STUDENT SUMMER INTERNSHIP TECHNICAL REPORT

Autonomous Navigation and Radiation Mapping Platform - Radiation Sensor Package Development

DOE-FIU SCIENCE & TECHNOLOGY
WORKFORCE DEVELOPMENT PROGRAM

Date submitted:

December 10, 2021

Principal Investigators:

Thi Tran (DOE Fellow Student)
Florida International University

Alexander Pappas (Mentor)
Washington River Protection Solutions

Ravi Gudavalli Ph.D. (Program Manager)
Florida International University

Leonel Lagos Ph.D., PMP® (Program Director)
Florida International University

Submitted to:

U.S. Department of Energy
Office of Environmental Management
Under Cooperative Agreement # DE-EM0005213

Applied Research Center
FLORIDA INTERNATIONAL UNIVERSITY

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

This research work has been supported by the DOE-FIU Science & Technology Workforce Initiative, an innovative program developed by the U.S. Department of Energy's Office of Environmental Management (DOE-EM) and Florida International University's Applied Research Center (FIU-ARC). During the summer of 2021, a DOE Fellow intern, Thi Tran, spent 12 weeks doing a summer internship at Washington River Protection Solutions under the supervision and guidance of Technology Integration Project Manager Alexander Pappas. The intern's project was initiated on May 17, 2021 and continued through August 5, 2021 with the objective of supporting the development of an autonomous radiation mapping robot by developing and integrating a radiation sensor package, as well as designing mount solutions for other sensor packages for the existing Clearpath Robotics® Husky^{TM}.

To support the commitment of Washington River Protection Solutions (WRPS) to Hanford's cleanup mission, the Chief Technology Office (CTO) continuously works to provide solutions that improve the safety and efficiency of operations through the maturation of technology. The research describes the improvement of the off-the-shelf robotics platform, the Clearpath Robotics® Husky (referred to as the Canary), which was equipped with a new sensor package as part of the effort of CTO to bolster workflow through autonomous radiation mapping. As part of the effort, the development of a radiation sensor package was necessary to aid in planar radiation mapping. Radiological mapping is essential in providing awareness to possible contamination and minimizing human exposure to radioactive dose. The radiation package was developed using two approaches: Jetson Nano and microcontroller (Arduino) interface, and direct sensor interface to a Raspberry Pi 4. Both methods provided data readings, but the direct interface with the Raspberry Pi 4 was selected due to the more robust and stable connection for relaying the radiation data to the on-board computer on the Canary.

1. INTRODUCTION

With the urge to fulfill the cleanup mission at the Hanford site, the Chief Technology Office (CTO) at Washington River Protection Solutions (WRPS) continuously develops, matures, and implements technology to increase safety and efficiency in operations. Within this effort, CTO pursues development and maturation of commercially available robotic platforms for tasks identified through stakeholder engagement. Despite the available robots on the commercial market, there is still a lack of autonomous robotic systems to decrease human exposure to radioactive waste, as well as aid in bolstering data acquisition and supporting retrieval operations,.

Within the available robotics platform at CTO, Clearpath Robotics® Husky^{TM} (also known as the Canary) was chosen to fully develop an autonomous radiation mapping platform. The Husky platform initially came with only cameras and multiple inertial measurement units (IMUs). It was not, however, equipped for hands-off operation or radiological mapping. Additional compute units and sensors were incorporated on the Canary to enhance its capabilities in autonomous navigation and localization.

In addition, the scope of work also aimed to provide the Canary with autonomous radiation mapping capability. Radiological mapping allows the measurement the radioactivity level of contamination areas and provides awareness of the local dose rates. With the extent of sensor packages and fusing odometry data from multiple sensors, it enables high accuracy mapping of the area to facilitate more efficient monitoring of the dose rate of facilities at Hanford site. With all these benefits, deployment of radiation mapping capabilities on the Husky would have a significant impact on the Hanford clean-up mission.

