Brief Operating Manual for Miro Robot

Written By: James Zhu

April 2018

Table of Contents

[1. Introduction 2](#_Toc513160293)

[2. Preparation 2](#_Toc513160294)

[3. START UP 2](#_Toc513160295)

[4. INterfaces 3](#_Toc513160296)

[A. Platform Interface 3](#_Toc513160297)

[B. Bridge Interface 3](#_Toc513160298)

[5. Writing Code 4](#_Toc513160299)

[A. Reading Sensor Data on Platform Interface 4](#_Toc513160300)

[B. Writing Commands on Platform Interface 4](#_Toc513160301)

[C. Reading Mircophone Data 5](#_Toc513160302)

[D. Bridge Interface 5](#_Toc513160303)

[E. Running Programs 6](#_Toc513160304)

[F. Behavior Integration 6](#_Toc513160305)

[6. Conclusion 7](#_Toc513160306)

# Introduction

MIRO is an animal-like interactive robot developed by Consequential Robotics. The Robotics and Autonomous Systems Laboratory at Vanderbilt University has been developing the robot to potentially be used by the elderly to decrease loneliness that can often be experienced in nursing homes. This manual will cover the basic operating procedure for the MIRO robot including start up, sensor reading, behavior integration, and some typical troubleshooting tips. The manual will be presented using the off-board profile and all sample code is written using Python. Additional information about operating MIRO can be found on <https://consequential.bitbucket.io/> and some demo programs andbehaviors can be found on <https://github.com/partlygloudy/RASL-MIRO>. The demo behavior was written by Jake Gloudmans and James Zhu.

# Preparation

If the workstation has already been prepared to interact with MIRO, skip this section. Prior to operating MIRO, ensure that a computer with Linux OS is available. Ubuntu 16.04 LTS is the currently recommended workstation operating system. The workstation will also need to be equipped with ROS (Kinetic is recommended, but other current versions should be fine). Additionally, the MIRO Developer Kit (MDK) should be downloaded from <http://labs.consequentialrobotics.com/miro/mdk/>. Once everything is installed and downloaded, edit the file *~/.profile* by adding the following lines:

* *# configuration*

*export MIRO\_PATH\_MDK=~/mdk*

*export ROS\_IP=[WORKSTATION\_IP]*

*export ROS\_MASTER\_URI=http://localhost:11311*

*# usual ROS setup*

*source /opt/ros/kinetic/setup.bash*

*# make our custom messages available to ROS/python*

*export ROS\_PACKAGE\_PATH=$MIRO\_PATH\_MDK/share:$ROS\_PACKAGE\_PATH export PYTHONPATH=$MIRO\_PATH\_MDK/share:$PYTHONPATH*

Note: replace [WORKSTATION\_IP] with the appropriate local IP address of the work station. This can be found using the command *ifconfig*.

# START UP

Ensure that the local IP address of the workstation as well as the IP address of the MIRO is known. The global IP address of the MIROs can be found using the MIRO app on an Android device. In a command terminal, run roscore with the command *roscore*. If an error state is delivered, a common solution might be to manual reset the ROS IP with the command:

* *export ROS\_IP=[WORKSTATION\_IP]*

Once roscore is running, turn on MIRO, open another terminal and login to MIRO using the command:

* *ssh root@[MIRO\_IP]*

For RASL, the password is *!amMIRO*. Once logged in, it is critical to ensure the Master IP is set to the workstation IP address. This can be done by the command *vi .profile* and editing the *ROS\_MASTER\_IP* if necessary. Note that if the Master IP is changed, MIRO must be restarted to update the new Master IP. Once this is correct, run the MIRO bridge with the command: *run\_bridge\_ros.sh*.

After this is completed, MIRO can be run in a separate command terminal with an appropriate program. Writing code and programs for MIRO will be explained in the following sections.

# INterfaces

Miro is equipped with several interfaces that can communicate to the workstation. Each interface relays different input and output signals between MIRO and the workstation. This manual will describe the platform interface and bridge interface, as they are the most applicable for developer usage. Example code for integrating each interface will be shown in the next section. For additional information about the other interfaces, see <https://consequential.bitbucket.io/Technical_Interfaces.html>.

## Platform Interface

The platform interface provides access to most of the sensors and actuators of the robot. Sensors that can be read from the platform interface include temperature, distance, microphone, and touch sensors. Control signals include body movement, head movement, blinking, and tail movement. A comprehensive list of capabilities can be found at <https://consequential.bitbucket.io/Technical_Interfaces_Platform_Interface.html>.

## Bridge Interface

The bridge interface can be used to play custom, preloaded sound files from the SD card. Audio files must be signed 8-bit files loaded to */home/root/sound* of the SD card. The file *sound.ini* also should be edited so that it consists of a list of the sound files to be played by MIRO. The files in *sound.ini* will be indexed beginning at *1*, so in code, each audio file will be called by its index number. For more information, see <https://consequential.bitbucket.io/Technical_Interfaces_Bridge_Interface.html>.

# Writing Code

## Reading Sensor Data on Platform Interface

Consider the following Python program that accesses the platform interface, excerpted from *RASL-MIRO/Python/main/interfaces.py* of the GitHub.

*from miro\_msgs.msg import platform\_control, platform\_sensors*

*from geometry\_msgs.msg import Twist*

The *platform\_control* and *platform\_sensors* packages are necessary to read data and write commands to and from MIRO through the platform interface. *Twist* will be explained later but is important for the movement command.

