# Design Document

# ARC - Autonomous RC

Tao Chen, Cierra Shawe, Daniel Stoyer

Version 1.0

December 2, 2016

## CONTENTS

# 1 Overview

This document will provide an overview of current design and milestones for the ARC system (ARCS).

## 1.1 Scope

This document is issued on December 2, 2016. At this time, system design is not final, due to the nature of a research project, and shall be modified as necessary.

## 1.2 Purpose

Each section of the document will provide an overview of the different systems contained within ARCS.

## 1.3 Intended Audience

The intended audience for this design document is the development team. As the design is not final, technical users are currently not taken into account.

# 2 System Interfaces - Cierra

The main computational unit (MCU), an Intel NUC SkullCanyon, is used as a starting point to test the feasibility of stereo-vision for a vision system as per FR-IA1. The NUC runs Ubuntu 14.04 to provide a graphical user interface for observing inputs such as the vision system and sensor input. Ubuntu 14.04 is also a standard used to run ROS and handle input from other nodes. All of the data processing will be done on the MCU.

The MCU interfaces in the following ways:

- Telemetry radio via a USB connection
- Raspberry Pi 2 via an ethernet connection
- Vision system via two USB ports.

19V of power is the manufacturer's rated usage during peak performance for the Intel NUC. Testing must be done to see if the NUC can run on 12V power, which is the RC battery standard, while running complex computations such as processing images into a point cloud.
The NUC should be able to process at least 15 frames a second without powering off while using 12V power. If the NUC is unable to run on 12V power, a new method for powering the NUC is needed, or a less powerful board, such as the UPBoard or a NVIDIA Jetson must be used. Using a less powerful board limits the speed in which vision is processed.

The Raspberry Pi interfaces with the PXFmini autopilot through a 40pin connection. Power is supplied via a micro USB port, or through the PXFmini, which has the option to be powered through a pass-through connection to main LiPo battery. The Raspberry Pi runs Debian Jessie ("cyan"), which is provided by Erle Robotics and is designed to process input from the PXFmini. The Raspberry Pi and PXFmini control motors and sensors, and collect sensor data.

Data is passed via a client-server model to the MCU using a TCP connection over an Ethernet connection. Packets are sent as JSON if the Raspberry Pi cannot be configured as a node within ROS.

The PXFmini provides sensor data and can directly interface through UART and I2C connections with a ublox Neo GPS with Compass unit and provides built-in support. The GPS unit can also interface through an IMU for more complex and accurate motion information. An IMU unit requires adapters to interface with the new Drone Foundation standard. Servos and a motor speed controller functions interface through the pulse width modulation pins on the PXFmini. The PXFmini abstracts the majority of motor and servo functions.

The system will be tested in incremental steps to ensure that all of the devices are talking to each other. The first step is to establish a connection between the MCU and the Raspberry Pi via TCP. Then a ping packet will be sent, with an expected response of pong.

With a working connection, the first goal is to control the direction of a motor from a command sent from ROS to the PXFmini. The motor should be able to spin one way when told to go forward, and reverse when a reverse command is sent. Then servo control will be tested.

After motor and servo control is established, sensor data will be collected and sent to the MCU. Two servos should be able to run synchronously to simulate steering. Data should be able to be displayed on to the user through the ROS interface. At this point, the system can be interfaced into the car with extreme caution. Once the servos and speed controller from the car has been connected to the PXFmini, all functions should be tested on minimum speed for the case of motors or the using least amount of travel for servos, to ensure all actions are as expected.

## 3 Vision System - Cierra

ARCS uses a stereoscopic vision system to aid in navigation and obstacle avoidance. Vision systems are the ”eyes” of the vehicle and are what allow the vehicle to operate autonomously. The vision system relates to requirement FR-IA1 and is a precursor to requirements FR-IA2 and FR-IA3. FR-IA1 is relevant because images must be captured and processed at a rate of 15 frames per second or higher to allow the vehicle to navigate quickly without colliding with objects. The vision system uses input from two USB cameras to create a depth map to detect objects at least 8” high to avoid collisions with objects that the vehicle is unable to drive over. OpenCV and ROS are used to process images from two different USB cameras, plugged into the MCU. ROS handles depth maps using a SLAM library in either a point cloud or occupancy grid map format, and the output is required for FR-NAV6.