2. MOUNTING SOLUTION DEVELOPMENT

A new sensor package which included two light detection and ranging (LIDAR) - the SLAMTEC® RPLIDAR A2 and the Intel® RealSense^{TM} LIDAR Camera L515- and three Intel® RealSense™ depth cameras – D415, D435, and D455, was included in the Husky. These stereo cameras with their depth sensing capabilities, allow extracting 3D information from the scene and generating point cloud. In conjunction to depth cameras, the two LIDARs were utilized for planar detection and motion planning. This package enabled an improvement in the navigation capabilities and localization of the Canary, not only in tight space, but also larger and complex space while producing a detail maps of the environment around. Additionally, this package was essential to the integration of radiation sensor package, which would be discussed in Section 3, in providing an accurate location and visualization of the local dose rate measure.

Due to the difference in field of view and range covering amongst Realsense depth cameras, consideration was made regarding to the placement of the sensor. For the purpose of frontier navigation, all three depth cameras were placed facing forward to capture the most features around. The biggest depth sensor, D455, with a wider baseline, provided better accuracy compared to the D415 and D435. Thus, D455 was mounted facing directly forward and nearer to the ground. The other cameras, D415 and D435, were placed on the sides at an angle of 45 degrees to cover a wider area. Meanwhile, the solid-state LIDAR was mounted on top of the platform, also at an angle of 45 degrees facing the rear. Lastly the RPLIDAR was mounted on top of the batteries to provide obstacle avoidance for the Husky. The mounting solutions for the depth sensor were designed using SolidWorks.

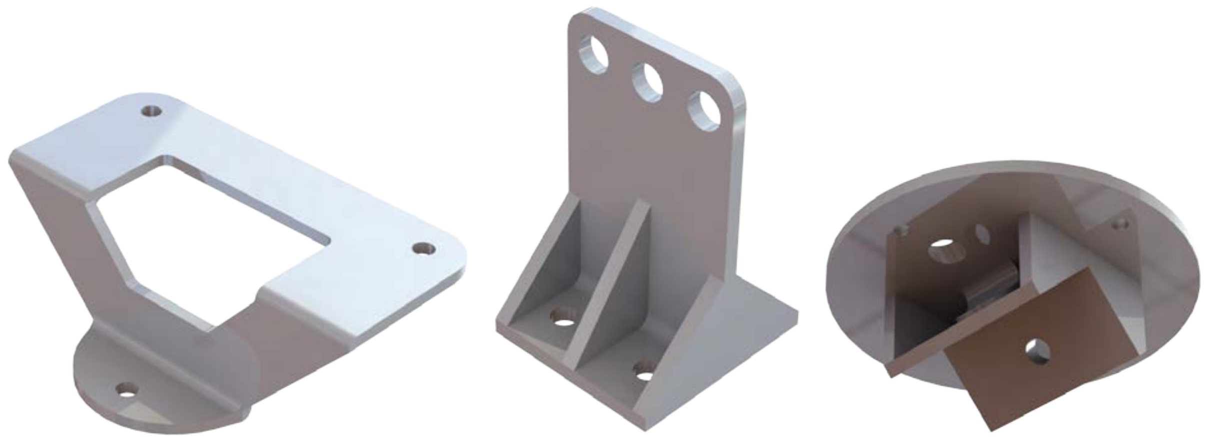


Figure 1. Mounting solutions for depth cameras and LIDARs.

3. RADIATION SENSOR PACKAGE DEVELOPMENT

3.1 Setup and Integration of Additional Compute Units

A key component of the development of radiation sensor package was the communication and transmission of data amongst the radiation sensor package, the on-board computer on the Canary, and the additional sensor package for navigation, as described in Section 2. To deal with the massive influx of data coming from the additional sensors and LIDARs, two compute units, NVIDIA® Jetson Xavier™ and Jetson Nano™, were added to the configuration of the Husky. The two computers were beneficial to the system due to their powerful performance at compact size, which was ideal for critical embedded applications. The Husky on-board computer runs on Robot Operating System (ROS), which is an open-source framework for developing robotic applications. To integrate the new compute units on the Canary, ROS Melodic was installed to establish a common baseline with the onboard computer that was also updated to Melodic.

