interface* tab. From this tab, select *ARCL server setup.*

3. Ensure that the parameter *OpenTextServer* is set to True, and that a value is set for the *Password* parameter. The ARCL server will not open unless a password is set.

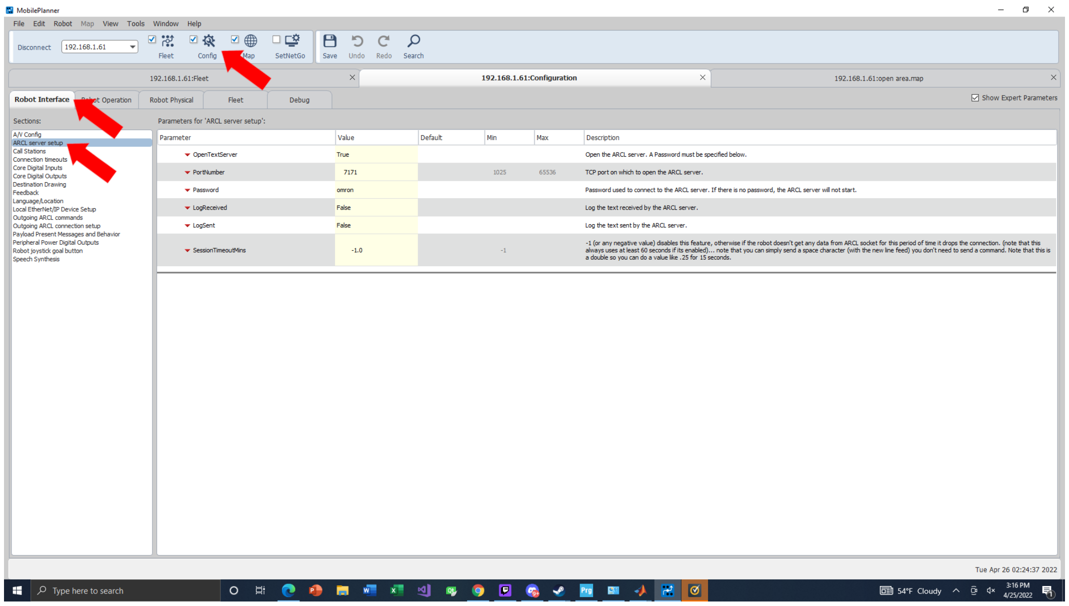
4. Take note of the *PortNumber* parameter, as this is the port the user must use to connect to the ARCL server. By default, this value is 7171.

5. Save the configuration to the robot if any changes were made.

6. Connect to the ARCL server using a Telnet client such as PuTTY (see Fig. 2 for examples). The command `telnet <IP> <port>` can also be used in a terminal on a system with Telnet enabled. The IP address, port number, and password noted in the previous steps are used to connect and login to the ARCL server.

1.2 Additional ARCL Resources

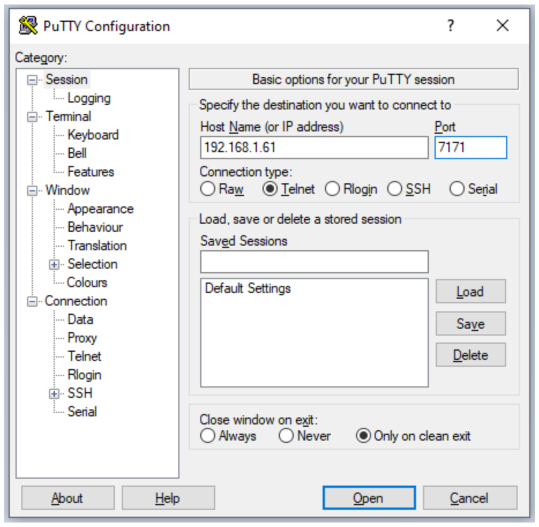
Additional information about the ARCL command language can be found in Omron's *Advanced Robotics Command Language Reference Guide*, though it should be noted that some commands are undocumented. This guide only lists the available ARCL commands; MobilePlanner must be used to find the list of tasks that the mobile robot is able to carry out using the `dotask` command.



Parameters for 'ARCL server setup':

Parameter	Value	Default	Min	Max	Description
OpenTextServer	True				Open the ARCL server. A Password must be specified below.
PortNumber	7171		1025	65536	TCP port on which to open the ARCL server.
Password	omron				Password used to connect to the ARCL server. If there is no password, the ARCL server will not start.
LogReceived	False				Log the text received by the ARCL server.
LogSent	False				Log the text sent by the ARCL server.
SessionTimeoutMins	-1.0		-1		-1 (or any negative value) disables this feature, otherwise if the robot doesn't get any data from ARCL socket for this period of time it drops the connection. (note that this always uses at least 60 seconds if its enabled)... note that you can simply send a space character (with the new line feed) you don't need to send a command. Note that this is a double so you can do a value like .25 for 15 seconds.

