# Project 1: Teleop Keyboard

Due 2/20/2026 at 11:59 PM

## Introduction

The Project 1 objective is to implement a command line interface (CLI) for keyboard teleoperation of the MBot Omni. Once complete, a user should be able to drive the MBot using the WASD keys to move forward, left, right, and backwards, the QE keys to rotate counter-clockwise and clockwise, and the spacebar to stop.

This project requires you to implement two programs: *mbot_driver* and *teleop_keyboard*. The *mbot_driver* program will read drive commands from stdin and forward them to the MBot Control Board via USB. This program is designed to interface with the other two programs using unnamed pipes from the command line. The *teleop_keyboard* program will open a named pipe (FIFO) for parsing keyboard input. It will read individual characters from the FIFO and convert them into drive commands. To write characters to the FIFO, use executables such as *echo* or *tee*.

```
# Downloading the Project 1 tarball
wget -O Project1.tar.gz
"https://www.dropbox.com/scl/fi/2zam3qzjumei0k0unozwm/project1_template.gz?rlkey=3sd06
kchxii5ezah5jxx1b5lm&st=j6uht9lj&dl=0"
# Extracting the files
tar -xzf Project1.tar.gz
# Delete the tarball
rm Project1.tar.gz
```

Or simply go here to download to your systems
https://www.dropbox.com/scl/fi/2zam3qzjumei0k0unozwm/project1_template.gz?rlkey=3sd06kchxii5ezah5jxx1b5lm&st=j6uht9lj&dl=0

## Part 1: File, Pipe, Fifo, and Signal

The projects in this course are designed to build on each other. In projects 2 and 3, you will reuse the classes that you implement here. To maintain a clean and readable codebase, we are enforcing the use of interfaces, which are abstract base classes without member variables that define the behavior of derived classes. Interfaces are useful because it provides a user with a predictable behavior for a class depending on the interfaces that it extends. They are also useful

for unit testing and mocking, which is how we will autograde your projects in this course. Read more on interfaces in this blog post.

In this project, you will be introduced to the *IO* and *Notification* interface, available in the files *include/rix/ipc/interfaces/io.hpp* and *include/rix/ipc/interfaces/notification.hpp*. The *IO* interface is used to synchronously read and write data to some buffer in non-blocking or blocking mode. For more information regarding non-blocking IO, see the appendix. The *Notification* interface is used to synchronously detect when some state is raised high. The *IO* interface will be extended for the *File* and *Fifo* classes to send data to and from the respective kernel-level buffers. The *Notification* interface will be extended for the *Signal* class to detect when a specified signal has been received by the process.

## File

The File class should implement the IO interface using a file descriptor and system calls. You must implement the member functions of the File class in src/rix/ipc/file.cpp. The expected behavior of each function is defined in the header file include/rix/ipc/file.hpp. Public tests are available in tests/file.cpp. Use these tests to inform your implementation of the member functions!

| **Feature** | **Points** |
|---|---|
| Pass unit tests | 3 |

Necessary system calls: unlink, open, dup, fcntl, poll, close, read, and write.

## Pipe

A pipe is a unidirectional data channel. The pipe system call is used to obtain 2 file descriptors, one for reading and one for writing. It is often used with the fork system call to communicate data from one process to another (although we do not cover that in this project). It is also the mechanism behind command line pipes: |.

The Pipe class implements the IO interface, but it derives from File instead of IO directly. A pipe is a special file that does not have an entry on the file system. All of the data written to a pipe is buffered at the kernel level and never reaches the file system.

You must implement the member functions of the Pipe class in src/rix/ipc/pipe.cpp. The expected behavior of each function is defined in the header file include/rix/ipc/pipe.hpp. Public tests are available in tests/pipe.cpp. Use these tests to inform your implementation of the member functions!

| Feature | Points |
| --- | --- |
| Pass unit tests | 3 |

Necessary system calls: pipe, close, and dup

## Fifo

A FIFO special file (a named pipe) is similar to a pipe, except that it is accessed as part of the filesystem. It can be opened by multiple processes for reading or writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the filesystem.

The Fifo class implements the IO interface, but it derives from File instead of IO directly. On POSIX compliant operating systems, a named pipe is represented using a file descriptor, and shares much functionality with a regular file. Like an anonymous pipe, data written to this file never reaches the file system and is buffered at the kernel, despite having an entry in the file system. This entry enables processes to communicate with each so long as they both know of the path to the fifo special file.

