SoftwarePilot Development Guide

Dec 2019

Version 1.0

By: Jayson Boubin

With assistance from: Dr. Christopher Stewart, Shiqi Zhang, Naveen Tumkur Ramesh Babu, Venkata Mandadapu and Zichen Zhang

Document Revisions

Date	Version Number	Document Changes
12/01/2019	0.1	Initial Draft
12/27/2019	1.0	First Release Version

Table of Contents

1 Introduction

1.1 Scope and Purpose

SoftwarePilot is an open source middleware and API that supports aerial applications. SoftwarePilot allows users to connect consumer DJI drones to programmable Java routines that provide access to the drones flight controller, camera, and navigation system as well as computer vision and deep learning software packages like OpenCV, DLIB, and Tensorflow.

SoftwarePilot is used by researchers to create and test aerial systems that use novel autonomy policies, architectural configurations, and vision algorithms with application domains ranging from agriculture to autonomous photography. Educators also use SoftwarePilot to teach middle school to university students about drones, autonomous systems, computational thinking, and programming in general

This guide covers all facets of the SoftwarePilot docker environment, explains and details drivers, routines, simulation of SoftwarePilot code, and debugging.

1.2 Terminology

Below is a list of important terminology that will be used throughout this document and associated definitions.

- Driver
  - A SoftwarePilot driver is one of two critical software components for executing FAAS missions. A driver is a Java file, compiled to a JAR file, that specifies a specialized microservice API that describes certain functions of autonomous flight. For instance, the FlyDroneDriver specifies functions for UAV flight. Drivers are detailed in the SoftwarePilot development guide.
- Routine
  - Routines are Java files, compiled to JAR files, that specify a series of driver calls and other logic that constitute a FAAS mission. For instance, the AutoSelfie routine flys a UAV to autonomously capture high-quality human facial images. Routines are detailed in the SoftwarePilot development guide.
- Kernel
  - The SoftwarePilot Linux Kernel is an instance of SoftwarePilot that executes outside of the SoftwarePilot virtual machine. The kernel is used for both simulating SoftwarePilot to test functionality, and executing components of SoftwarePilot in-mission that can not be run on the VM, like complex machine learning algorithms. The kernel and its functionality are detailed in the SoftwarePilot development guide

- $AUAVHOME
  - AUAVHOME is a SoftwarePilot docker container environment variable that is used to specify the root directory of the SoftwarePilot codebase, both for the purpose of reference in this guide, and to allow SoftwarePilot to specify absolute paths for all requisite operations in code.
- APK
  - The SoftwarePilot APK is the android application build to execute in the SoftwarePilot android-x86 VM. The APK includes all drivers and routines as JAR files, and can be installed using the SoftwarePilot compile script detailed in Section 3 of this guide.

2 The SoftwarePilot Docker Environment

SoftwarePilots principal development environment is packaged as a docker container. Installation, access, and management of this environment are detailed in the SoftwarePilot User Guide. In this section, we will describe the principal directories and files at $AUAVHOME in the SoftwarePilot docker environment and their various uses.

2.1 Files

2.1.1 compile.py

Compile.py is a python script used to compile all SoftwarePilot code and install it onto the virtual machine. This script is detailed in the SoftwarePilot User Guide.

2.1.2 cmple.pl

cmple.pl is a deprecated compile script that has been replaced by compile.py. This script performs many of the same functions as compile.py, but is not properly documented.

2.1.3 Californium.properties

Californium.properties is a properties file for our chosen implementation of CoAP, the protocol our drivers and routines use for communication. This file provides the ability to change many low-level network features of our CoAP implementation.

2.2 Source Directories

2.2.1 routines.src

routines.src holds all code associated with SoftwarePilot routines. This directory is quite deep in structure, and is detailed in full in Section 4 of this guide.

2.2.2 drivers.src

drivers.src holds all code associated with SoftwarePilot drivers. This directory, like the routines directory, is large, and is detailed in its own section, section 3, of this guide.

2.2.3 interfaces

interfaces holds all SoftwarePilot interface code. SoftwarePilot has interfaces for both routines ($AUAVHOME/interfaces/src/main/java/AuavRoutines.java) and drivers ($AUAVHOME/interfaces/src/main/java/AuavDrivers.java). These routines provide helper functions and required function implementations for both Drivers and Routines for SoftwarePilot.

2.2.4 Models

the Models directory provides implementations of prebuilt machine learning models that are useful for Drivers and Routines. Currently, the models directory holds python files used by the Vision Driver to recognize objects in UAV sensed images.

- $AUAVHOME/Models/FaceRecognition.py uses DLIB to find facial bounding boxes in UAV sensed images
- $AUAVHOME/Models/YoloRecognition uses YOLOv3 and the YOLOFace library to find facial bounding boxes in UAV sensed images
- $AUAVHOME/Models/DarknetRecognition uses YOLOv3 similarly to YoloRecognition but with GPU support

2.2.5 externalModels

externalModels contains a series of code directories useful for a number of SoftwarePilot techniques and processes:

- A_Star_Code and KNN_code: Both of these directories contain FAAS pathfinding code implemented for a series of scholarly research papers that can be useful for future tasks.
- The python directory contains a series of python scripts useful for extracting pathfinding features from UAV sensed images for human faces. the Corn directory, similarly, performs the same process for agricultural images of Corn.
- The Power directory contains the EnergyProfile, an application useful for modeling the total energy expended by FAAS routines based on a series of inputs: Aircraft energy Profile, Edge energy Profile, and Mission Trace. Some Aircraft and Edge traces are provided, but usually these must be constructed or modified for specific research purposes.

- Note: Many of these directories (A_Star_code, KNN_code, and python) will be deprecated in the near future when we release neural-network based pathfinding for SoftwarePilot.

2.2.6 AUAVAndroid

AUAVAndroid holds the code for the SoftwarePilot android app, build scripts, and libraries.

2.2.7 AUAVLinux

AUAVLinux, similar to AUAVAndroid, holds all code, build scripts, and libraries necessary to build the SoftwarePilot linux kernel.

2.3 Build Directories

When SoftwarePilot compiles code directories, the build outputs (JAR files, APKs, etc), are placed into specific build directories. From these build directories, they can be accessed by both Android, to build the AUAVAndroid app, and the SoftwarePilot kernel where they can be started as microservices for simulation or host communication.

2.3.1 libs

The libs directory is where all SoftwarePilot driver JAR files are held after they are compiled.

2.3.2 routines

The routines directory is where all SoftwarePilot routine JAR files are held after they are compiled.

2.3.3 kernels

The kernels directory holds both the AUAVLinux kernel JAR file, and the AUAVAndroid APK file. Kernels also holds the AUAVLinux shell script used to start the SoftwarePilot Linux kernel.

2.4 Documentation

2.4.1 docs

Docs holds all SoftwarePilot Javadoc for SoftwarePilot Routines, Drivers, and Interfaces. All documentation is in HTML format, and can be accessed from the index.html file in $AUAVHOME/docs

2.4.2 dji.docs:

The dji.docs directory holds all documentation for the DJI SDK, which is a core library for SoftwarePilot. Any DJI documentation required for SoftwarePilot can be accessed using the $AUAVHOME/dji.docs/index.html file.

2.5 Libraries and Storage

2.5.1 external

External holds all JAR files for external libraries that are not installed directly onto the SoftwarePilot Docker environment. At compilation, gradle compiles every java file against each of the JAR files in this directory. If you must add a library in the form of a JAR file to a SoftwarePilot routine, driver, or interface, the JAR file must be added here.

2.5.2 externalDLLs

externalDLLs holds all shared library objects for libraries that are distributed as shared libraries, and not JAR files or separate software. If you must add a library in the form of a shared object to a SoftwarePilot routine, driver, or interface, the .so file must be added here.

2.5.3 Datasets

the Datasets directory holds compiled datasets used by machine learning and pathfinding algorithms.

2.6 Tools

the SoftwarePIlot tools directory contains a series of potentially useful tools that are covered in the User Guide. Particularly, the coap-tool, and the Docker tools directory.

2.7 Temporary files

Temporary files exist in the $AUAVHOME/tmp directory. This directory contains temporary images captured by the UAV and downloaded, config files for temporary routines, temporary json and yaml metadata for transferred data, and more. Files in this directory are often replaced by software, but metadata and images from prior driver and routine executions can be viewed for debugging purposes on an inconsistent basis.