Figure 1: MobilePlanner ARCL Config



PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): 192.168.1.61

Port: 7171

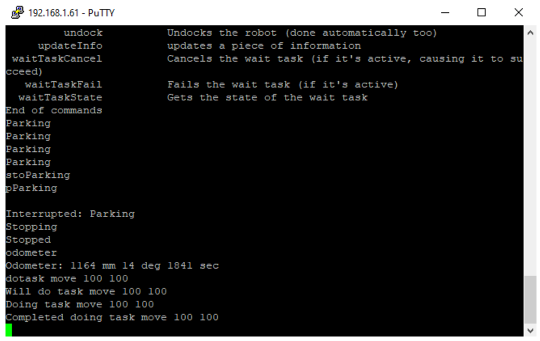
Connection type: ( ) Raw [✓ Telnet] ( ) Rlogin ( ) SSH ( ) Serial

Load, save or delete a stored session

Saved Sessions: Default Settings

Close window on exit: ( ) Always ( ) Never [✓ Only on clean exit]

(a) PuTTY Setup



192.168.1.61 - PuTTY

```
         undock           Undocks the robot (done automatically too)
     updateInfo           updates a piece of information
 waitTaskCancel           Cancels the wait task (if it's active, causing it to su
cceed)
   waitTaskFail           Fails the wait task (if it's active)
  waitTaskState           Gets the state of the wait task
End of commands
Parking
Parking
Parking
Parking
stoParking
pParking

Interrupted: Parking
Stopping
Stopped
odometer
Odometer: 1164 mm 14 deg 1841 sec
dotask move 100 100
Will do task move 100 100
Doing task move 100 100
Completed doing task move 100 100
```

(b) ARCL Commands Sent via Terminal

Figure 2: Telnet with PuTTY

2 Package Overview

Out-of-the-box, Omron's LD-series mobile robots are not compatible with ROS, and can only be interfaced with via Omron's software (e.g. MobilePlanner) and the ARCL command language. The aim of this ROS package is to provide an interface for ROS to control an LD-series mobile robot.

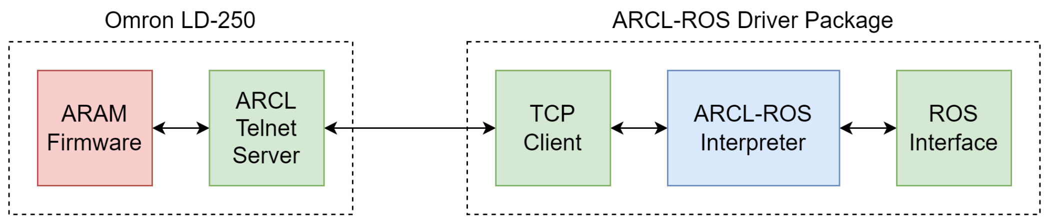


Figure 3: ARCL-ROS Driver Overview

Fig. 3 shows a high-level overview of the ROS package, whose primary function is to act as an interpreter for the interfaces at both ends. Topics that are published in ROS can be translated to ARCL commands via the package to control the robot, and status information that is sent back by the ARCL server is reformatted into ROS topics and published.

The package is split into a driver package (`umd_arcl_driver`) which is the primary package that interfaces the mobile robot with ROS, and a teleop package (`umd_arcl_teleop`) which is setup to use the joy package to drive the mobile robot using a controller. The teleoperation package acts as a demo for what the driver package is capable of.

Sec. 2.1 - 2.5 details all of the package dependencies, ROS topics that the package either subscribes or publishes to, all of the ROS parameters that the node accepts, and all of the custom message types added by the package.

2.1 Package Dependencies

All of the dependencies for the `umd_arcl_driver` package are listed below. In addition to the list below, the `joy` package is also required to use the `umd_arcl_teleop` package.

```
roscpp
std_msgs
geometry_msgs
sensor_msgs
message_generation
```

2.2 Published Topics

`arcl/battery (std_msgs/Float32)`

The state-of-charge of the mobile robot's battery, in percent.