You must implement the member functions of the Fifo class in src/rix/ipc/fifo.cpp. The expected behavior of each function is defined in the header file include/rix/ipc/fifo.hpp. Public tests are available in tests/fifo.cpp. Use these tests to inform your implementation of the member functions!

| Feature | Points |
| --- | --- |
| Pass unit tests | 3 |

Necessary system calls: mkfifo, open, and dup

## Signal

The Signal class derives from the Notification interface. This class is used to detect when a signal has been received by the process. Below is an example of the expected functionality of the Signal class.

```
int main() {
      rix::ipc::Signal signal(SIGINT);
      sig.wait(rix::util::Duration::max());
      rix::util::Log::info << “Signal received!” << std::endl;
}
```

The code snippet above creates a Signal object with the signal number SIGINT. The program then sleeps until the signal is received by the program. Once the signal is received, it will log a message. This is useful when you need to be able to terminate a program from the command line, but also need to "clean up" before the program terminates.

The signal class defines an array of SignalNotifier structs as a private member variable. This array has 32 elements, one for each valid signal number. Note that the signal numbers (such as SIGINT, SIGPIPE, etc.) range from **1 to 32**, but the array indices range from **0 to 31**. The SignalNotifier struct contains 2 Pipe objects (one for reading, one for writing) and a flag that should be used to determine if the SignalNotifier has been initialized. In the constructor for the Signal class, you must initialize the Pipes in the array that correspond to the valid signal number. Then, you will register the Signal::handler function for the specified signal number. In the Signal::handler function, you will write data to the write-end of the Pipe corresponding to the signal number that was received.

We use Pipes to synchronize our main thread with our signal handler because writing to a file descriptor is an async signal safe operation (read more on signal safety here). This allows us to wait for a signal to arrive by waiting for the read-end of the Pipe corresponding to the signal number to become readable. If we want to check if a signal has arrived without waiting, we can invoke the Signal::wait function with a 0 duration.

You must implement the member functions of the Signal class in src/rix/ipc/signal.cpp. The expected behavior of each function is defined in the header file include/rix/ipc/signal.hpp. Public tests are available in tests/signal.cpp. Use these tests to inform your implementation of the member functions!

| **Feature** | **Points** |
|---|---|
| Pass unit tests | 3 |

Necessary system calls: signal, raise, and kill

# Part 2: Serialization

We now have a means of marshalling data from one process to another (FIFOs), however, we are only able to send strings and binary data (arrays of bytes). How can we marshall useful data objects that don't have a clear binary representation? We will define a serialization protocol for a set of data objects: numbers (bool, char, int8, uint8, int16, uint16, int32, uint32, int64, uint64, float, and double), strings (std::string), arrays of numbers (std::array<T> where T is an arithmetic type), arrays of strings (std::array<std::string>), vectors of numbers (std::vector<T> where T is an arithmetic type), and vectors of strings (std::vector<std::string>).

These serialization methods can be combined to implement serialize/deserialize functions for custom data objects, such as rix::msg::standard::Header in rix/msg/standard/Header.hpp.

You must implement the serialize and deserialize functions in rix/msg/serialization.hpp. **Important**: more information on the expected RIX serialization format can be found here.

| Feature | Points |
|---|---|
| Pass unit tests | 6 |

# Part 3: Executables

## mbot_driver

The *mbot_driver* is a program that is responsible for commanding the velocity of the MBot from messages read from stdin. This program should run until SIGINT is received. Before terminating, the program should send a stop command to the MBot, which is a drive command with 0 speed in each field. Use the *MBot* class defined in *include/mbot/mbot.hpp* and implemented in *src/mbot/mbot.cpp* to send 2D twist drive commands to the MBot.

The program should use the *File* class implemented in Part 1 to read data from stdin. The file descriptor for stdin is STDIN_FILENO, which is equal to 0. The data will be serialized *rix::msg::geometry::Twist2DStamped* messages prefixed with the total size of the message. The stdin buffer will look like the following diagram, where $n_i$ is a 32-bit (4-byte) unsigned integer that represents the total size in bytes of the proceeding serialized Twist2DStamped message.

| $n_1$ (4 bytes) | serialized *Twist2DStamped* ($n_1$ bytes) | $n_2$ (4 bytes) | serialized *Twist2DStamped* ($n_2$ bytes) | … |
|---|---|---|---|---|

In order to successfully read and deserialize this message, you must first read 4 bytes from stdin and deserialize them as an unsigned 32-bit integer (see *rix::msg::standard::UInt32*). Then, perform another read from stdin with the number of bytes specified and store this data into a byte array. Finally, deserialize the *Twist2DStamped* message and drive the MBot accordingly.