A running ROS system can comprise an infinite amount of nodes, spread across multiple machines. Communication among Husky configurations was achieved by setting the configuration to the same master, the Husky on-board computer, via ROS_MASTER_URI. Each machine was set up to advertise itself by a recognizable name and IP address. This allowed for easier access to the NVIDIA devices from the main computer via Secure Shell Protocol (SSH). As a result, sensors like the depth cameras installed on the Jetson Xavier on the common network could be accessed and computing load could be distributed.

Later on, the Jetson Nano was replaced with a Raspberry Pi 4 to power the radiation sensor board, retrieve the data stream from the sensor, and run the solid-state LIDAR. Although the Raspberry Pi 4 was not as powerful as the Jetson Nano, the Pi provided more stability, reliability, and robustness for communicating and data transmission from the radiation package. Although the setup for the Pi 4 was similar to others, it differed slightly since it uses Raspbian distribution instead of Ubuntu.

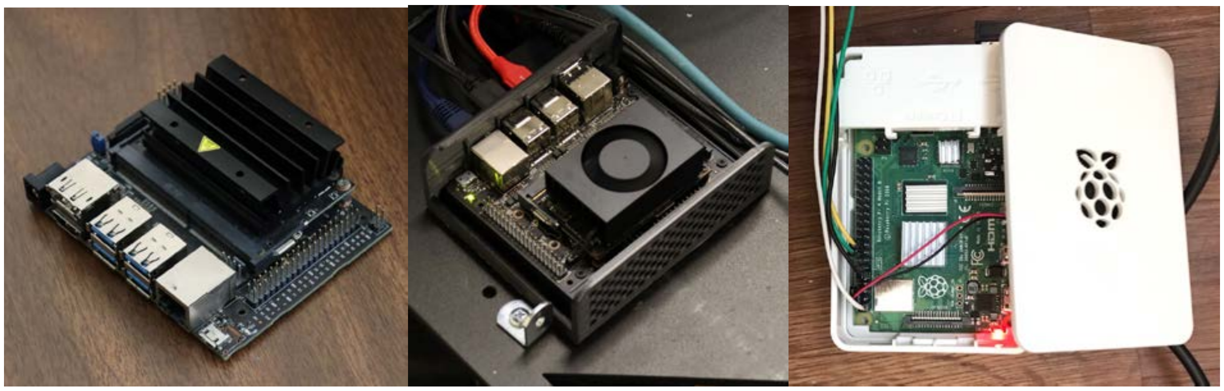


Figure 2. Additional compute units.

3.2 Development of Radiation Sensor Package

The main goal of this project was to develop a radiation mapping system. To achieve this goal, a radiation sensor package was developed for integration into the Husky’s system. The radiation sensor package used a Type 5 Pocket Geiger Radiation Sensor, which was designed for embedded systems. It features a measurement range of 0.05uSv/h to 10mSv/h at 0.01cpm to 300 Kcpm with a required measurement time of two minutes. The Pocket Geiger radiation sensor board has four pins: two for alimentation (+V, GND) and two for detecting signals (SIG, NS). The sensor pulls the radiation pin (SIG) to a high voltage level when it detects radiation. The board uses an X100-7 PIN photodiode from FirstSensor for gamma-ray detection; however, its photodiode sensor is sensitive to noise. Thus, the board also comes with an accelerometer to detect corresponding false positives noticed through the noise pin (NS). There are currently two mediums to interface with the Pocket Geiger board: Arduino library and Raspberry Pi library. Both interfaces were tested to determine which would provide more stability and be compatible with ROS.

3.2.1 Using Arduino Nano

Table 1. Connection between Radiation Sensor and Arduino Nano

Pocket Geiger Pin	Arduino Nano Pin	Standing for
+V	3V3	Power Supply (3.3V)
GND	GND	Ground
SIG	2	Radiation-detection pulse pin
NS	3	Noise-detection pulse pin