`arcl/pose (umd_arcl_driver/Pose2DStamped)`

The current pose of the mobile robot. The (x, y) position is measured in mm, and the angle θ in degrees. A timestamp is also contained within the message's header.

`arcl/odom (umd_arcl_driver/Odometer)`

The mobile robot's current odometer reading. The total linear distance travelled in mm, total angle rotated in degrees, and total time elapsed in seconds since the robot powered on (or since the odometer was reset) are included in the topic. A timestamp is also contained within the message's header.

`arcl/lidar (sensor_msgs/PointCloud)`

A point cloud containing the most recent scan of the mobile robot's primary LiDAR. Each point's (x, y, z) position is measured in mm, with the z coordinate set to 0 mm. A timestamp is also contained within the message's header.

`arcl/lidarLow (sensor_msgs/PointCloud)`

Same as `arcl/lidar`, but contains the toe laser LiDAR data instead of the primary LiDAR's data.

2.3 Subscribed Topics

`arcl/move (umd_arcl_driver/Command2)`

Commands the mobile robot to move forwards in a straight line. The distance to move is measured in mm, and the speed is measured in mm/sec.

`arcl/rotate (umd_arcl_driver/Command2)`

Commands the mobile robot to rotate in place by a specified amount. The angle to rotate is measured in degrees, and the angular velocity in deg/sec.

`arcl/rotateTo (umd_arcl_driver/Command2)`

Same as `arcl/rotate`, but commands the mobile robot to rotate to a specific heading, rather than rotate relative to its current heading.

`arcl/stop (std_msgs/Empty)`

Commands the robot to stop, interrupting the current task.

`arcl/say (std_msgs/String)`

Commands the robot to use its text-to-speech synthesizer on the string contained by the message.

`arcl/cmd_vel (geometry_msgs/Twist)`

Commands the robot's linear or angular velocity. Note that control of both simultaneously is not possible - the robot will prioritize turning in place over linear motion. The linear x value is measured in mm/sec, and the angular z value in deg/sec.

`arcl/odomReset (std_msgs/Empty)`

Commands the robot to reset its odometer. Its total distance travelled, total angle rotated, and total uptime will all be reset to 0.

2.4 Parameters

`arcl/ip (std::string, default = "192.168.1.61")`

The IP address of the mobile robot to connect to.

`arcl/port (int, default = 7171)`

The port number of the ARCL server hosted on the mobile robot.

`arcl/pw (std::string, default = "omron")`

The password used to connect to the ARCL server hosted on the mobile robot.

`arcl/statusRate (int, default = 10)`

The rate at which the topics `arcl/battery`, `arcl/odom`, and `arcl/pose` are published. Note that the node runs at 200 Hz, so the actual rate may not be exact.

`arcl/lidarRate (int, default = 1)`

The rate at which the topics `arcl/lidar` and `arcl/lidarLow` are published. Note that the node runs at 200 Hz, so the actual rate may not be exact.

`arcl/primaryLidarName (std::string, default = "Laser_1")`

The name of the mobile robot's primary LiDAR, as defined in MobilePlanner.

`arcl/lowLidarName (std::string, default = "Laser_2")`

The name of the mobile robot's low LiDAR, as defined in MobilePlanner.

2.5 Messages

`umd_arcl_driver/Pose2DStamped`

```
std_msgs/Header header
geometry_msgs/Pose2D pose
```

`umd_arcl_driver/Odometer`

```
std_msgs/Header header
int32 distance
int32 angle
int32 time
```

`umd_arcl_driver/Command2`

```
int32 target
int32 speed
```

3 Package Low-Level Design

The low-level design of the ROS package can be separated into an initialization phase where the network is set up and the node connects to the ARCL server, and a ”loop” phase, where the program runs indefinitely inside the main program loop. Both of these are detailed further below. For the complete source code, visit the Github repository `https://github.com/SBellaire7/umd_arcl`

3.1 Logging into the ARCL Server

During the initialization process of the node, a connection to the ARCL server must be established and the node must login; this process is shown in block diagram form in Fig. 4. Once the node initializes the connection with the ARCL server, it reads from the server until it finds the ”password” keyword. This is handled by a low-level function library that communicates directly with the ARCL server via a TCP socket; several other modules in Fig. 4 and Fig. 5 also interface with this library.

Once the keyword is found (indicating that the server has prompted the user to enter a password), the node sends the password to the ARCL server, and then checks for a response using a blocking read. If the read times out, then the connection was likely rejected by the server and the program terminates. Otherwise, the node is considered to have successfully logged in, and data is read from the socket to clear the read buffer. This is done because the ARCL server will send a list of valid commands to the client upon login; this information does not have to be processed by the node and can be discarded.

