Parrot®

For Developers

Developing applications for the SkyController

Parrot SA

February 4, 2016

Contents

Purpose of this document

This document describes the differences between the Parrot SkyController and other Android devices.

It is designed for Android developers which want to make apps that run directly on the Parrot SkyController, it does not cover the topic of using the ARSDK (see the Android samples for this).

1 Environment setup

1.1 Requirements

In order to develop for the Parrot SkyController, you need a working Android development environment, able to deploy application on Android 4.2.2 targets.

A good understanding of adb shell commands can also be useful, as a lot of the SkyController configuration is done by stand-alone programs and shell scripts, and not by a Java API.

1.2 Configuring ADB

The SkyController has a micro-usb port, which is used both for device flashing (*not covered in this document*) and for adb access. When starting the SkyController, you **must** unplug the micro-usb cable, else it will start in flashing mode and won't be usable for development.

The SkyController USB vendor ID is `0x0525`, and must be added to your `adb_usb.ini` file. On Linux (and other Unix-likes, like Mac OSx), this file is located in the `~/.android/` directory.

After adding the USB vendor ID to the `adb_usb.ini` file, you should restart your adb server (`adb kill-server; adb start-server`). At this point, you should be able to see the SkyController in the list of adb devices.

1.3 Adding peripherals

By default, the SkyController supports a wide range of HDMI screens and USB peripherals. For ease of development, using an HDMI screen and a USB mouse is advised. Some touch screens with USB interface might also work.

Of course you can also work with the built-in controls and no display, but it might be more difficult for you to start developing in this setup.

1.4 Stopping FreeFlight on the SkyController

By default, the SkyController launches FreeFlight at start-up.

In order to stop FreeFlight, the easiest solution is to use an HDMI screen, and to leave the FreeFlight app by using the Back button and clicking OK on the confirmation dialog. By doing this, FreeFlight will a) reset the gamepad mode to the default one (more on that later) and b) stop the system daemon from restarting it. If FreeFlight is force-killed (or crashes), a system daemon will restart it automatically.

You can also completely uninstall FreeFlight from the SkyController, but you will need to force-update the SkyController to reinstall the app (*i.e. you can't install an APK from the Play Store, they're not exactly the same application !*)

1.5 Ready to develop

If you can see your SkyController on adb, and you have properly closed FreeFlight, you're ready to start!

2 Advanced setup

This section presents some advanced thing you can do with the SkyController system. ***These manipulations should only be used if you understand what you're doing, as errors can transform your SkyController into an expensive paperweight!***

2.1 Root access

The SkyController Android version is based on Android 4.2.2. It is a *userdebug* build, meaning that root acces is available. In the same way, it means that the `/system` partition can be remounted as read-write.

Warning: abusing root acces can totally brick the device, do not modify any file in /system unless you EXACTLY understand what you're doing!!!

2.2 Uninstalling FreeFlight

FreeFlight is installed as a system application, and thus can not be deleted without root access.

With root access and the `/system` partition remonted as read-write, you can close the FreeFlight app, and uninstall it by deleting the `/system/app/ARFreeFlight.apk` file. After doing this, you should reboot the SkyController.

2.3 Forced update

All the `/system` partition is rewritten during an update, so updating the SkyController will reinstall FreeFlight if you uninstalled it previously. *Note that the data partition is left untouched during update process, so your app won't be deleted during an update.*

Should you want to reinstall FreeFlight, you can force the SkyController to update from a USB drive, with the version currently installed (downgrading is not supported and can cause issues during boot !). To do this, connect to your SkyController in a root adb shell, then type the command `pinst_trigger`. The following lines should be printed as a response:

```
shell@android:/ # pinst_trigger
```

```
sysfs_read_int on ubi1:-1 mtd_num=2
ubi_attach on mtd1 :  3
ubi_get_vol_name on ubi3:1 alt_boot
ubi_rename_boot on ubi3 updater -> 0
ubi_detach on ubi3 :  0
```

If any other text is printed, it means that the command did not work (ar you sure you launched it in a root shell ?).