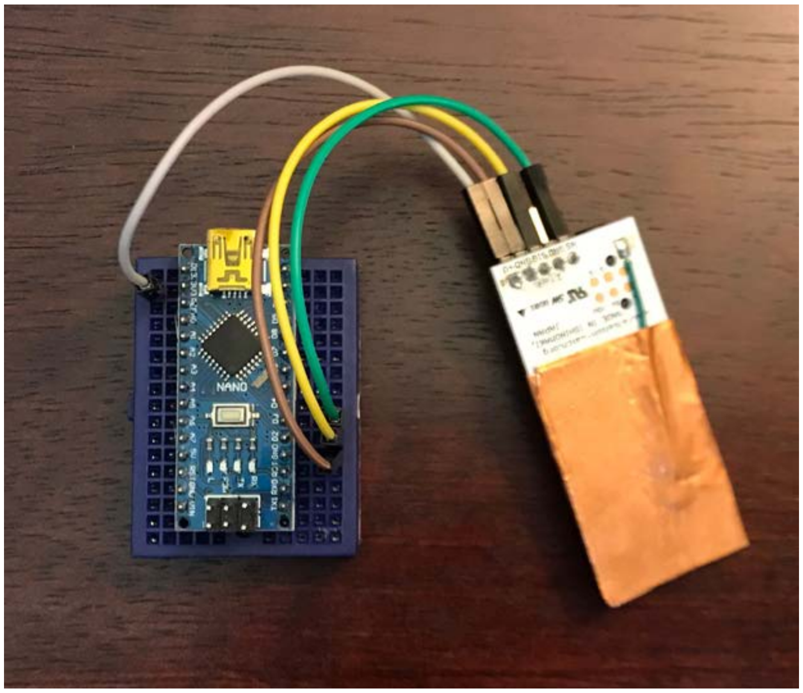


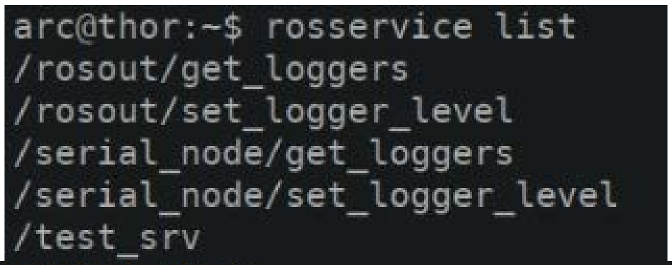
Figure 3. Set up the Pocket Geiger radiation sensor on the Arduino Nano.

For the nodes to communicate with each other, the *rosserial* package and library were installed on the Jetson Nano and Arduino Nano respectively. The package contained necessary extensions to integrate custom hardware like the radiation board into the ROS system using an Arduino. The

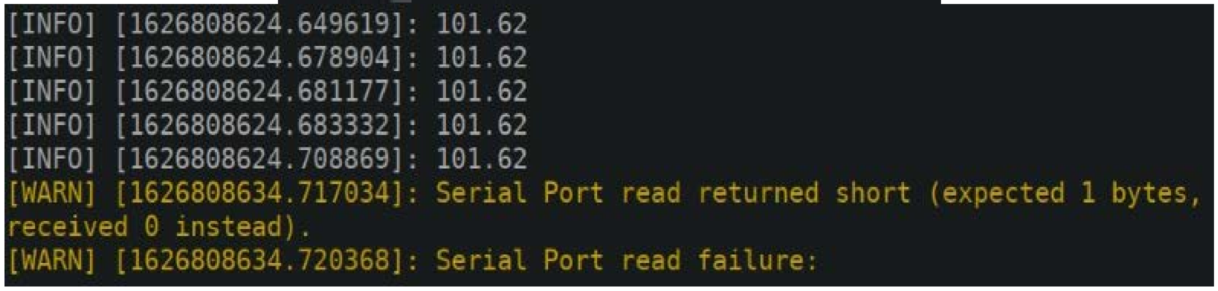
Arduino Nano was connected to the Jetson Nano via the USB serial port. Since there was no direct connection from the Arduino Nano to the ROS network, the *rosserial* package provided a node as a bridge between the two.

When the Husky platform navigated autonomously, a state machine was implemented into its path to call for service or action to take readings from the sensor. Service and action both perform a call to a different function; however, an action is asynchronous, meaning it does not block other ROS processes. An action would be more beneficial for longer tasks and provide feedback during the execution. Using an action would be ideal, but since *rosserial* Arduino does not support ROS action, ROS service was used instead.