3 Drivers

The core component of the SoftwarePilot API is the SoftwarePilot driver. Drivers are java-based microservices that are exposed over CoAP to other drivers, routines, and any CoAP client on your network (like the CC app or SoftwarePilot CoAP Tool). Each driver is essentially a stand-alone server with its own dedicated port. Drivers await requests from routines, drivers, or CoAP clients. When a request is made, the appropriate driver executes code associated with the request. Example may include flying the UAV in a certain direction, capturing data, performing a machine learning task, etc.

3.1 Driver Basics

SoftwarePilot drivers are implemented as Java programs. They are CoAP servers that are hosted on the device execution SoftwarePilot code (either the Android VM running the SoftwarePilot Android app, the host machine running the SoftwarePilot Linux kernel, or both). SoftwarePilot drivers bind to a random port, and accept HTTP PUT requests over CoAP to perform certain implemented functions.

All drivers are located in the $DRIVER_SOURCE directory:

$DRIVER_SOURCE = $AUAVHOME/drivers.src/org/reroutlab/code/auav/drivers/

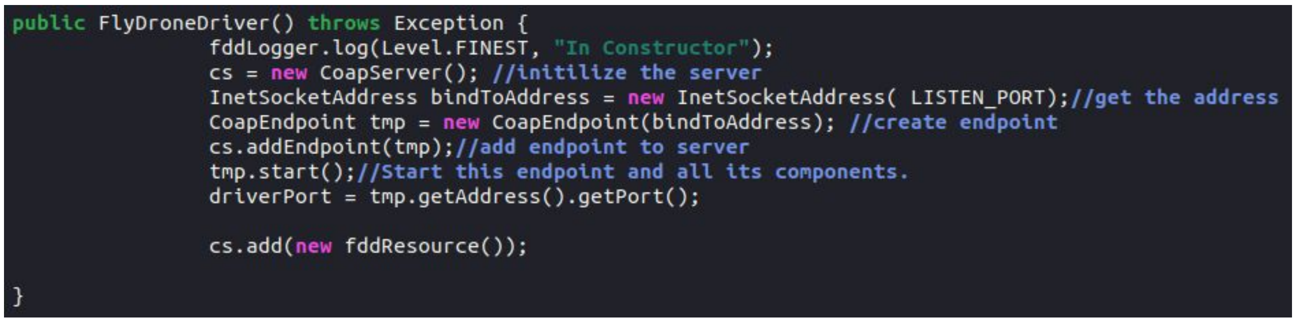
Here, you will find the source directories for all SoftwarePilot drivers. Each driver's source directory contains gradle build scripts and Java source code located in $DRIVER_SOURCE/$DRIVER_NAME/src/main/java/

3.2 Driver Implementation

Basic driver implementation is relatively simple. At their core, driver java files have subcomponents, the Constructor, and the Resource nested class

3.2.1 Driver Code

- The Driver Constructor:
  - Driver constructors are simple. The entire purpose of a SoftwarePilot driver constructor is to start a CoAP server that can handle HTTP PUT requests. Example code for this process can be seen below from the “FlyDroneDriver”

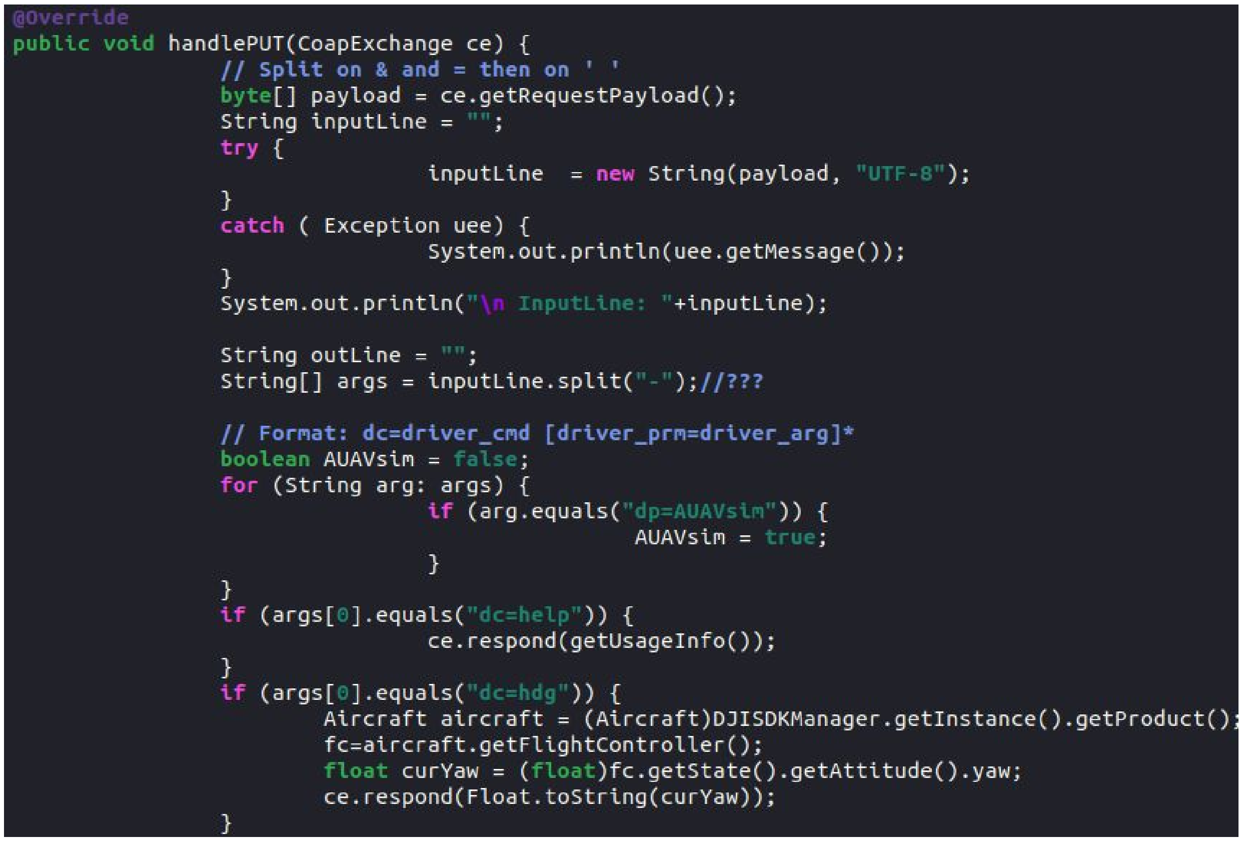


```
public FlyDroneDriver() throws Exception {
                fddLogger.log(Level.FINEST, "In Constructor");
                cs = new CoapServer(); //initilize the server
                InetSocketAddress bindToAddress = new InetSocketAddress( LISTEN_PORT);//get the address
                CoapEndpoint tmp = new CoapEndpoint(bindToAddress); //create endpoint
                cs.addEndpoint(tmp);//add endpoint to server
                tmp.start();//Start this endpoint and all its components.
                driverPort = tmp.getAddress().getPort();

                cs.add(new fddResource());

}
```

Above, the full construct for the FlyDroneDriver can be seen. The driver simply constructs a CoAP Server, binds a CoAP endpoint to a randomly generated port (LISTEN_PORT is set to 0), adds both the CoAP endpoint and fddResource (renamed variant of the Resource class for FlyDroneDriver) to the CoAP server, and starts the server.

- Resource:
  - The Resource nested class is implemented in every driver. It only implements one function, handlePUT, which accepts CoAP put requests.
  - handlePUT accepts a CoapExchange which can be used to accept PUT request Payloads. a PUT request payload, in this context, refers to the Driver name, command, and parameter formatted strings described in the User Guide. Driver names are used by the External Commands Driver (detailed in section 3.4 of this guide) to find which port the named driver is listening on. The External Commands Driver then sends an HTTP PUT request to the IP and Port of that driver which includes the driver command, and any driver parameters.