| Feature | Points |
|---|---|
| If a notification is received (SIGINT in the main program), the program sends a stop command to MBot and returns 0. | 2 |
| The program deserializes bytes read from stdin into a *rix::msg::geometry::Twist2DStamped* message and uses the *vx*, *vy*, and | 2 |

| | |
|---|---|
| *wz* parameters to drive the MBot. | |
| When the end-of-file (EOF) is reached on the input, the program sends a stop command to MBot and returns 0. | 2 |

The usage of the *mbot_driver* executable is as follows.

```
./mbot_driver
```

## teleop_keyboard

The *teleop_keyboard* program will write 2D twist drive commands to stdout that correspond to keys pressed on the keyboard. The program will use a FIFO to read keyboard input. There are two input arguments to *teleop_keyboard*: *angular_speed* and *linear_speed*. These input arguments determine the speed in the drive commands sent to the MBot (*V* and *W* in the table below).

Table 1: Mapping for characters to 2D twists for keyboard teleoperation. *V* is a scalar value for linear speed and *W* is a scalar value for angular speed.

| **Key** | **2D Twist (<*vx, vy, wz*>)** |
|---|---|
| W | <+*V*, 0, 0> |
| A | <0, +*V*, 0> |
| S | <-*V*, 0, 0> |
| D | <0, -*V*, 0> |
| Q | <0, 0, *W*> |
| E | <0, 0, -*W*> |
| Space | <0, 0, 0> |
| Anything else | No command |

The program should use the *File* class implemented in Part 1 to write data to stdout. The file descriptor for stdin is STDOUT_FILENO, which is equal to 1. To read keyboard input, use the *Fifo* class implemented in Part 1. This program should create a *Fifo* object named “teleop”, read a single char, construct a *rix::msg::geometry::Twist2DStamped* message corresponding to the key pressed, serialize the message into a byte array, and write the byte array prefixed with its size to stdout. This program should run until SIGINT is received. **Do not** write any drive commands after the signal is received.

The *Twist2DStamped* message contains a *rix::msg::standard::Header* with a sequence ID *seq*, the name of the coordinate frame of the twist *frame_id*, and the timestamp from when the

message was created *stamp*. The frame ID of the messages should be “mbot”, the sequence ID should increment every time a new message is sent (starting with 0), and the timestamp should be set to the time when the message was created (use *rix::util::Time::now()*).

Use the *size()* function of the *Twist2DStamped* class to calculate the total size of the serialized message after setting the message fields. Write the total size of the serialized message as a 32-bit (4-byte) unsigned integer (see *rix::msg::standard::UInt32*) to stdout before writing the serialized message. The *mbot_driver* will use this to know precisely the number of bytes to read for each command. The stdout buffer should look like the following diagram.

| $n_1$ (4 bytes) | serialized *Twist2DStamped* ($n_1$ bytes) | $n_2$ (4 bytes) | serialized *Twist2DStamped* ($n_2$ bytes) | … |
|---|---|---|---|---|

| **Feature** | **Points** |
|---|---|
| Program terminates when SIGINT is received and does not send a drive command after receiving the signal. | 2 |
| Program translates valid keys to twist commands, writes serialized *rix::msg::geometry::Twist2DStamped* messages prefixed with their total size. and returns when end-of-file (EOF) is reached. | 2 |
| Program ignores invalid keys. | 2 |

The usage of the *teleop_keyboard* executable is as follows.
The command to drive with a linear speed of 0.5 m/s and 1.5 rad/s is also included below.

```
./teleop_keyboard [--angular_speed (-a)] [--linear_speed (-l)]
./teleop_keyboard -a 1.5 -l 0.5
```

Other than unit tests, running this code in your Linux won’t give any verifiable result. You need to run this command in the MBot’s terminal to check if it works.

To drive your MBot with keyboard command, you need to follow the underlying instructions.
1. Open one terminal and combine *teleop_keyboard* and *mbot_driver* with a pipe.

```
./teleop_keyboard -a 1.5 -l 0.5 | ./mbot_driver
```

2. To write data to the FIFO, open another terminal and use one of the two commands.

```
echo “<char>” > teleop
tee teleop > /dev/null
```

For example, to add `W` as an input,

1. Use `echo “W” > teleop`
   or

2. First enter `tee teleop > /dev/null` in your terminal. When prompted, type `W` and press `Enter`

# Building and Testing

Before building the project for the first time, you must create a build directory.

```
mkdir build
```

Then, navigate to the build directory, initialize CMake, and compile the project. Optionally, you can specify the name of the target to compile to the `make` executable. This is useful if you want to test a single part of the project when other parts may not compile.

```
cd build
cmake ..
make [target name]
```

There are 2 libraries and 2 executables that are built for this project. Their target names are *mbot*, *project1*, *mbot_driver*, and *teleop_keyboard* respectively.