Once the service received a request from the client, the Husky, it started setting up the radiation sensor and taking readings. The Pocket Geiger requires time to get a reliable reading. After a few test runs, it was observed that the sensor becomes stable when the error reading reaches 0.01. Therefore, the service was modified to send the first radiation reading to the state machine when the corresponding error falls within the range of 0.00 to 0.01. The finalized version of the code can be found in Appendix A.



```
arc@thor:~$ rosservice list
/rosout/get_loggers
/rosout/set_logger_level
/serial_node/get_loggers
/serial_node/set_logger_level
/test_srv
```



```
[INFO] [1626808624.649619]: 101.62
[INFO] [1626808624.678904]: 101.62
[INFO] [1626808624.681177]: 101.62
[INFO] [1626808624.683332]: 101.62
[INFO] [1626808624.708869]: 101.62
[WARN] [1626808634.717034]: Serial Port read returned short (expected 1 bytes,
received 0 instead).
[WARN] [1626808634.720368]: Serial Port read failure:
```

Figure 4. Result of running the code: test for available service (top) and logging radiation reading (bottom).

During initial testing, the service returned a usable result reading only once out of 20 test runs even though the service was available, as shown in the top figure above. This was due to the unstable nature of services on Arduino, which was still in an experimental state. The common error appeared to be the failure of reading the serial port. Troubleshooting was performed, such as varying the baud rate and buffer, setting the baud rate in the console, and tweaking the Python library. However, the issues persisted, so an alternative solution was sought. Implementation of a Python library to run the Pocket Geiger sensor was accessible for the Raspberry Pi 4, as such, the Jetson Nano and Arduino Nano were replaced by the Pi for further testing.

3.3 Using a Raspberry Pi 4

Like the Arduino setup, a library for the sensor was installed on the Raspberry Pi 4. The Pocket Geiger was wired on general-purpose input and output (GPIO) pin-out on the Raspberry Pi. As described in 3.2, the Raspberry Pi also needs to call the function to start setting and taking measurements of the sensor. *rospy*, a Python client library for ROS, and *actionlib*, a library for action, were included in the code for that purpose. The Raspberry Pi will act as an Action Server using multiple messages and service definitions that are generated from a custom action definition.

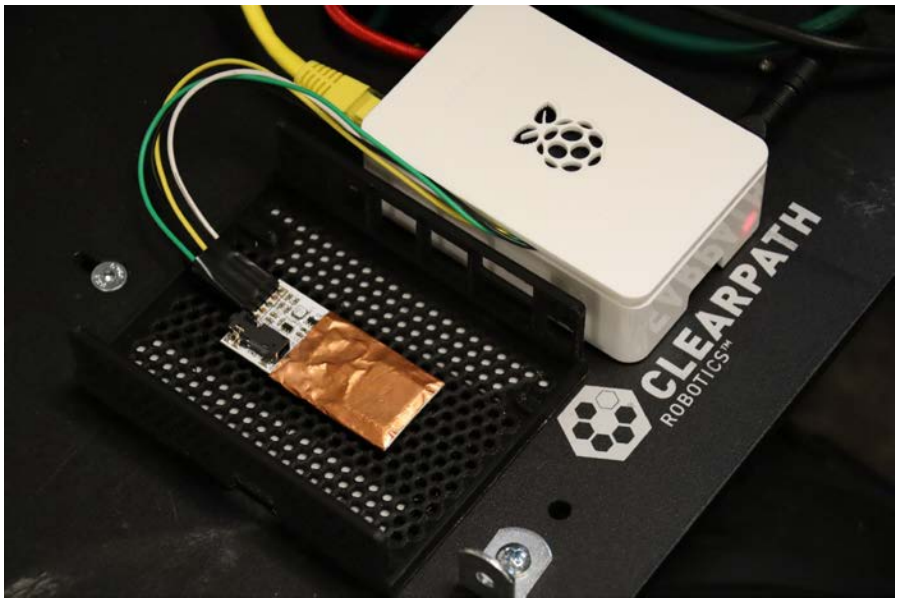


Figure 5. Set up of the Pocket Geiger on the Raspberry Pi 4.