- ○ The handlePUT function then parses the arguments from the CoAP payload, and determines which driver command has been invoked, and then executes code relative to that command, or returns an error if the command was not found.
- ○ Below is an image showing the beginning of the handlePut function in the fddResource (specific Resource class variant) nested class of FlyDroneDriver. The handlePUT function operates as follows:
    - ■ Extract the binary PUT payload from the coapExchange
    - ■ Attempt to convert the payload to a UTF-8 string
    - ■ Print the string to system.out for logging purposes
    - ■ Split the string by the “-” character
    - ■ Check to see if any driver parameters are set to Simulation, “dp=AUAVsim” (described in section 5 of this guide).
    - ■ Begin checking the driver command to determine which driver function to execute.
        - ● In this example, we see both the “help” and “hdg” command code. More functions are available below.



```
@Override
public void handlePUT(CoapExchange ce) {
                // Split on & and = then on ' '
                byte[] payload = ce.getRequestPayload();
                String inputLine = "";
                try {
                                inputLine  = new String(payload, "UTF-8");
                }
                catch ( Exception uee) {
                                System.out.println(uee.getMessage());
                }
                System.out.println("\n InputLine: "+inputLine);

                String outLine = "";
                String[] args = inputLine.split("-");//???

                // Format: dc=driver_cmd [driver_prm=driver_arg]*
                boolean AUAVsim = false;
                for (String arg: args) {
                                if (arg.equals("dp=AUAVsim")) {
                                                AUAVsim = true;
                                }
                }
                if (args[0].equals("dc=help")) {
                                ce.respond(getUsageInfo());
                }
                if (args[0].equals("dc=hdg")) {
                        Aircraft aircraft = (Aircraft)DJISDKManager.getInstance().getProduct();
                        fc=aircraft.getFlightController();
                        float curYaw = (float)fc.getState().getAttitude().yaw;
                        ce.respond(Float.toString(curYaw));
                }
```

3.2.2 Adding commands to existing drivers

To add a command to an existing driver is incredibly simple. Adding a command only requires adding another if statement in the handlePUT function of the drivers Resource class to identify the driver command.

- Example: If you want to add a driver command “fly” to the FlyDroneDriver, simply add the following if statement to the bottom of the handlePUT function in FlyDroneDriver’s resource class:

    -
      ```
      if(args[0].equals(“dc=fly”)){

              //DRIVER CODE HERE

      }
      ```

3.2.3 Adding a new Driver

Adding a new driver requires a bit more effort than updating an existing driver with new commands. Luckily, we provide a driver template, $DRIVER_SOURCE/TemplateDriver, to help with the process. The steps to create a new driver (called $NEW_DRIVER) include:

- Create a new Driver directory at $DRIVER_SOURCE with the name $NEW_DRIVER by copying $DRIVER_SOURCE/TemplateDriver to $DRIVER_SOURCE/$NEW_DRIVER

  Note: Driver names must end with “Driver”. For instance, FlyDroneDriver is a valid driver name, whereas DriverToFlyDrone is not, so name accordingly.

- After creating your driver’s directory, go to $DRIVER_SOURCE/$NEW_DRIVER/src/main/java and change the name of the driver java file to $NEW_DRIVER.java. In this file, next change the class name to $NEW_DRIVER, and update all instances of that class name throughout the file:
    - The class name on line 59
    - The constructor name on line 127

    - The invocation when the Logger is initialized on line 64
- Once you have properly updated the Java file, add your Driver code.
- To compile the JAR file for $NEW_DRIVER, you must next link it to gradle:
    - In $DRIVER_SOURCE/$NEW_DRIVER/build.gradle:
        - add $NEW_DRIVER to the jar.baseName on line 6
        - Example:

```
jar.baseName='org.reroutlab.code.auav.drivers.$NEW_DRIVER'
```

    - In $AUAVHOME/drivers.src/settings.gradle:
        - Add an include line for your driver to the end of the file:
        - Example:

```
include "org:reroutlab:code:auav:drivers:$NEW_DRIVER"
```

Once all of these changes have been added, you can compile drivers using compile.py with either the -drv, -code, or -all flags. We recommend you use the -drv flag to assure that your driver is compiling without errors. If your driver has compiled correctly, it will return no errors, and its JAR file with the specified name will be available in the $AUAVHOME/libs/ directory

3.3 Invoking Drivers using CoAP

Drivers can be invoked with CoAP using the CC app, the CoAP tool provided in $AUAVHOME/toosl/coap-tool/

The process for using both of these features is covered in the SoftwarePilot User Guide. Information on testing and simulating Driver functions is available in Section 5 of this guide.6

3.4 Current Drivers:

In this section, we list all important drivers that currently exist in SoftwarePilot and their commands. Some SoftwarePilot drivers that have deprecated are still available in code, and some drivers have deprecated commands, but only supported drivers and commands are listed hperiodiallyere.

3.4.1 BatteryDriver

The SoftwarePIlot BatteryDriver is used to capture battery information from the DJI UAV for use in SoftwarePilot routines. It supports the following commands

- dc=cfg:

  This command configures the battery driver, starting a DJi callback that allows the battery driver to gather battery information periodically from the UAV. This command must be run before dc=dji to gain proper battery information..

- dc=dji:

  This command returns the battery percentage, voltage, MAH capacity, and current of the DJI UAV battery in real time.

3.4.2 CaptureImageV2Driver

CaptureImageV2Driver (updated from V1) uses the DJI drone's camera to capture images and store them on the SoftwarePilot VM.

- dc=ssm

  ssm stands for "Set Shoot Mode". In this context, SSM sets the shoot mode of the DJI camera to shoot pictures. Note: This is a necessary configuration step, and must be run before dc=get is invoked

- dc=get

  get captures a photo using the DJI drone camera. When get is executed, the UAV captures a photo and stores it to its internal SD card for later download.

- dc=dld

  dld will download a thumbnail style image from the DJI drone onto the SoftwarePilot Virtual Machine. This file will be of size 960x720 and will be located in /storage/emulated/0/AUAVtmp/pictmp.dat.

- dc=dldFull

  dldFull works similarly to dld, but downloads an image a the full resolution the UAV in question is capable of capturing (ex. the Spark can capture 4000x3000 images, which dldFull captures). This full image will be downloaded to /storage/emulated/0/AUAVTmp/fullPic.JPG

  Note: Helper functions to download images are available in RoutineTemplate, discussed in section 4 of this guide.

3.4.3 DroneGimbalDriver

DroneGimbalDriver adjusts the gimbal angle of any DJI UAV that has a Gimbal.

- dc=cal

  This command calibrates the gimal, setting the appropriate DJI callback to modify the gimal position. This function must be called before any other gimbal functions are called.

- dc=pos

  This command returns the UAV's gimbal pitch position, which is an integer value between 0 and 90 degrees.

- dc=res

  res sets the gimbal position of the UAV to a pitch angle of 0, essentially resetting the position to a forward heading.

- dc=dna-dp=angle

  dna sets the gimbal's pitch position down an absolute number of degrees, between 0 and 90. The driver parameter angle specifies the number of degrees.

- dc=upa-dp=angle

  una sets the gimbal's pitch position up an absolute number of degrees, between 0 and 90. The driver parameter angle specifies the number of degrees.

- dc=shk

  shk shakes the gimbal up and down twice. The purpose of this command is to provide visual acknowledgement from the UAV for some task completion. This can be useful for acknowledging a target recognition, a correct task completion, or simply testing connection between the UAV and edge system.

- dc=dwn-dp=angle

dwn moves the UAV's gimbal pitch angle down a relative amount. The driver parameter angle specifies the number of degrees.

- dc=ups-dp=angle

  ups moves the UAV's gimbal pitch angle up a relative amount. The driver parameter angle specifies the number of degrees.

- dc=lft-dp=angle

  dwn moves the UAV's gimbal yaw angle left a relative amount. The driver parameter angle specifies the number of degrees.

- dc=rgt-dp=angle

  dwn moves the UAV's gimbal pitch angle right a relative amount. The driver parameter angle specifies the number of degrees.

3.4.4 FlyDroneDriver

- dc=cfg

  cfg configuers the DJI sdk flight controller to allow for proper movement of the UAV in all environments. Note: before any flight actions are undertaken, the cfg command must be run.

- dc=hdg

  hdg can be used to get the aircraft's current yaw heading in world-coordinates.