After running this, reboot the SkyController with a USB drive containing the update file (`nap_update.plf`) plugged into the USB port. Note that the SkyController will keep booting in update mode until it successfully find an update file on the USB drive, so it won't be usable between the `pinst_trigger` call and the actual update.

3 Fully integrated features

Some of the SkyController features are fully integrated into the Android API. Here is a list of such features.

3.1 Wifi

The *long range* wifi (`wlan0`) is managed through the Android WifiManager interface (as you would manage the WiFi on a phone or tablet), except for the country selection.

Country is persistent across reboots and updates, so you should almost never have to change it. *Note that some SkyController are locked to a given country due to regulations.*

3.2 Sensors

The SkyController GPS sensor is fully integrated into the Android APIs. The same goes for the Accelerometer, Gyroscope and Magnetometer sensors. The Orientation sensor is used as the sensor fusion result and should be used if your app requires the attitude or the heading of the SkyController.

Note : The SkyController magnetometer calibration status is avaible through the `accuracy` *field of the Magnetometer and Orientation SensorEvents. The precise measurement of the calibration use a custom protocol that is likely to change between versions and is not discussed here.*

3.3 Gamepad

The SkyController joysticks and buttons are seen as a USB gamepad, sending MotionEvents and KeyEvents to the applications. You can use it as any other USB gamepad on Android.

One of the gamepad axis will actually be an image of the battery voltage, and is detailed in the next section.

4 Non-fully integrated features

This section presents some of the features that exists on Android phones and on the SkyController, but for which the SkyController API differs from the Android API.

4.1 Battery level

The battery level is *not* integrated into the Android system, and thus can not be requested through the standard API. It is transmitted to the application as the last axis of the gamepad.

To convert from the axis value to the battery voltage, you can use the following equation:

```
float voltage = (float) (((battery + 1) * 1.25) * 1.6) + 9;
```

Where `battery` is the axis value. To further convert the voltage into a percentage, we use the following equation:

```
int percent = 100 - (int)(-14.52759414 * Math.pow(voltage, 3)
 + 543.15310727 * Math.pow(voltage, 2)
 - 6801.85740495 * voltage
 + 28532.85560624);
```

Which is then saturated between 0 and 100.

5 SkyController specific features

This section describes some features specific to the SkyController, and completely inexistant in other Android devices. Almost all of these features are using binaries and scripts to work, and are not wrapped into Java APIs.

5.1 LEDs control

The SkyController has 16 leds, divided into 4 visual banks:

- 1 red led for the record indicator. Id: 0
- 1 red and 4 while leds for the wifi indicator. The red led shares the leftmost position with a white led. Ids: 1 (red), 2,3,4,5 (white)
- 1 red and 4 while leds for the local battery indicator. The red led shares the leftmost position with a white led. Ids: 6 (red), 7,8,9,10 (white)
- 1 red and 4 while leds for the drone battery indicator. The red led shares the leftmost position with a white led. Ids: 11 (red), 12,13,14,15 (white)

A command to the LED driver will update the 16 leds at once, and it is not possible to read the current LED state from the driver, so your application will have to save the requested state in order to allow toggles of individual leds. If blinking is required, it should be implemented in your application (i.e. toggle a led every N sec).

The system does not drive any led, all indicators should be updated from within your application (as FreeFlight does).

5.1.1 LED protocol

The LED driver is connected through an SPI interface, driven by GPIOs. The `/sys/class/gpio/gpio<xx>/value` files are writeable by everyone, so you don't need root access to use the leds.

Here is a table of the 4 GPIOs numbers and names:

GPIO Nr	Usage
55	SPI Clock
56	Latch signal
57	Blank signal
58	SPI MOSI

Here is a pseudocode representing a send to the driver:

```
set_leds (boolean leds[16]) {
    for (i=0; i<16; i++) {
        SPI_MOSI = leds[i];
        SPI_CLOCK = 1;
        SPI_CLOCK = 0;
    }
    BLANK = 1;
    LATCH = 1;
    LATCH = 0;
    BLANK = 0;
}
```

The `/system/xbin/setleds` script gives an example of how to use the driver. This script will light the leds whose IDs are given as command line arguments.