Table 2. Connection between Radiation Sensor and Raspberry Pi 4

Pocket Geiger Pin	GPIO Pin	Standing for
+V	3V3	Power supply (DC 3.3V)
GND	GND	Ground
SIG	GPIO24	Radiation-detection pulse pin
NS	GPIO23	Noise-detection pulse pin

An action definition file contains three parts: goal, feedback, and result. Each part is a list of messages with field type and field line. In the above action definition, the Action Client (the State Machine) would send an empty goal to the Action Server (Raspberry Pi). As soon as the Server receives a request from the client, the Pi would start taking measurements and send the radiation reading and error to feedback. When the error reaches stability (from 0.0 to 0.02), the sensor will stop reading and return the radiation measurement to the result topic. The current version of the code can be found in Appendix A Section 2. The relationship of the Client/Server and how data is sent is shown more clearly in Figure 6. Figure 7 is a section of the feedback from the Raspberry Pi accepting the goal it receives from the Client, the State Machine. The feedback also showed the location of the Canary as well as the radiation and error measurements. Once the condition was met, the server sent the final radiation reading to the /result topic on the State Machine as shown in Figure 8.

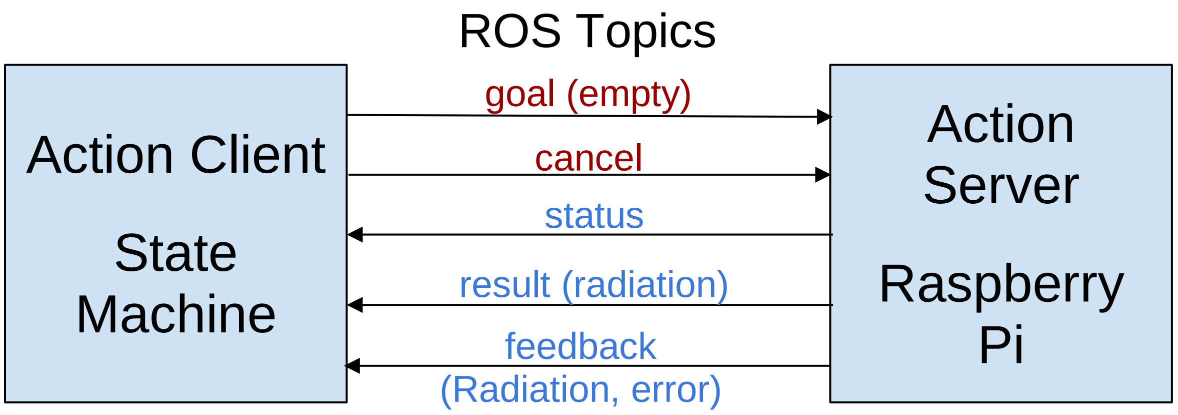
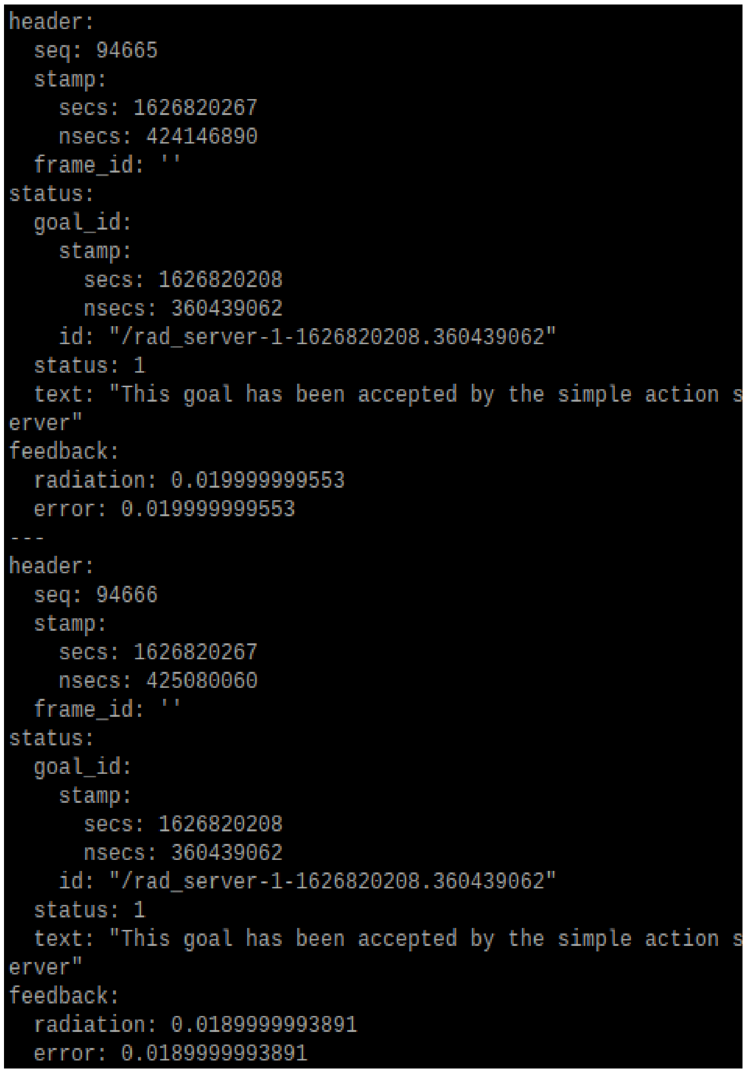
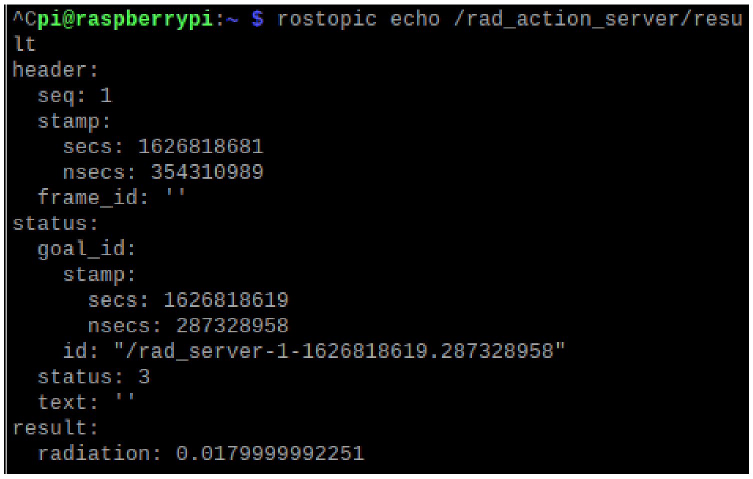