- Takeoff and landing:

  Takeoff and landing are broken down into two commands, dc=lft and dc=lnd. lft will take the UAV off by flying it one meter off the ground and hovering. lnd will land the UAV from any height back onto the ground and turn the propellers off,

- Rotation:

  UAV rotation can be performed using the dc=rca-dp=angle and rc=rcc-dp=angle commands and parameters. rca will rotate the UAV clockwise along its yaw axis by a number of degrees specified by angle. rcc will rotate the UAV counterclockwise,

- Movement:

  There are a number of driver commands implemented for flying the UAV. A series of command exist to move the UAV small amounts (2 feet) in any direction. Those are:

  - dc=rgh, move the UAV right 2 feet
  - dc=lef, move the UAV left 2 feet
  - dc=fwd, move the UAV forward 2 feet
  - dc=bck, move the UAV backward 2 feet
  - dc=ups, move the UAV up 2 feet
  - dc=dwn, move the UAV down 2 feet

  You can also specify vector movement. A movement vector has 5 components, pitch angle, roll angle, yaw angle, throttle, and time. The UAV will move along a vector in each provided direction once per second for the specified number of seconds.

  - dc=vec-dp=pitch-dp=roll-dp=yaw-dp=throttle-dp=time

3.4.5 VisionDriver

The Vision Driver performs a number of machine learning tasks on images captured by the UAV. All tasks that the vision driver currently performs are too complicated to be run on the Android VM, and therefore must be run on in the SoftwarePilot docker environment. There are a number of helper functions that must be used to invoke these commands, which are detailed in section 4 and available in the template routine. Note: to execute any VisionDriver command, you must have an instance of the SoftwarePilot kernel running in the docker environment.

- dc=fce

  fce runs DLIB facial recognition on images sensed from the UAV and returns a facial recognition bounding box for each face in the image.

- dc=yolo

  yolo runs YoloFace facial recognition on images sensed from the UAV and returns a facial recognition bounding box for each face in the image.

YoloFace is the more updated and preferred version of the two facial recognition algorithms.

Note: We are currently implementing a general YOLO recognition system to be released soon.

- Facial Recognition Pathfinding

  Two functions exist for facial recognition optimization pathfinding using our feature extraction approach from our paper: Managing Edge Resources for Fully Autonomous Aerial Systems presented at ACM/IEEE SEC 2019. These algorithms will return movement directions from the current UAV position that will improve facial recognition images. dc=knn uses our knn pathfinding approach, dc=ast uses our A* pathfinding approach.

3.4.6 External Commands Driver

The External Commands Driver is a special driver that is never directly invoked by the user. It holds information that maps driver names to IPs and Ports so CoAP calls to drivers can be disseminated correctly. The External Commands driver always sits at the same port (5117) of any machine with an active kernel, and routes every CoAP request sent by either the coap tool or CC app to the appropriate driver. Routines also have a function that they can use to invoke drivers, which is detailed in Section 4, that directly calls external commands.

4 Routines

4.1 Routine Basics

SoftwarePilot routines can be thought of as applications, as compared to drivers which are components of the SoftwarePilot API. Routines are generally comprised of a series of commands which constitute an automated or fully autonomous aerial system mission. Routines, like Drivers, are JAR files which are built from Java source files.

All routines are located in the $ROUTINE_SOURCE directory:

ROUTINE_SOURcE = $AUAVHOME/routines.src/org/reroutlab/code/auav/routines/

Here, you will find the source directories for all SoftwarePilot routines. Each routine’s source directory contains gradle build scripts and java source code located in $ROUTINE_SOURCE/$ROUTINE_NAME/src/main/java

4.2 Routine Implementation

Routine implementation is quite similar to driver implementation. Routines, at their core, are java classes comprised of four methods:

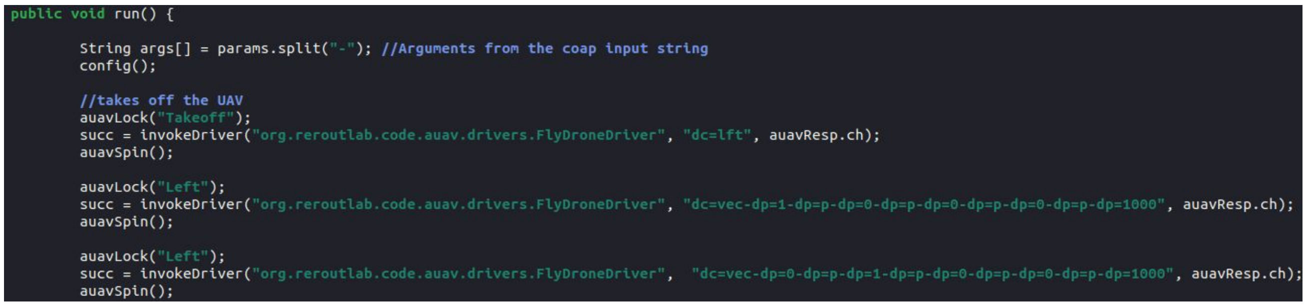
- The constructor
- startRoutine()
- stopRoutine()
- run()

Each of these functions is detailed in the following section.

4.2.1 Routine Code

- The Routine Constructor:
  - Routine constructors are much simpler than Driver constructors. Routines are threads, which implement a run() method that contains the routine’s core logic. Routine constructors simply construct a new thread which, if started, will execute the routines run() method.
- startRoutine() and stopRoutine():
  - These functions are used to start and stop the routines run() function. startRoutine() starts a new instance of the routine main thread.
  - stopRoutine() sets the forceStop public boolean to True. Routines should check this boolean value regularly to determine whether they should stop before finishing execution.

- run():
  - The run method of a routine is where most logic is implemented. Run functions usually contain a series of driver calls and other java logic which amount to a mission that flys the UAV, senses data, and manipulates that data to determine future UAV actions. Below is an example from BasicRoutine, our routine template .



```java
public void run() {

        String args[] = params.split("-"); //Arguments from the coap input string
        config();

        //takes off the UAV
        auavLock("Takeoff");
        succ = invokeDriver("org.reroutlab.code.auav.drivers.FlyDroneDriver", "dc=lft", auavResp.ch);
        auavSpin();

        auavLock("Left");
        succ = invokeDriver("org.reroutlab.code.auav.drivers.FlyDroneDriver", "dc=vec-dp=1-dp=p-dp=0-dp=p-dp=0-dp=p-dp=0-dp=p-dp=1000", auavResp.ch);
        auavSpin();

        auavLock("Left");
        succ = invokeDriver("org.reroutlab.code.auav.drivers.FlyDroneDriver",  "dc=vec-dp=0-dp=p-dp=1-dp=p-dp=0-dp=p-dp=0-dp=p-dp=1000", auavResp.ch);
        auavSpin();
```

  - The run function first parses arguments from the CoAP request, specified as "params". These parameters may be empty, but can be used to find parameters passed through the CoAP request that initiated the routine.
  - Next, the config() function is run. This is a helper function that runs configuration driver calls for both Flight and the UAV Camera, as well as turning "sim" mode off, which is discussed further in Section 5 of this guide.
  - Then, the routine begins to call Driver functions, as detailed in Section 4.3 of this guide.

4.2.2 Adding a New Routine

Constructing a new routine is similar in practice to adding a new Driver. The steps to create a new routine (called $NEW_ROUTINE) are as follows:

- Create a new Routine directory at $ROUTINE_SOURCE with the name $NEW_ROUTINE by copying $ROUTINE_SOURCE/BasicRoutine to $ROUTINE_SOURCE/$NEW_ROUTINE

  Note: unlike drivers, routines do not have name requirements. You can name a routine whatever you please.