5.2 Gamepad modes

The SkyController gamepad can be configured into two modes: a compatibility mode, which use the top-left joystick as a mouse, and a full-gamepad mode which uses the top-left joystick as the DPAD controls.

5.2.1 Compatibility mode

This is the default mode when booting, or after exiting FreeFlight. In this mode, the top-left joystick is used a a mouse, for compatibility with apps not designed for DPAD navigation.

This mode uses the left joystick as the DPAD, with the TakeOff button having the DPAD_CENTER fallback function. The back and home (settings, on black SkyControllers) are mapped to either L2/R2 (with Back-/Home as fallbacks) when an HDMI screen is plugged, or L1/R1 (without fallbacks) when no HDMI screen is plugged. This is a protection to avoid exiting an application without a screen. The return-to-home button has a fallback to the APP_SWITCH feature.

Fallback functions are triggered only if the running application did not handle (i.e. return `true` in the `dispatchKeyEvent()` function) the primary event, so by handling all primary event, you can prevent fallbacks features to be triggered.

5.2.2 Full gamepad mode

This mode is used by the FreeFlight app, and can be used by any application which supports a full DPAD navigation. In this mode, the top-left joystick is used as a DPAD, and the joystick click has the DPAD_CENTER fallback function.

Contrary to the compatibility mode, this mode home (settings, on black SkyControllers) and back buttons behavior does not change when a screen is plugged or not : the back button is always mapped to L2 (with BACK fallback), and the home/settings is mapped to R1 (without fallback). Other mappings (and axis numbering) will differ between the two modes.

5.2.3 Changing the mode

Changing the mode is done through the `atmegCheckUpdate` script, which is in the SkyController PATH. This script has many uses which are not covered by this document, but the mode switch command works in the following way:

- `atmegCheckUpdate mode FREEFLIGHT` : put the gamepad in full gamepad mode (the mode used by FreeFlight)

- `atmegCheckUpdate mode HOME_SCREEN` : put the gamepad in compatibility mode, as if a screen is currently plugged
- `atmegCheckUpdate mode HOME_NOSCREEN` : put the gamepad in compatibility mode, as if a no screen is present

Note: When in `HOME_XXX` mode, the mode will automatically whange its screen settings on HDMI hotplug/unplug events

5.3 Screen detection

The presence of an HDMI screen is indicated by the `hw.external_screen` system property. A value of "1" means that an external screen is plugged, any other value means that no screen is plugged.

5.4 SkyController configuration

The SkyController system configuration is done through a system similar to the Android property system. Accessing this configuration is done through the `SCConfig` binary. Main commands are:

- `SCConfig dump` : Dumps the current config in stdout.
- `SCConfig set KEY VALUE` : Sets the config KEY to the value VALUE
- `SCConfig remove KEY` : Remove the config KEY, if present
- `SCConfig get KEY [DEFAULT]` : Get the value associated with KEY. If KEY is not present, then DEFAULT will be returned (or an empty string if DEFAULT is not provided)

Note: Using a bad configuration might disable the wifi on the SkyController. Before changing the configuration, you should make a copy of the previous configuration, and switch back to this copy if anything fails

Note: The configuration is stored in the `/data/skycontroller/config` file. You should NOT modify this file directly. The `SCConfig` tool handles concurrent access and file locking.

5.4.1 Common configuration keys

Here is a list of the common configuration keys:

`KEEP_FF_RUNNING`

This config key should always be "0". A different value means that a system daemon will try to restart FreeFlight when it's not running. FreeFlight sets this value to "1" when starting, and resets it to "0" when exiting properly.

`WIFI_BAND`

The requested band of the access point. "0" means 2.4GHz, "1" means 5GHz, other values have undefined behavior.