Figure 6. Diagram of action client and server interaction.



```
header:
  seq: 94665
  stamp:
    secs: 1626820267
    nsecs: 424146890
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1626820208
      nsecs: 360439062
    id: "/rad_server-1-1626820208.360439062"
  status: 1
  text: "This goal has been accepted by the simple action s
erver"
feedback:
  radiation: 0.019999999553
  error: 0.019999999553
---
header:
  seq: 94666
  stamp:
    secs: 1626820267
    nsecs: 425080060
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1626820208
      nsecs: 360439062
    id: "/rad_server-1-1626820208.360439062"
  status: 1
  text: "This goal has been accepted by the simple action s
erver"
feedback:
  radiation: 0.0189999993891
  error: 0.0189999993891
```

Figure 7. Feedback shows server accepted goal from client and publishes readings every 5 seconds.



```
^Cpi@raspberrypi:~ $ rostopic echo /rad_action_server/resu
lt
header:
  seq: 1
  stamp:
    secs: 1626818681
    nsecs: 354310989
  frame_id: ''
status:
  goal_id:
    stamp:
      secs: 1626818619
      nsecs: 287328958
    id: "/rad_server-1-1626818619.287328958"
  status: 3
  text: ''
result:
  radiation: 0.0179999992251
```

Figure 8. The server sent the radiation reading to the /result topic.