- After creating your routine's directory, go to $ROUTINE_SOURCE/$NEW_ROUTINE/src/main/java and change the name of the routine java file to $NEW_ROUTINE.java. In this file, next change the

class name to $NEW_ROUTINE, and update all instances of the class name throughout the file:
    - The Class name on line 77
    - The constructor on line 296
    - Any printout strings with the routine name in them
- Once you have properly updated the java file, add your routine code.
- To compile the JAR file for $NEW_ROUTINE, you must next link it to gradle:
    - in $ROUTINE_SOURCE/$NEW_ROUTINE/build.gradle
        - add $NEW_ROUTINE to the jar.baseName on line 5
        - Example:

```
jar.basename= 'org.reroutlab.code.auav.routines.$NEW_ROUTINE'
```

    - In $AUAVHOME/routines.src/settings.gradle:
        - Add an include line for your routine at the end of the file:
        - Example:

```
include "org:reroutlab.code.auav.routines.$NEW_ROUTINE"
```

Once all of these changes have been added, you can compile routines using compile.py with either the -rtn, -code, or -all flags. We recommend that you use the -rtn flag to assure that your routine is compiling without errors. If your routine has compiled correctly, it will return no errors, and its JAR file with the specified name will be available in the $AUAVHOME/routines/ directory

4.3 Invoking Drives from Routines

SoftwarePilot routines make considerable use of driver calls to accomplish complicated UAV flight and machine learning tasks. To call a driver from a routine, you must use three functions:

- auavLock
- invokeDriver
- auavSpin

Each function is detailed below:

4.3.1 invokeDriver:

the invokeDriver function executes a CoAP call to a driver, and returns the CoAP response to a specified CoAP handler. Invoke driver is defined in the AuavRoutines.java interface file in $AUAVHOME/interfaces/src/main/java

The function invokeDriver takes the following arguments:

- String dn
  - The name of the Driver jar file
  - Ex: org.reroutlab.auav.drivers.FlyDroneDriver
- String params
  - The driver command and any parameters sent over CoAP
  - Ex: dc=lft
- CoapHandler ch
  - A CoAP handler that can be used to process the CoAP request such that the caller can access the CoAP response
  - Ex: the class AUAVHandler object auavResp in the AuavRoutines interface provides a CoAP handler to all routines that can be used: auavResp.ch

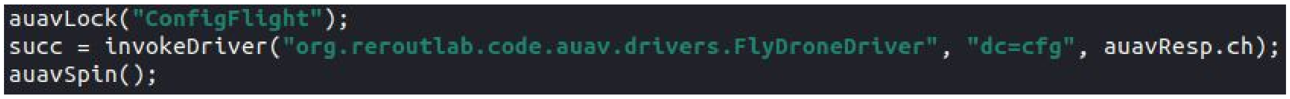
InvokeDriver returns a string denoting its success or failure.

4.3.2 auavLock and auavSpin

auavLock and auavSpin are functions implemented in the AuavRoutines.java interface that function as spinlocks to assure that the control flow of SoftwarePilot routines remains synchronous as driver calls execute. InvokeDriver is inherently asynchronous, meaning it does not wait for a response from the CoAP handler before returning a response to its calling routine. Without some sort of locking mechanism, routine control flow could return before a driver's action has been completed.

auavLock takes one string argument, meant to denote the reason for locking. auavLock should be set before the invokeDriver call, and set to a string that is not equal to "continue". After a driver call, auavSpin should be called to assure that the driver call has completed before control flow is continued.

4.3.3 Example Driver Call



```
auavLock("ConfigFlight");
succ = invokeDriver("org.reroutlab.code.auav.drivers.FlyDroneDriver", "dc=cfg", auavResp.ch);
auavSpin();
```

Above is a call to the FlyDroneDriver to configure UAV flight. First, the calling routine acquires the auavLock with the "configFlight" string. invokeDriver is then called to execute the dc=cfg driver call to the FlyDroneDriver using the CoAP handler auavResp.ch. invokeDriver will eventually return string succ to denote its success or failure. However, until the above invokeDriver call returns, auavSpin will continually sleep the main thread of the routine. Once invokeDriver returns, control flow can continue past auavSpin.

4.4 Starting and Stopping Routines

Starting and stopping routines using CoAP can be done using the CoAP client in the SoftwarePilot docker environment or CC app on the SoftwarePilot VM:

- Starting a Routine
    - Using the CC app, starting a routine can be performed by typing the following:
        - dn=rtn-dc=start-dp=$ROUTINE_NAME
        - You may add any other parameters as -dp's after $ROUTINE_NAME
    - Using the CoAP client in the docker environment, type the following command in $AUAVHOME/tools/coap-tool/
        - ./auav.pl IP rtn start $ROUTINE_NAME
        - IP is the IP address of your currently running kernel, on the VM, Docker Environment, or elsewhere
        - You may add any other parameters after $ROUTINE_NAME, separated by spaces
- Stopping a Routine
    - Stopping a routine is quite similar to starting a routine. Stopping a routine using either of the above methods requires simply replacing “start” with “stop” as the driver command for either command invocation method.
    - Note: Not all routines properly implement stopping procedures. It is recommended that you implement mechanisms as described in this section for stopping routines at safety-critical points, especially routines which fly the UAV.

4.5 Current Notable routines

Below is a list of current notable routines, their arguments, and their functionality:

- AutoSelfie
    - AutoSelfie is a routine referenced in our 2019 SEC paper. This routine flys up, finds a nearby human face, and attempts to take the best possible picture of that face, taking into account proximity, location, lighting, and more features.
    - Arguments: IP: the IP address of the host machine to access the vision driver.

- BasicVision
  - Arguments: IP: the IP address of the host machine to access the vision driver.
- FindPassword
  - This routine was used at BuckeyeCode 2019. It plays a game with the user, where the UAV flys to head height and allows the user to input a password. The password is binary, with a 1 denoted by a visible human face, and a 0 denoted by no visible human faces. If the password is correctly input, the UAV shakes its gimbal and lands.
- Detect
  - Detect is an early version of the AutoSelfie routine, where the UAV scans a room by rotating slowly around it, searching for human faces. Once it finds a human face, it returns the image to the VM and lands.
- BasicRoutine
  - Basic routine demonstrates simple UAV flight capabilities. BasicRoutine takes off, flys the UAV in a square, and lands. This routine is meant to be copied and transferred to new routines.

5 Simulation and Debugging

5.1 Overview

Development of SoftwarePilot routines and drivers is hard. There are lots of moving parts with any robotics, machine learning, or micro-service system, and we have combined all three. For this reason, we have provided SoftwarePilot simulation mechanisms and a few tips on debugging for SoftwarePilot users.

5.2 SoftwarePilot Simulation

Simulating SoftwarePilot routines and drivers can be a helpful mechanism for assuring that all components work properly before deployment onto the SoftwarePilot VM. Many SoftwarePilot drivers and routines require calls to libraries (like the DJI SDK) that can be tricky to deal with, are asynchronous, or both. Therefore, it can be helpful to assure that all other components of your driver or routine work before testing. To test SoftwarePilot code before deployment, you can use the SoftwarePilot Linux Kernel and Coap tool described in both this guide and the User Guide.

Simulating SoftwarePilot code on the Linux Kernel can cause problems if VM-only code is executed. For instance, any driver call that requires connection to the UAV will fail in the Docker environment without proper handling.

5.2.1 Preparing Code for Simulation:

Properly executing simulated code requires two steps:

- Executing code with the AUAVsim driver parameter to denote simulation
  - Ex: dn=rtn-dc=start-dp=BasicRoutine-dp=AUAVsim
- Preparing code for simulation by checking the simulation state in areas where driver calls or control flow could cause errors outside of the Virtual Machine.
  - Drivers and routines both have mechanisms to check the sim state (i.e whether AUAVsim was passed as a parameter)
    - Routines: the getSim() function will return AUAVsim if the driver is being executed in simulation, therefore, the statement: if(getSim().equals(“AUAVsim”) can be used to wrap code that should be run in simulation, and exclude code that should be run outside of simulation.

- Drivers: each driver has a boolean variable AUAVsim in its handlePUT method. This value is set to true at the beginning of the method if any driver parameter is set to AUAVsim. Therefore, users can check against the AUAVsim boolean value to provide code that is run in and out of simulation.