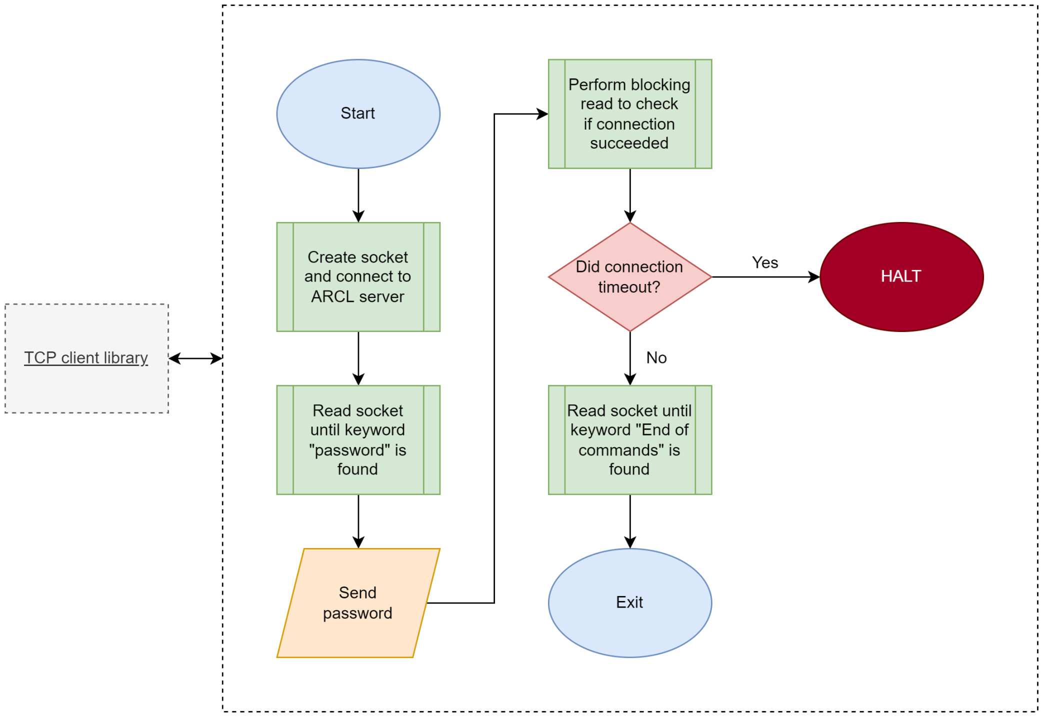

Start
Create socket and connect to ARCL server
Read socket until keyword "password" is found
Send password
Perform blocking read to check if connection succeeded
Did connection timeout?
Yes
HALT
No
Read socket until keyword "End of commands" is found
Exit
TCP client library

Figure 4: ARCL Login Process

3.2 Program Loop

After connecting to the ARCL server and logging in, the node will hand over control to the program loop, which runs at 200Hz. Fig. 5 shows a block diagram of this loop, which handles all of the interfacing between ARCL and ROS.