4. CONCLUSION

Throughout the development of the radiation sensor, different methods were used to test the data retrieval from the radiation sensor. In the first approach, the radiation sensor was powered using a microcontroller that was connected to the Jetson Nano computer via a USB Serial Port. Even though the computer was able to receive data from the sensor, the connectivity between devices easily got lost when the service in ROS was called, while for the second approach using GPIO on the Raspberry Pi 4, the connectivity was robust and data acquisition was successful at every reading.

The Pocket Geiger Radiation Sensor was recommended for use with the embedded systems such as Arduino, Raspberry Pi, etc. However, when it comes to integrating into the ROS system, using a Raspberry Pi would be a better option. Since the development of the ROS service in the *rosserial* library is still in an experimental state, publishing readings while checking for the requirement broke the connectivity amongst devices. Meanwhile, the Python environment in Raspberry Pi allowed the use of the ROS concept more efficiently and was not limited to the ROS service as in the first approach. Moreover, the Raspberry Pi bolstered consistency in retrieving data from the sensor since it did not require any bridge node. Thus, it would be ideal and more efficient to use a Raspberry Pi.

To conclude, the radiation sensor package was successfully developed to be integrated into any platform for radiological usage. There were several obstacles in obtaining data from the radiation sensor at the beginning, however, the second approach which used the Raspberry Pi pinout was able to provide much more stable connectivity and reliable readings. The team was able to power the package by using the power source on the Husky platform. Through the implementation of the package, the Husky successfully performed the radiation mapping of the farms at the Hanford site.

APPENDIX A.

Section 1: ROS Service

```
#include <ros.h>
#include <roscpp_tutorials/TwoInts.h>
#include "RadiationWatch.h"

ros::NodeHandle  nh;
using roscpp_tutorials::TwoInts;

RadiationWatch radiationWatch (2,3);
int radiation;
int error;
int rad;
int c;
float error_log;
int error_check(){
  bool i = true;
  radiationWatch.setup();
  while (i){
      radiationWatch.loop();
      radiation = (int) (radiationWatch.uSvh()*1000);
      error = (int) (radiationWatch.uSvhError()*1000);
      error_log = radiationWatch.uSvhError()*1000;
      char str[4];
      dtostrf(error_log, 6, 2, str);
      nh.loginfo(str);
      nh.spinOnce();
      if (error <= 100 && error > 0)
           {
            nh.spinOnce();
            rad = radiation;
            i = false;}
  }
  return rad;
  delay(5000);
}

void callback(const TwoInts::Request & req, TwoInts::Response & res){
   c = error_check();
   res.sum= c;
}

ros::ServiceServer<TwoInts::Request, TwoInts::Response> server("test_srv",&callback);

void setup()
{
  //Serial.begin(9600);
  radiationWatch.setup();
  nh.initNode();
  nh.getHardware()-> setBaud(57600);
  nh.advertiseService(server);
  //nh.advertise(chatter);
}

void loop(){
  nh.spinOnce();
  delay(5000);
}
```

Section 2: ROS Action

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import rospy
import actionlib
import time
from PiPocketGeiger import RadiationWatch
from radiation.msg import rad_testAction, rad_testFeedback, rad_testResult

class ActionServer:
  def __init__(self):
    #self._action_name = name
    self.server = actionlib.SimpleActionServer('rad_action_server', rad_testAction,
execute_cb=self.execute_cb, auto_start=False)
    self.server.start()

  def execute_cb(self, goal):
    i = True;
    r = rospy.Rate(5000)
    with RadiationWatch(24, 23) as radiationWatch:
        feedback = rad_testFeedback()
        result = rad_testResult()
        while i:
            success = False
            readings = radiationWatch.status()
            error = readings["uSvhError"]
            feedback.radiation = readings["uSvh"]
            feedback.error = readings["uSvhError"]
            self.server.publish_feedback(feedback);
            if (error<0.02 and error>0.0):
                result.radiation = readings["uSvh"]
                success = True
                i = False

        if success:
                self.server.set_succeeded(result)
if __name__ == '__main__':
  rospy.init_node('rad_server')
  server = ActionServer()
  rospy.spin()
```