5.3 SoftwarePilot Debugging

Due to the number of moving parts SoftwarePilot contains, it can be difficult to debug applications. Debugging is required both at compile time (when JAR files and APKs are built) and at runtime (when routines and drivers are executing). Compile time and runtime debugging strategies are presented below

5.3.1 Compile Time

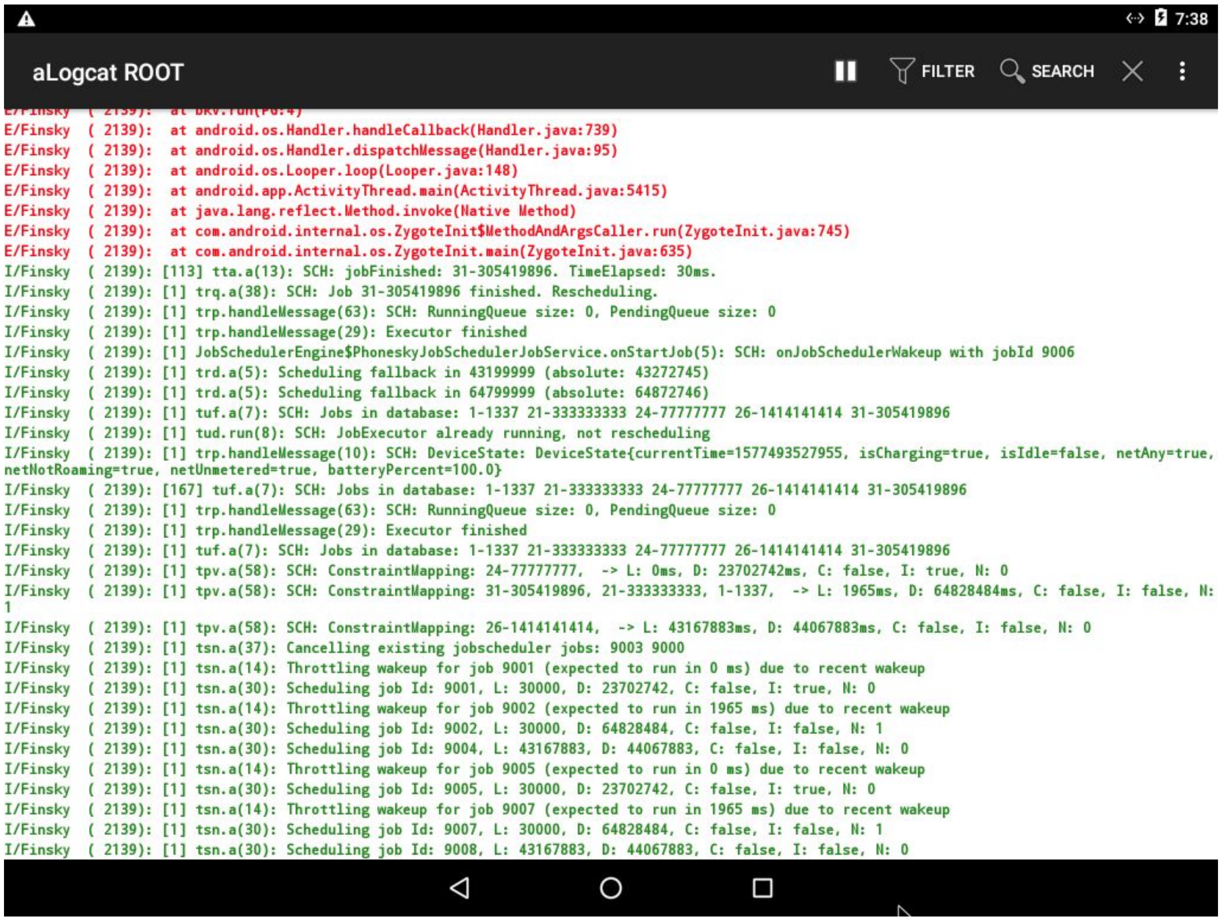
The $AUAVHOME/compile.py python script is our principal compile mechanism as described in the SoftwarePilot User Guide. When developing routines, drivers, or other additions to SoftwarePilot, we recommend that users use specific flags for each modified category before using general flags like -all or -code.

For instance, if you are building or modifying a routine, use the -rtn flag. Gradle outputs will show any Compilation errors in your new routine. A correctly compiled routine, driver, interface, or APK will generate a JAR or APK file in its appropriate build directory. If your code is compiling, but not generating a JAR or APK file, or is not compiling at all, make sure that you have properly modified the gradle settings outlined in this guide.

5.3.2 Runtime

Runtime errors can be quite difficult to find in SoftwarePilot applications, especially applications that are running and being tested on the VM. Testing all non-VM specific components of your routine, driver, or other code in simulation is recommended. For any code that is VM specific, we recommend you use Logcat to debug:

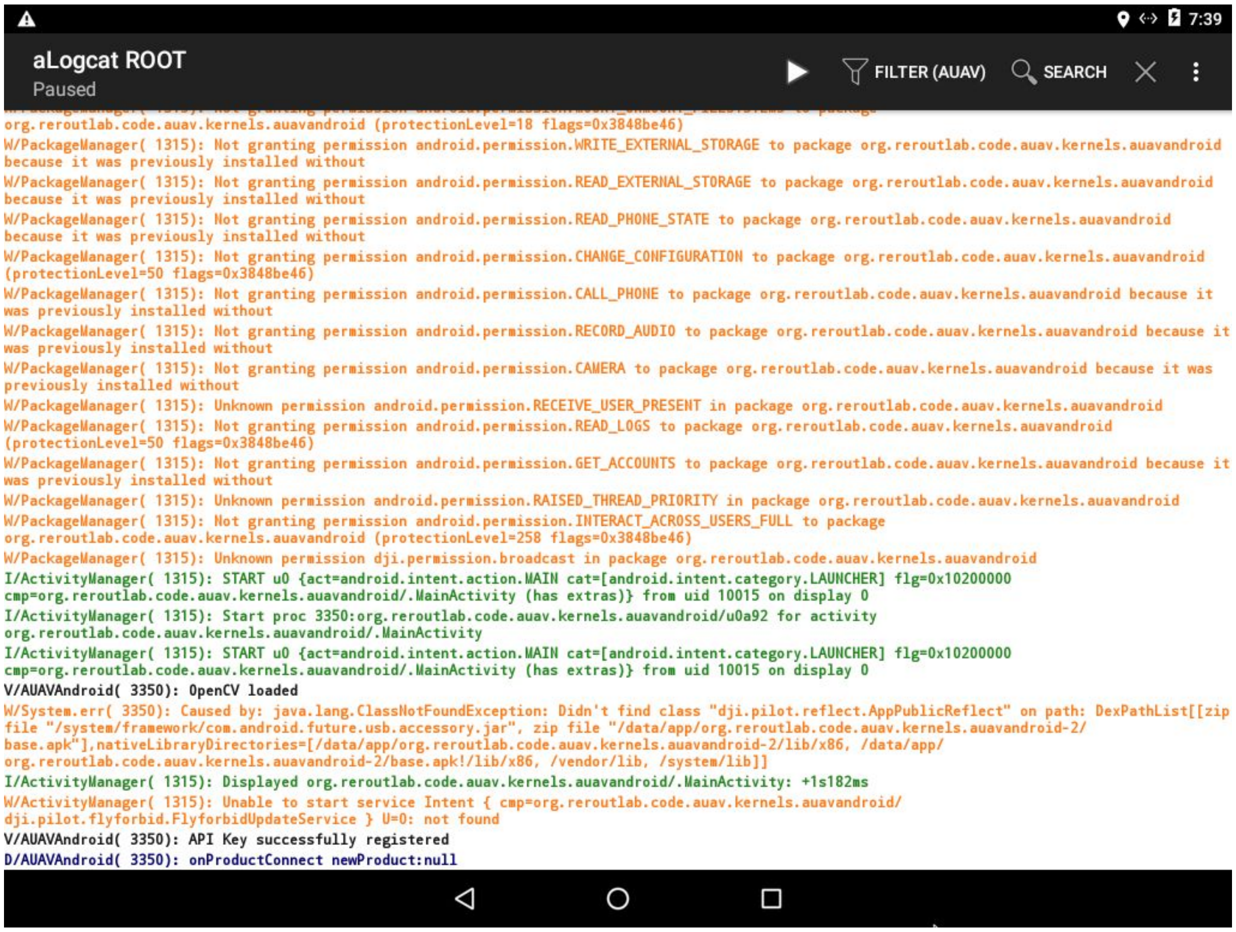
- Logcat is an Android app in the SoftwarePilot VM that can be used to access live Android system logs.



```
E/Finsky  ( 2139):  at bkv.run(PG:4)
E/Finsky  ( 2139):  at android.os.Handler.handleCallback(Handler.java:739)
E/Finsky  ( 2139):  at android.os.Handler.dispatchMessage(Handler.java:95)
E/Finsky  ( 2139):  at android.os.Looper.loop(Looper.java:148)
E/Finsky  ( 2139):  at android.app.ActivityThread.main(ActivityThread.java:5415)
E/Finsky  ( 2139):  at java.lang.reflect.Method.invoke(Native Method)
E/Finsky  ( 2139):  at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:745)
E/Finsky  ( 2139):  at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:635)
I/Finsky  ( 2139): [113] tta.a(13): SCH: jobFinished: 31-305419896. TimeElapsed: 30ms.
I/Finsky  ( 2139): [1] trq.a(38): SCH: Job 31-305419896 finished. Rescheduling.
I/Finsky  ( 2139): [1] trp.handleMessage(63): SCH: RunningQueue size: 0, PendingQueue size: 0
I/Finsky  ( 2139): [1] trp.handleMessage(29): Executor finished
I/Finsky  ( 2139): [1] JobSchedulerEngine$PhoneskyJobSchedulerJobService.onStartJob(5): SCH: onJobSchedulerWakeup with jobId 9006
I/Finsky  ( 2139): [1] trd.a(5): Scheduling fallback in 43199999 (absolute: 43272745)
I/Finsky  ( 2139): [1] trd.a(5): Scheduling fallback in 64799999 (absolute: 64872746)
I/Finsky  ( 2139): [1] tuf.a(7): SCH: Jobs in database: 1-1337 21-333333333 24-77777777 26-1414141414 31-305419896
I/Finsky  ( 2139): [1] tud.run(8): SCH: JobExecutor already running, not rescheduling
I/Finsky  ( 2139): [1] trp.handleMessage(10): SCH: DeviceState: DeviceState{currentTime=1577493527955, isCharging=true, isIdle=false, netAny=true,
netNotRoaming=true, netUnmetered=true, batteryPercent=100.0}
I/Finsky  ( 2139): [167] tuf.a(7): SCH: Jobs in database: 1-1337 21-333333333 24-77777777 26-1414141414 31-305419896
I/Finsky  ( 2139): [1] trp.handleMessage(63): SCH: RunningQueue size: 0, PendingQueue size: 0
I/Finsky  ( 2139): [1] trp.handleMessage(29): Executor finished
I/Finsky  ( 2139): [1] tuf.a(7): SCH: Jobs in database: 1-1337 21-333333333 24-77777777 26-1414141414 31-305419896
I/Finsky  ( 2139): [1] tpv.a(58): SCH: ConstraintMapping: 24-77777777,  -> L: 0ms, D: 23702742ms, C: false, I: true, N: 0
I/Finsky  ( 2139): [1] tpv.a(58): SCH: ConstraintMapping: 31-305419896, 21-333333333, 1-1337,  -> L: 1965ms, D: 64828484ms, C: false, I: false, N:
1
I/Finsky  ( 2139): [1] tpv.a(58): SCH: ConstraintMapping: 26-1414141414,  -> L: 43167883ms, D: 44067883ms, C: false, I: false, N: 0
I/Finsky  ( 2139): [1] tsn.a(37): Cancelling existing jobscheduler jobs: 9003 9000
I/Finsky  ( 2139): [1] tsn.a(14): Throttling wakeup for job 9001 (expected to run in 0 ms) due to recent wakeup
I/Finsky  ( 2139): [1] tsn.a(30): Scheduling job Id: 9001, L: 30000, D: 23702742, C: false, I: true, N: 0
I/Finsky  ( 2139): [1] tsn.a(14): Throttling wakeup for job 9002 (expected to run in 1965 ms) due to recent wakeup
I/Finsky  ( 2139): [1] tsn.a(30): Scheduling job Id: 9002, L: 30000, D: 64828484, C: false, I: false, N: 1
I/Finsky  ( 2139): [1] tsn.a(30): Scheduling job Id: 9004, L: 43167883, D: 44067883, C: false, I: false, N: 0
I/Finsky  ( 2139): [1] tsn.a(14): Throttling wakeup for job 9005 (expected to run in 0 ms) due to recent wakeup
I/Finsky  ( 2139): [1] tsn.a(30): Scheduling job Id: 9005, L: 30000, D: 23702742, C: false, I: true, N: 0
I/Finsky  ( 2139): [1] tsn.a(14): Throttling wakeup for job 9007 (expected to run in 1965 ms) due to recent wakeup
I/Finsky  ( 2139): [1] tsn.a(30): Scheduling job Id: 9007, L: 30000, D: 64828484, C: false, I: false, N: 1
I/Finsky  ( 2139): [1] tsn.a(30): Scheduling job Id: 9008, L: 43167883, D: 44067883, C: false, I: false, N: 0
```

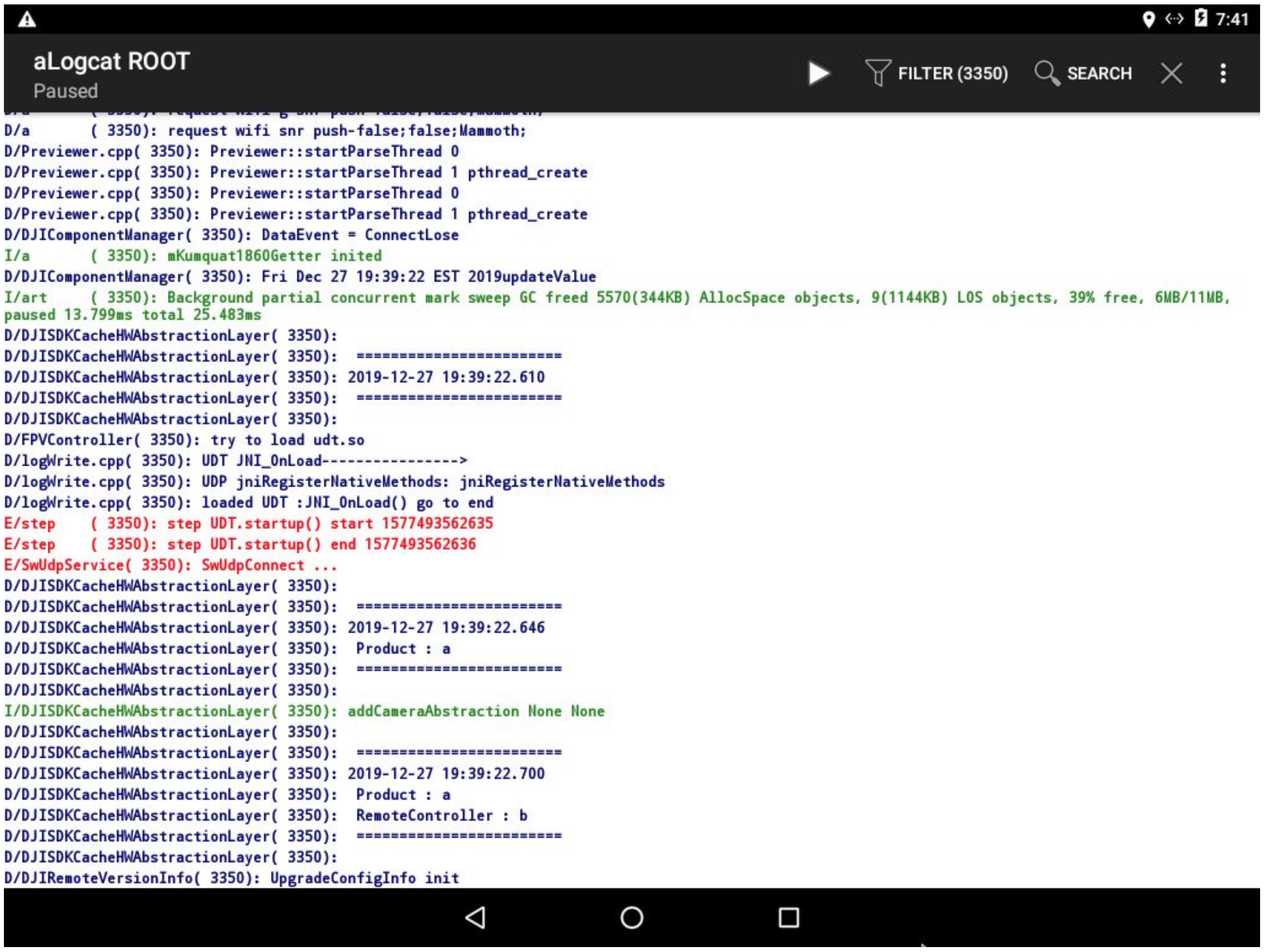
- Above is an example of the Logcat output window with no filter. Logcat streams android logs from all apps, so you should use the filter feature in the top right of the screen to search for SoftwarePilot logs specifically.
- One way to find all streaming logs for one application is to filter the process ID (PID) of that application, which is the 4 or5 digit number to the left of each log line.
- To find SoftwarePilot's PID, search for “AUAV” or “AUAVAndroid”, which is the internal name of the SoftwarePilot android app.

