BRIEF OPERATING MANUAL FOR MIRO ROBOT

WRITTEN BY: JAMES ZHU
APRIL 2018
EXTENDED AND EDITED BY: SIDHARTH BABU
JULY 2018

TABLE OF CONTENTS

1. 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.

2. 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*.

3. 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.

4. 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.

A. 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.

B. 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.

5. WRITING CODE

A. 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.

B. 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.

C. Reading Microphone 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.

D. 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.

E. Running Programs

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

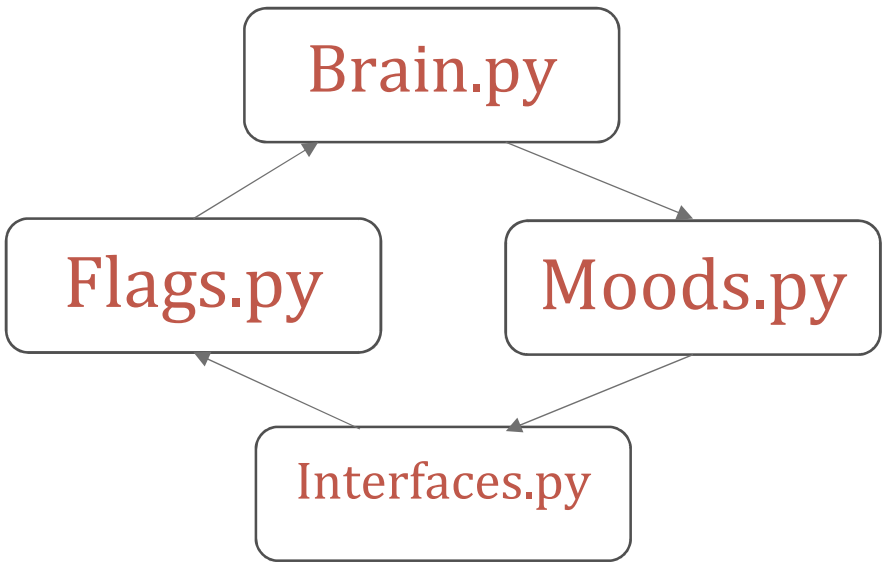
- *python [my_program].py*

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]*

F. 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.



6. USING THE RASL-MIRO SENSOR REACT PACKAGE (AS OF JULY 2018)

The Rasl-Miro Sensor React Package is a framework, based off of code developed by James Zhu and Jacob Gloudemans, built by Sidharth Babu. It was created to make it easier to use the MIRO robot, and James and Jacob's handling code. It is a collection of scripts that allows for easily implementing sensor reactive programs. It consists of 3 main files: Initiator.py, PrimaryHandler.py, interfaces.py, and (The Package can be found at this github link: https://github.com/babusid/RASL-MIRO_Sensor_React).

- Interfaces.py is used for directly interacting with the robots sensors. It also handles all the ROS message handling, and provides many methods that make it much easier to interact with the robot compared to individually modifying variables within the class. Examples include drive_straight() and tail_move(). Using these methods in conjunction with sensor readings allow one to develop a reactive robot relatively quickly. This is the most important file for interacting with the Miro directly. It has a primary interface class which is the base for the Primary Handler python file, as well as other interfaces for use with the cameras and microphones
- The second file is PrimaryHandler.py. This is the file one should edit to develop a reactive software. Within the file, there is a method (sensor_interrupt()). As found in the packages stock state, when the Miro is touched it will execute a headnod and a tail wag action. The if statements within this method are what one would have to modify to develop a reactive program. The method is present within a class called SecondaryInterface. This class, when constructed, allows one to give a default state of motion to the robot, which sensor_interrupt will revert to whenever there is no sensor input.
- The third file is Initiator.py. This file is the executor file, and all it does is create a ROS node for the robot, and create an instance of the SecondaryInterface class. The only thing one should edit here, is the arguments of the SecondaryInterface constructor, so that they can change their state of default motion.

In overview, the main quick development should be done within PrimaryHandler.py which is the ideal place to put together a reactive program using already declared methods and variables if needed. If you need to simplify a complex set of actions down to a method, that should be done in the interfaces.py file. And Initiator.py is the execution file where you declare the default state of motion for the robot.

The files already have a default behavior coded within them. If you run Initiator.py with both files as they are given now, the MiRo will move around the room with a basic obstacle avoidance routine, whilst moving its ears for aesthetic effect. Furthermore, when stimulated through either set of touch sensors (head or body), it will respond through a dedicated movement for each stimulus.

7. 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 and any higher level questions about MIRO can be sent to labs@consequentialrobotics.com.