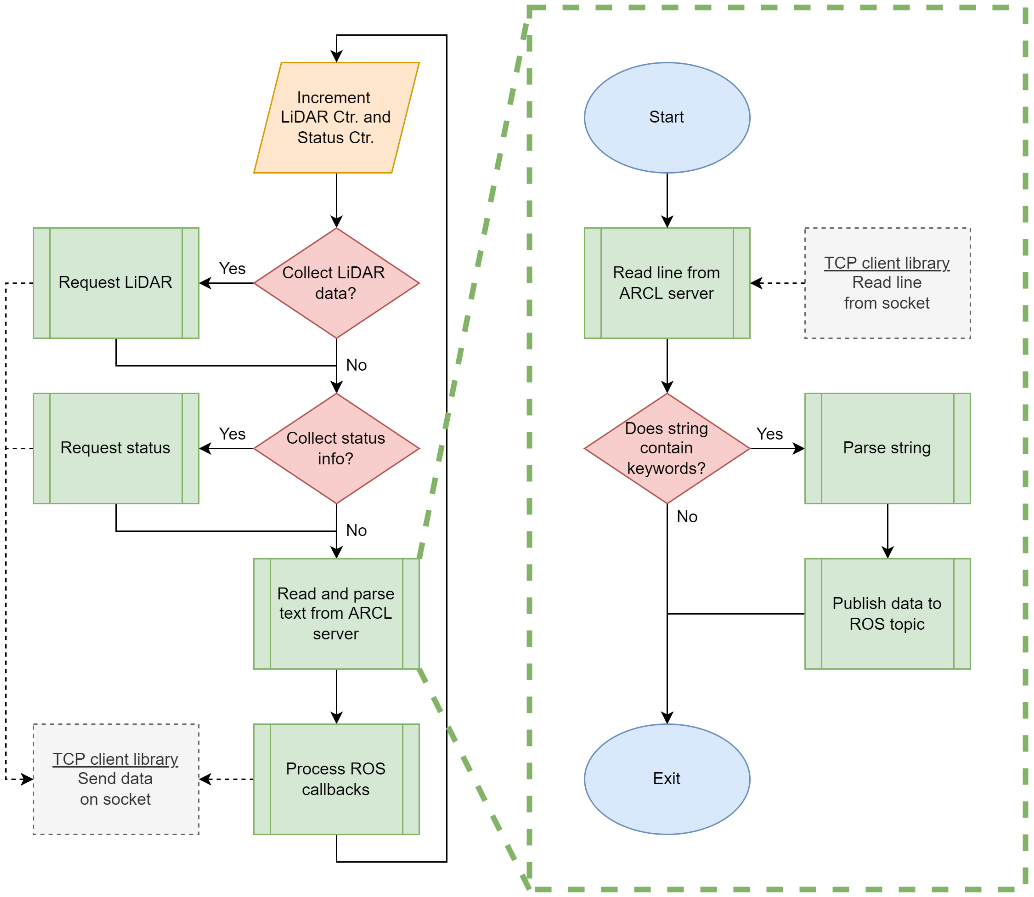


Figure 5: Main Program Loop Block Diagram

3.2.1 Data Requests

In every iteration of the program loop, two counters are incremented to determine if a request should be sent to the robot for LiDAR data or status data. If the counter is equal to the desired rate, then the counter is reset and the data is requested (see Lst. 1 for an example).

3.2.2 Reading from the ARCL Server

After requesting data from the mobile robot, the package checks to see if data was received from the ARCL server. The internal workings of this block can be seen on the right-hand-side of Fig. 5. The function first reads one line from the ARCL server. This is accomplished in the TCP library by peeking the socket to check if a newline (`\n`) character is present in the receive buffer. If a newline character is present, then the data is read from the receive buffer up to and including the newline character.

```
statusCtr++;
lidarCtr++;

//request lidar data
if(lidarCtr == 200 / Driver.getLidarRate())
{
    lidarCtr = 0;
    Driver.requestLidar();
    Driver.requestLidarLow();
}
...
```

Listing 1: C++ Code for Data Request

After a line is read from the server, a check is performed to determine if the string contains any keywords that indicates it is a response to a request for data. If a keyword is present, then the string is parsed to extract the data, and is subsequently published to its respective ROS topic.

3.2.3 ROS Callbacks

After executing all of the main processes in the loop, all of the ROS callbacks are processed, which sends commands to the mobile robot if any of the topics the node subscribes to received a message. Lst. 2 shows an example of such a callback, simplified to pseudocode for readability in this document. The callback commands the robot to move in a straight line for `msg.target` mm at `msg.speed` mm/sec.

```
void moveCB(msg)
{
    string cmd = "dotask move " + msg.target + " "
                 + msg.speed + "\n";
    socketSend(socket, cmd);
}
```

Listing 2: ROS "Move" Callback Pseudocode

4 Future Work

Although this ROS package implements many important features, such as subscribing to commands to allow the robot to move and publishing the mobile robot's status, odometry, and LiDAR data, there are still many improvements that can be made in this package that are discussed in Sec. 4.1 and 4.2

4.1 Package Limitations

One of the primary limitations of this ROS package stems from the limitations of the ARCL command language. Perhaps the most obvious limitation is the lack of any real way to command the robot's linear and angular velocity without using a "hack" approach. Even then, control of both linear and angular velocity simultaneously is currently not possible without gaining lower-level access to the mobile robot's firmware.

4.2 Future Improvements

The (non-exhaustive) list below describes some improvements that can be made to the package to expand its functionality or improve its robustness.

- Splitting the package into multiple sub-packages or nodes could make it more modular.
- Situations where the connection to the ARCL server is suddenly closed, or the connection drops, are currently unhandled. The node does not terminate or attempt to reconnect to the ARCL server.
- Only a handful of ARCL commands and tasks were included in the package that were deemed as essential. In the future, more could be added.