The first step to testing whether the vision system works is ensuring input is received from both cameras simultaneously through visual observation on a display. Then after input is successfully received, the cameras must be calibrated using OpenCV. The calibration process involves using a checkerboard and taking images from various locations. The MCU plugged into a display, must then display a depth map based on the disparity between the left and right images. Once a depth map is created, frame rate must be compared to find the optimal resolution to process images successfully at 15 frames a second, with 60 frames being an ideal number. Frames per second are outputted onto a display for the user to see in order to ensure the frame-rate is above the target.

Currently, the optimal distance between cameras is not known, so testing needs to be done to see how far apart the

cameras must be placed to view objects between two and 20 feet away. The optimal range would be 6 inches to 100 feet of range. However, this goal is unlikely due to the restraints in image quality and the ability of the MCU to process the disparity maps in real-time. Using lower quality images provides fewer data points for OpenCV to compare, with a tendency towards unclear pixels on the edges of the images, unclear images when objects are very close to the cameras, and poor depth calculation when pixels are too far apart.

Other sensors, such as sonar or LiDAR, can be used alongside the stereo-vision cameras if the stereo cameras are unable to process objects within three feet of the vehicle. Other vision sensors are only used when it is determined stereo-vision alone is not enough to detect obstacles at close range.

Testing for the optimal distance for the two USB cameras is done by calibrating the cameras and holding a white piece of paper in front of the cameras from different distances, and then determining when the cameras can no longer detect the depth. After the camera distance have been calculated, the two cameras are mounted either via a 3d printed case or integrated near the car's headlights.

## 4 Sensors - Cierra

Sensors provide critical information that is used to provide feedback for where the vehicle is within a space. Sensors are a pre-requisite for all navigation criteria. FR-SN1, FR-SN2, FR-SN3 are the requirements for all sensor input. Sensors required for the vehicle are the following:

- Accelerometer to calculate acceleration and deceleration
- Gyroscope for orientation
- GPS with a compass for location
- Barometer for altitude
- Ultrasonic sensors for parallel parking and forward collision avoidance
- Encoders to measure wheel rotations (as needed)

The accelerometer and gyroscope are included on the PXFmini as a 9-axis sensor, or the data can be pulled from the external IMU. The GPS unit can be plugged directly into the PXFmini using one of the UART and I2C ports. Barometer data is not strictly required. However, the PXFmini has a built-in sensor, so it can be used as needed.

Sensor data is sent through the Raspberry Pi to the MCU, where it is processed by path planning and control algorithms. Obtaining sensor data is required before these algorithms are able to be implemented.

An I2C port will be multiplexed to support additional sensors if needed. Each sensor will be assigned a UID to be accessed through the multiplexed port.

Testing sensor output will be done by displaying values to the user to ensure the sensors are sending information to the MCU. The 9-axis sensor will be placed on a level sensor to ensure that it is properly calibrated.

## 5 HARDWARE - CIERRA

All hardware components will be tied down and encased with the minimal amount of port access required for the car to function as per FR-HW1. 3D printing and laser cut cases will be used, and venting will be used as necessary. This is to protect the hardware from rollovers as per FR-HW2.

Hardware layout will first be done on cardboard to ensure components are in the correct placement on the board. After the layout is determined, cases for objects will be modeled and printed, then secured onto an acrylic or polycarbonate laser-cut plate.

## 6 MOTION MODEL

Each iteration, the system issues a series of commands to the actuators, which are the motor and the servos in this project. Actuators then actuate the commands for the duration of one iteration. The time taken by each iteration may vary due to the different amount of information the system has to process. A control scheme/method must be needed to avoid the inconsistency in the actuators' behaviors caused by the inconsistent duration.

The speed control package from the AutoRally project is used to satisfy the speed consistency requirement. The system controls the speed by publishing a speed message to the listener, same as the steering.

### 6.1 Milestones:

1) Complete all simulations in ROS.
2) The motor can physically spin forward and backward according to instructions given by the system.
3) Servos can physically turn clockwise and counterclockwise according to instructions given by the system.
4) The speed of the vehicle is consistent with the speed issued by the system regardless of the road condition.

Motion model determines the correct commands to be issued by the system. A person has to adapt to the acceleration and steering of a particular car. The ARC system also has to adapt to the vehicle's configuration, such as weight, steering ratio, etc.