There are 6 available unit tests: *messages_test*, *serialization_test*, *signal_test*, *file_test*, *fifo_test*, and *pipe_test*. These tests correspond to the tests on the autograder. To run these tests, enter the command corresponding to the test that you want to run after compiling.

```
./<test_name>
```

There are 2 tests that are public on the autograder that you **do not** have available in the stencil code. These are the tests for the *mbot_driver* and *teleop_keyboard*. You should test your code on the MBot according to the specifications described in Part 3 of this project.

## Building Project 1 Natively on MacOS

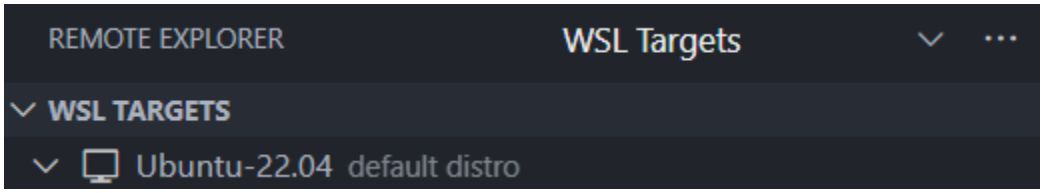
Ensure you have a C++ compiler available locally on your mac. Running `install_gtest.sh` should display your compiler. Follow the following steps:

- Open the folder containing your project
- Open a terminal in the project directory
- Run `sh install_gtest.sh`
- Create a build folder and cd into it
- Run `cmake .. && make -j6`

## Building Project 1 Natively on WSL

Follow the following steps:

- In VSCode, go to Remote Explorer → WSL Targets




- Connect to the Ubuntu version you want
- Open the folder containing your project
    - If in your Windows directory, go to /mnt/c/Users/<path-to-directory>
- Open a terminal
- Run `sh install_gtest.sh`
- Follow the rest of the Building and Testing instructions

# Appendix

## Blocking vs. Non-Blocking IO with FIFOs

“When a process opens one end of a FIFO, it blocks if the other end of the FIFO has not yet been opened. Sometimes, it is desirable not to block, and for this purpose, the O_NONBLOCK flag can be specified when calling *open()*. If the other end of the FIFO is already open, then the O_NONBLOCK flag has no effect on the open() call–it successfully opens the FIFO immediately, as usual. The O_NONBLOCK flag changes things only if the other end of the FIFO is not yet open, and the effect depends on whether we are opening the FIFO for reading or writing.” (Kerrisk, pg. 915-16). This behavior is described in the table below.

Table 2: Semantics of open() for a FIFO (Kerrisk, pg. 916)

| **Type of *open()*** | | **Result of *open()*** | |
|---|---|---|---|
| **open for** | **additional flags** | **other end of FIFO open** | **other end of FIFO closed** |
| reading | none (blocking) | succeeds immediately | blocks |
| | O_NONBLOCK | succeeds immediately | succeeds immediately |
| writing | none (blocking) | succeeds immediately | blocks |
| | O_NONBLOCK | succeeds immediately | succeeds immediately |

“The O_NONBLOCK flag affects not only the semantics of *open()* but also–because the flag then remains set for the open file description–the semantics of subsequent *read()* and *write()*

calls.” (Kerrisk, pg. 917). The behavior of blocking and non-blocking read and write operations are described in the tables below.

Table 3: Semantics of reading *n* bytes from a pipe or FIFO containing *p* bytes (Kerrisk, pg. 918)