- Below is an example Logcat filter for “AUAV”. You can see that the AUAVAndroid app (visible to the far left of bottom-most lines) has a PID of 3350.



```
org.reroutlab.code.auav.kernels.auavandroid (protectionLevel=18 flags=0x3848be46)
W/PackageManager( 1315): Not granting permission android.permission.WRITE_EXTERNAL_STORAGE to package org.reroutlab.code.auav.kernels.auavandroid
because it was previously installed without
W/PackageManager( 1315): Not granting permission android.permission.READ_EXTERNAL_STORAGE to package org.reroutlab.code.auav.kernels.auavandroid
because it was previously installed without
W/PackageManager( 1315): Not granting permission android.permission.READ_PHONE_STATE to package org.reroutlab.code.auav.kernels.auavandroid
because it was previously installed without
W/PackageManager( 1315): Not granting permission android.permission.CHANGE_CONFIGURATION to package org.reroutlab.code.auav.kernels.auavandroid
(protectionLevel=50 flags=0x3848be46)
W/PackageManager( 1315): Not granting permission android.permission.CALL_PHONE to package org.reroutlab.code.auav.kernels.auavandroid because it
was previously installed without
W/PackageManager( 1315): Not granting permission android.permission.RECORD_AUDIO to package org.reroutlab.code.auav.kernels.auavandroid because it
was previously installed without
W/PackageManager( 1315): Not granting permission android.permission.CAMERA to package org.reroutlab.code.auav.kernels.auavandroid because it was
previously installed without
W/PackageManager( 1315): Unknown permission android.permission.RECEIVE_USER_PRESENT in package org.reroutlab.code.auav.kernels.auavandroid
W/PackageManager( 1315): Not granting permission android.permission.READ_LOGS to package org.reroutlab.code.auav.kernels.auavandroid
(protectionLevel=50 flags=0x3848be46)
W/PackageManager( 1315): Not granting permission android.permission.GET_ACCOUNTS to package org.reroutlab.code.auav.kernels.auavandroid because it
was previously installed without
W/PackageManager( 1315): Unknown permission android.permission.RAISED_THREAD_PRIORITY in package org.reroutlab.code.auav.kernels.auavandroid
W/PackageManager( 1315): Not granting permission android.permission.INTERACT_ACROSS_USERS_FULL to package
org.reroutlab.code.auav.kernels.auavandroid (protectionLevel=258 flags=0x3848be46)
W/PackageManager( 1315): Unknown permission dji.permission.broadcast in package org.reroutlab.code.auav.kernels.auavandroid
I/ActivityManager( 1315): START u0 {act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000
cmp=org.reroutlab.code.auav.kernels.auavandroid/.MainActivity (has extras)} from uid 10015 on display 0
I/ActivityManager( 1315): Start proc 3350:org.reroutlab.code.auav.kernels.auavandroid/u0a92 for activity
org.reroutlab.code.auav.kernels.auavandroid/.MainActivity
I/ActivityManager( 1315): START u0 {act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000
cmp=org.reroutlab.code.auav.kernels.auavandroid/.MainActivity (has extras)} from uid 10015 on display 0
V/AUAVAndroid( 3350): OpenCV loaded
W/System.err( 3350): Caused by: java.lang.ClassNotFoundException: Didn't find class "dji.pilot.reflect.AppPublicReflect" on path: DexPathList[[zip
file "/system/framework/com.android.future.usb.accessory.jar", zip file "/data/app/org.reroutlab.code.auav.kernels.auavandroid-2/
base.apk"],nativeLibraryDirectories=[/data/app/org.reroutlab.code.auav.kernels.auavandroid-2/lib/x86, /data/app/
org.reroutlab.code.auav.kernels.auavandroid-2/base.apk!/lib/x86, /vendor/lib, /system/lib]]
I/ActivityManager( 1315): Displayed org.reroutlab.code.auav.kernels.auavandroid/.MainActivity: +1s182ms
W/ActivityManager( 1315): Unable to start service Intent { cmp=org.reroutlab.code.auav.kernels.auavandroid/
dji.pilot.flyforbid.FlyforbidUpdateService } U=0: not found
V/AUAVAndroid( 3350): API Key successfully registered
D/AUAVAndroid( 3350): onProductConnect newProduct:null
```

- To find all log outputs from AUAVAndroid, we can filter by the PID, as shown below:



```
D/a       ( 3350): request wifi snr push-false;false;Mammoth;
D/Previewer.cpp( 3350): Previewer::startParseThread 0
D/Previewer.cpp( 3350): Previewer::startParseThread 1 pthread_create
D/Previewer.cpp( 3350): Previewer::startParseThread 0
D/Previewer.cpp( 3350): Previewer::startParseThread 1 pthread_create
D/DJIComponentManager( 3350): DataEvent = ConnectLose
I/a       ( 3350): mKumquat1860Getter inited
D/DJIComponentManager( 3350): Fri Dec 27 19:39:22 EST 2019updateValue
I/art     ( 3350): Background partial concurrent mark sweep GC freed 5570(344KB) AllocSpace objects, 9(1144KB) LOS objects, 39% free, 6MB/11MB,
paused 13.799ms total 25.483ms
D/DJISDKCacheHWAbstractionLayer( 3350):
D/DJISDKCacheHWAbstractionLayer( 3350):  ========================
D/DJISDKCacheHWAbstractionLayer( 3350): 2019-12-27 19:39:22.610
D/DJISDKCacheHWAbstractionLayer( 3350):  ========================
D/DJISDKCacheHWAbstractionLayer( 3350):
D/FPVController( 3350): try to load udt.so
D/logWrite.cpp( 3350): UDT JNI_OnLoad---------------->
D/logWrite.cpp( 3350): UDP jniRegisterNativeMethods: jniRegisterNativeMethods
D/logWrite.cpp( 3350): loaded UDT :JNI_OnLoad() go to end
E/step    ( 3350): step UDT.startup() start 1577493562635
E/step    ( 3350): step UDT.startup() end 1577493562636
E/SwUdpService( 3350): SwUdpConnect ...
D/DJISDKCacheHWAbstractionLayer( 3350):
D/DJISDKCacheHWAbstractionLayer( 3350):  ========================
D/DJISDKCacheHWAbstractionLayer( 3350): 2019-12-27 19:39:22.646
D/DJISDKCacheHWAbstractionLayer( 3350):  Product : a
D/DJISDKCacheHWAbstractionLayer( 3350):  ========================
D/DJISDKCacheHWAbstractionLayer( 3350):
I/DJISDKCacheHWAbstractionLayer( 3350): addCameraAbstraction None None
D/DJISDKCacheHWAbstractionLayer( 3350):
D/DJISDKCacheHWAbstractionLayer( 3350):  ========================
D/DJISDKCacheHWAbstractionLayer( 3350): 2019-12-27 19:39:22.700
D/DJISDKCacheHWAbstractionLayer( 3350):  Product : a
D/DJISDKCacheHWAbstractionLayer( 3350):  RemoteController : b
D/DJISDKCacheHWAbstractionLayer( 3350):  ========================
D/DJISDKCacheHWAbstractionLayer( 3350):
D/DJIRemoteVersionInfo( 3350): UpgradeConfigInfo init
```

- Above, you can see outputs from the DJI SDK, and other components of SoftwarePilot. Scrolling through this output stream will allow you to find any error messages that SoftwarePilot has generated from Android, Java, or otherwise, and will allow you to debug your code properly.
- It can also be useful to filter by "system.out", which will display any print line statements you have inserted for logging or denotation purposes into your code. We do not recommend you rely on print line debugging, but print line statements can be useful for denoting points in driver and routine control flow, and can be helpful when scrolling through log streams.