Besides, to follow a given path produced by the path planning algorithms, the system requires another control scheme, to ensure the vehicle strictly follow the path. When operating on a rough surface, drifting/slip off path happens. Converging back to the path also requires the same control scheme. The adaptation of the system to the vehicle is done by entering a set of parameters that include:

- Center of gravity.
- Weight.
- Turning radius to steering ratio.
- Maximum speed.
- Minimum turning radius
- Note: We need to keep it simple given that we have not even started any testing. These 5 parameters will satisfy the basic operation of the vehicle. The system will change the control strategies based on the parameters set.

### 6.2 Milestones:

1) Test the vehicle and record all necessary data.
2) Analyze the collected data by hand and simulate different configurations in ROS.
3) Derive equations from the results returned by simulations.
4) Polish the equations by doing more simulations in ROS.
5) Test the results on the actual vehicle with the minimum amount of hardware mounted on it.
6) Test the results on the actual vehicle with fully loaded hardware.

The PID (proportional–integral–derivative) control model is used to make sure that the vehicle is always on track. The PID model makes sure that, when the vehicle runs over bumps and drift off paths, it converges back to the desired path without overshoot or undershoot. When operating on a bumpy surface, the PID model is extremely critical to keep the vehicle on the paths.

### 6.3 Milestones:

1) Fully understand the PID control model.
2) Customize the model to adapt to our vehicle.
3) Simulate the model in ROS.
4) Test the model on a smooth surface and regularly changing the paths. The vehicle should converge to the new path every time.
5) Test the model on a rough surface. The vehicle should converge back to the planned path every time it drifts off paths.

The system has to know the location of the vehicle at any given moment. After each iteration when a series of actions are taken, the uncertainty of the vehicle location increases. Thus, the system has to utilize probabilistic methods with sensor measurements to minimize the uncertainty. The inputs to this modules are the previous location of the vehicle, actions taken in the previous iteration, and the sensors measurements, such as GPS, wheel encoders, vision system. With Bayes filter, the module produces reliable results.

### 6.4 Milestones:

1) Fully understand Bayes filter so that we can modify the library according to our need.
2) Being able to gather sensor measurements and feed them to the module.
3) Simulate the module on ROS.
4) Optimize the module so that the computation can be completed within a normal iteration.
5) Test the module on the actual vehicle.

# 7 PATH PLANNING

## 7.1 Global path planning:

When operating outdoor in an open area with a pre-downloaded satellite map, a global path planning module is required for the vehicle to travel from start to destination as desired. Dijkstra's based A* algorithm is used. The inputs to the module are the start and end locations (GPS coordinates), and a satellite map. The pre-processing unit generates valid waypoints, edges and edge costs by analyzing the map. Those waypoints and edges then get passed to the algorithm, which makes sure the shortest path is found. Since the implementation of Dijkstra's algorithm is not complicated, it is fully customized to accommodate the inputs/data types specified above. The heuristic for the algorithm is defined to be the straight-line distance from the waypoints to the destination, calculated using the GPS data.

## 7.2 Milestones:

1) Being able to pull data from the GPS unit.
2) Being able to generate waypoints on map images provided by OpenStreetMap.
3) Implement the algorithm
4) Test the algorithm
5) Test the algorithm with customized data structs.
6) Test the algorithm on the vehicle outdoor.

## 7.3 Local path planning:

When operating indoor within a closed area (with pre-defined map and boundaries), local path planning is required for the vehicle to travel from start to destination. Rapidly-exploring random tree is the algorithm used by the system. The inputs to the algorithm are start location (known), destination, the map with boundaries but without marked features/obstacles. The start location is either the very start location where the vehicle initializes and that is input by humans, or the subsequence locations estimated by the motion analysis. The vehicle also maps the space while it is traversing within it. It detects obstacles as it goes.

## 7.4 Milestones:

1) Customize the algorithm so that it does not generate samples within the vehicle size.
2) Test the algorithm by giving it a starting point, a destination point, and a map with marked features.
3) Integrate the algorithm into the system.
4) Simulate the algorithm in ROS.
5) Optimize the algorithm so that it plans out a path within the shortest amount of time. It needs to be short enough that human can not notice any delay when it needs to reconstruct a new path.
6) Test the algorithm on the vehicle.