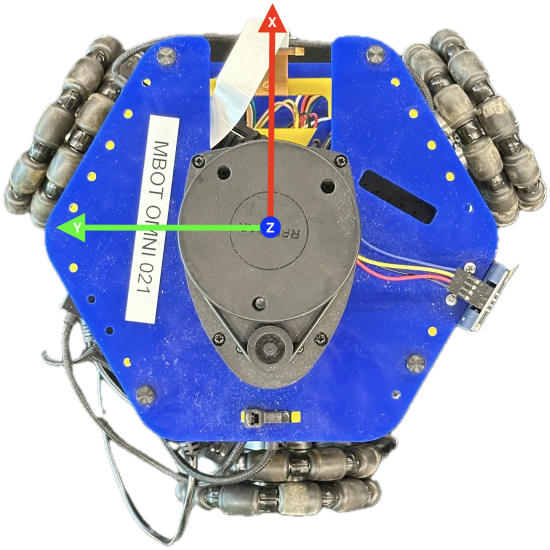
| **O_NONBLOCK enabled?** | **Data bytes available in pipe or FIFO (p)** | | | |
|---|---|---|---|---|
| | ***p* = *0*, write end open** | ***p* = *0*, write end closed** | ***p* < *n*** | ***p* >= *n*** |
| No | block | return 0 (EOF) | read *p* bytes | read *n* bytes |
| Yes | fail (EAGAIN) | return 0 (EOF) | read *p* bytes | read *n* bytes |

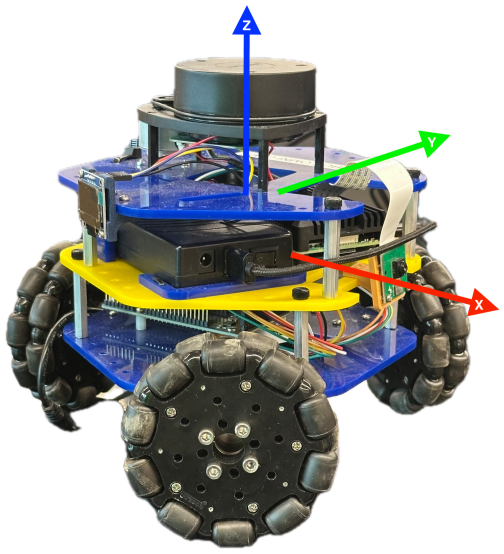
Table 4: Semantics of writing *n* bytes to a pipe or FIFO

| **O_NONBLOCK enabled?** | **Read end open** | | **Read end closed** |
|---|---|---|---|
| | ***n* <= *PIPE_BUF*** | ***n* > *PIPE_BUF*** | |
| No | Atomically write *n* bytes; may block until sufficient data is read for *write()* to be performed. | Write *n* bytes; may block until sufficient data is read for *write()* to complete; data may be interleaved with writes by other processes | SIGPIPE + EPIPE |
| Yes | If sufficient space is available to immediately write *n* bytes, then *write()* succeeds atomically; otherwise, it fails (EAGAIN). | If there is sufficient space to immediately write some bytes, then write between 1 and *n* bytes (which may be interleaved with data written by other processes); otherwise, *write()* fails (EAGAIN). | |

# The MBot Omni Coordinate Frame

The coordinate frame of the MBot is depicted below. It is a right-handed coordinate frame, so a positive rotation is counterclockwise. The positive x-axis points outward from the camera, the positive z-axis points up from the LiDAR, and the positive y-axis points from the battery to the Raspberry Pi perpendicular to the x-axis.




## The Linux Man-Pages

If you encounter an unfamiliar system call, or you want to see the semantics of C standard library functions, please refer to the Linux man-pages. The Linux man-pages project documents the Linux kernel and C library interfaces that are employed by user-space programs. They are organized into 8 sections.

1. User commands
2. System calls
3. Library functions
4. Special files
5. File formats and filesystems
6. Games
7. Miscellaneous
8. Administration and privileged commands

For this project, you will only need to use sections 2 and 3, however, there is interesting and useful information in all sections.

You may also use the *man <section number> <name>* executable in your command line to access these definitions in your terminal rather than searching the internet. For example, when you enter *man 2 open* in the command line, the man page entry for the *open* system call is displayed. If you were to enter *man 1 cat*, then the man page entry for the *cat* executable is displayed.

## Signal Safety

See the discussion of signal safety from the Linux man-pages section 7: https://www.man7.org/linux/man-pages/man7/signal-safety.7.html.

## Files to Modify

The following files contain TODOs and must be submitted to the autograder.

1. include/rix/msg/serialization.hpp
2. src/rix/ipc/file.cpp
3. src/rix/ipc/pipe.cpp
4. src/rix/ipc/fifo.cpp
5. src/rix/ipc/signal.cpp
6. src/mbot_driver.cpp
7. src/teleop_keyboard.cpp

## Code Formatting

Each project in this course includes a file named *.clang-format* in the root directory. If you have the *.clang-format* file in your VS Code workspace directory, you can press ctrl + alt + f to auto-format your code. Please ensure that your code is readable! It can be very difficult for the course staff to help you if they cannot read your code.