*class \_\_primary\_interface:*

*def \_\_init\_\_(self, robot\_name):*

*root = "/miro/" + str(robot\_name) + "/platform"*

*self.cmd\_out = rospy.Publisher(root + "/control", platform\_control, queue\_size=1)*

*self.data\_in = rospy.Subscriber(root + "/sensors", platform\_sensors, self.data\_in, queue\_size=1)*

*Self.cmd\_out* will publish commands to MIRO and *self.data\_in* reads data sent from MIRO.

*def data\_in(self, data):*

*self.battery\_voltage = data.battery\_voltage*

*self.temperature = data.temperature.temperature*

*self.sonar\_range = data.sonar\_range.range*

*self.light = list(array("B", data.light))*

*self.touch\_head = list(array("B", data.touch\_head))*

*self.touch\_body = list(array("B", data.touch\_body))*

*self.cliff = list(array("B", data.cliff))*

The above are examples of data that can be read from the platform interface. Some of the most useful are *self.sonar\_range*, which detects if an object is in front of MIRO, *self.cliff* which detects if MIRO is approaching a “cliff”, and *self.touch\_head* and *self.touch\_body* which detect if the MIRO is being touched. Note that some data is given as a single value while others are represented by arrays with 2 or 4 values. Again, see <https://consequential.bitbucket.io/Technical_Interfaces_Platform_Interface.html> for more information.

## Writing Commands on Platform Interface

The following code is a continuation of *interfaces.py* shown in the previous section.

*# Output new control command*

*cmd = platform\_control()*

*cmd.body\_vel = self.body\_vel*

*cmd.body\_move = self.body\_move*

*cmd.tail = self.tail*

*cmd.ear\_rotate = self.ear\_rotate*

*cmd.eyelid\_closure = self.eyelid\_closure\_out*

*cmd.sound\_index\_P1 = self.sound\_index\_P1*

*cmd.sound\_index\_P2 = self.sound\_index\_P2*

*self.cmd\_out.publish(cmd)*

Examples of several helpful commands are shown above, including movement, tail wagging, and sound production.

*def update\_body\_vel(self, linear, angular):*

*twist = Twist()*

*twist.linear.x = linear \* 1000.0*

*twist.angular.z = angular*

*self.body\_vel = twist*

Each command may accept different data types. One of the most important is the *Twist* data type that is required for the movement command. This data type stores a value for both the linear speed and angular speed of the MIRO.

## Reading Mircophone Data

Microphone data is collected from each ear at a rate of 20 kHz with 10 bits of resolution. It is read in a slightly different way then other sensors. The package *platform\_mics* is used to read audio data that is stored in 16 bit containers that interleave both left and right mic data. The program *RASL-MIRO/Python/main /mic\_test.py* is a useful sound localization algorithm that also demonstrates the process of basic audio signal processing.

## Bridge Interface

The following example code is taken from *RASL-MIRO/Python/main /sound\_test.py* of the GitHub.

*import rospy*

*from miro\_msgs.msg import bridge\_stream*

*class sound\_test:*

*def \_\_init\_\_(self):*

*root = "/miro/" + "rob01" + "/bridge"*

*self.cmd\_out = rospy.Publisher(root + "/stream", bridge\_stream, queue\_size=1)*

The bridge\_stream package is used to send sound commands through the *bridge\_stream*.

*def data\_in(self):*

*start\_time = rospy.get\_rostime()*

*while (rospy.get\_rostime() - start\_time).to\_sec() < .3:*

*cmd = bridge\_stream()*

*cmd.sound\_index\_P3 = 3*

*self.cmd\_out.publish(cmd)*

To play a sound, the index value of the appropriate sound file simply needs to be published. In practice, however, there seems to be a lag where the MIRO will not receive the message if it is just sent in one instance. This code pulses the index value for a brief time (.3 seconds) so that MIRO is able to receive it and play the sound.

## Running Programs

If *roscore* and the MIRO bridge are each running, a Python program can be run in a third terminal using the command:

* *python [my\_program].py*

A problem may occur where MIRO does not respond to any commands sent to it. This can likely be solved by reseting the *ROS\_IP* of the terminal with:

* *export ROS\_IP=[WORKSTATION\_IP]*

## Behavior Integration

The ultimate goal of MIRO is to act similarly to a typical pet companion such as a dog or cat. This, of course, encompasses many active and passive behaviors such as displaying emotion and facilitating social interaction. The current behavior model that has been developed includes the *brain.py, flags.py, interfaces.py, and moods.py* programs that can be found on the GitHub. The basic structure of this model is that interfaces.py reads and writes directly to all of the interfaces of MIRO. Data is sent to *flags.py* where if a certain stauts is achieved (low light, loud sound, etc.) then a flag will be sent to *brain.py* which in turn sends a message to *moods.py* to perform some sort of responsive behavior. This behavior command will be sent back to *interfaces.py* and the MIRO will behave as told. This model is quite flexible for integrating different behaviors and potentially adding new, more complex actions for MIRO to perform.

Brain.py

Interfaces.py

Flags.py

Moods.py

# Conclusion

This manual gave a rundown of many of the capabilities that MIRO has. There are certainly more that this robot can do and the reader is encouraged to explore the resources stated above to further develop MIRO. Any questions about this operating manual can be sent to [james.w.zhu@vanderbilt.edu](mailto:james.w.zhu@vanderbilt.edu) and any higher level questions about MIRO can be sent to [labs@consequentialrobotics.com](mailto:labs@consequentialrobotics.com).