# 8 Other Algorithms

## 8.1 Obstacle avoidance:

Obstacle avoidance is required for both local and global operations. Obstacle avoidance module grabs data packages from the vision system and tries to generate new paths on the go. Data included in the packages is defined by the vision system. There are two situations where the module may act differently. The first situation is when the vehicle has to come to a complete stop, due to high speed and path generation delay. The second situation is when the new path generation process is completed before the system decides to come to a complete stop.

## 8.2 Milestones:

1) Decide the data structs contained in the packages used to transfer data to the module.
2) Implement the module. There are existing packages that can be used. However, modifications must be required.
3) Integrate the path planning modules into the obstacle avoidance module.
4) Test the module in Simulation on ROS.
5) Optimize the algorithm.
6) Test the module on the vehicle.

## 8.3 Parallel Parking:

Parallel parking requires two special path planning algorithms. The first algorithm detects valid parking spots. This algorithm relies on images analysis results produced by the vision system. The second algorithm is a path planning algorithm specifically designed for parallel parking maneuvers. A similar approach to the Autonomous Parallel Parking RC Car project done at Cornell University is used. The software package provided by Cornell University is modified to use image information instead of readings from ultrasonic sensors.

## 8.4 Milestones:

1) Modified the software package so that in can be integrated into the system.
2) Test the algorithm by doing simulations on ROS.
3) Find out any optimizations that can be made to the algorithm. For example, minimize the number of turns it takes to complete the tasks.
4) Optimize the algorithm.
5) Set up an environment for testing.
6) Test the algorithms on the vehicle in the pre-set environment.

# 9 Image Analysis

Author: Dan Stoyer

The ROS image_transport (http://wiki.ros.org/image_transport) library should be used to pass images from the sensors into rtabmap_ros i. rtabmap_ros. The rtabmap_ros library handles depth-finding and environment mapping.

Milestone #1: Detect an object Use ROS image_transport (http://wiki.ros.org/image_transport) to pass images into rtabmap_ros (http://wiki.ros.org/rtabmap_ros). rtabmap_ros analyzes the images to detect objects. proj_min_cluster_size (int, default: 20) is used to control the size of object detected.

Milestone #2: Detect an object while moving After determining that rtabmap_ros is detecting an object while the vehicle is stationary, object detection should be tested while moving.

Milestone #4: Depth finding Once objects are detected, the distance to the object should be found by passing the images to rtabmap_ros's left/image_rect (sensor_msgs/Image) and right/image_rect (sensor_msgs/Image). The depth analyzed from the images should be tracked using subscribe_depth (bool, default: "false").

Milestone #5: Hand off analysis The object detection and depth information should be handed over to the path-finding algorithm.

## 10 User Interface

Author: Dan Stoyer

The user interface (UI) allows the user to issue commands and see the state of the vehicle. There are two UI platforms at play: the ground station and the companion computer. The companion computer UI is the most important to get working right away.

Milestone #1: CLI for companion computer. The robotics operating system (ROS) provides CLI for vehicle control. This interface is on the companion computer only.

Milestone #2: See vehicle status on remote terminal. The companion computer's UI should be transmitted over wifi through screen-share software. This approach allows immediate viewing of what is happening on the companion computer which allows time to work on more advanced remote UI solutions.

Milestone #3: Enter CLI commands on the ground station. The simplest place to start with remote commands is from the command line. The ground station will be running either Linux or Windows. In the case of Windows, a third-party terminal (such as MinGW or Cygwin) should be used.

Milestone #4: See vehicle status via ground station GUI. To see the vehicle status, a GUI is needed on the ground station. The ground station will run either a Linux distribution or Windows 7,8,10. ArduPilot runs on both and should be the first GUI tested. If ArduPilot does not Telemetry data should be sent with MAVLink protocol. Image data should be sent to rtabmap_ros using image_transport (http://wiki.ros.org/image_transport).

## 11 Radio Communication

Author: Dan Stoyer

Milestone #1: Send/Receive telemetry data The companion computer should send telemetry data by radio using the MAVLink protocol. The ground station should receive the telemetry data by radio using the MAVLink protocol.

Milesone #2: Send commands The ground station should send commands. Testing for seding commands should start with simple ”start” and ”stop” commands and gradually increase in complexity up to enter waypoints for autonomous navigation.

# 12 APPENDIX

## 12.1 Annex A Bibliography