Note: This key is only applied when the `WIFI_REQUEST_APPLY_SETTINGS` is set to "1"

`WIFI_CHANNEL`

The requested channel of the access point, within the selected band. If an invalid band/channel combination is given, the system will select a default one.

Note: This key is only applied when the `WIFI_REQUEST_APPLY_SETTINGS` is set to "1"

`WIFI_COUNTRY`

The requested country for all wifi (both access point and long range). This setting changes both the Tx powers and the valid channels/bands. Value is a 2 character country code (e.g. "*US*", "*FR*", ...). Putting an unknown country code will set the SkyController wifi to an universal mode (low power, few channels available).

Note: This key is only applied when the `WIFI_REQUEST_APPLY_SETTINGS` is set to "1"

Note: Changing the country will lead to a long range wifi disconnection as the wireless driver needs to be reloaded in order to change the regulatory domain.

Note: Some SkyController are locked to a specific country, and chaning this key won't have any effect on these devices. The `WIFI_COUNTRY_APPLIED` will always reflect the actual applied country.

`WIFI_SSID`

The SSID of the SkyController's AP. The value will be filtered (stripping invalid characters, shortening if necessary) by the system before being applied, but values should be strings of less than 32 charactes. It should be matched by the following regexp: `"^[-_a-zA-Z0-9]{1,32}\$"`.

Note: This key is only applied when the `WIFI_REQUEST_APPLY_SETTINGS` is set to "1"

`WIFI_REQUEST_UPDATE_SETTINGS`

Set this key to "1" to apply all pending `WIFI_XXX` configuration. Almost all configuration changes will triger an access point stop/start, thus disconnecting any connected device.

This key will be set back to "0" by the system after application.

`WIFI_XXX_APPLIED`

These keys shows the applied wifi settings.

- `WIFI_APPLIED_CTL` : The power domain applied for the current wifi country.
- `WIFI_APPLIED_REG_DOMAIN` : The regulatory domain applied for the current wifi country.
- `WIFI_COUNTRY_APPLIED` : The actual wifi country.
- `WIFI_SSID_APPLIED` : The actual access point SSID.
- `WIFI_BAND_APPLIED` : The actual access point band.
- `WIFI_CHANNEL_APPLIED` : The actual access point channel.

6 Handling ARSDK connections

6.1 Connecting to a drone

Connection to a drone is done in the same was as on any other Android device. Any app working on phones/tablet should work on the SkyController, as long as the UI is adapted to the DPAD navigation and the screen size.

6.2 Connecting to a tablet/smartphone

The actual code to handle incoming connections is part of the FreeFlight application embedded on SkyControllers. The code is not open-sourced, but it is implemented using components avaible in the ARSDK.

6.2.1 Publishing a mDNS service

To be discoverable on the network, the application should publish a mDNS service with the proper type (`_arsdk-0903._udp.local`).

6.2.2 Accepting incoming ARSDK connections

To accept a connection, the application should have an instance of an `ARDISCOVERY_Connection` object, running the `ARDISCOVERY_Connection_DeviceListeningLoop()` method. This discover server will then negociate connection as would a drone do. A complete rundown of the `ARDiscovery` protocol is available in the ARSDK protocols documentation.

6.2.3 Creating the ARNetwork instances

The ARNetwork instance can be created with the parameters from the Discovery phase. The instance is then used in the same way as a client instance (the ARNetwork protocol is peer to peer).

6.2.4 Routing commands

The minimum routing rules for the commands are the following:

- All `ARCommands` received from the drone must be sent to the tablet (if present)
- All `ARCommands` received from the tablet must be sent to the drone (if present), except for the SkyController.X.Y commands (see `ARCOMMANDS_Filter`)

Note: In order to grab the `ARCommands`, you can't use the `ARController` API, as it hides this part of the protocol.

6.2.5 Routing video

Routing an `ARStream1` stream is done in the same way as routing the commands: All stream packets received from the drone are sent to the tablet (at the `ARNetwork` level).

Routing an `ARStream2` stream is done by using the `libARStream2` library, and specifically the `resender`-related functions. You can find informations about this API in the following file:
`<SDK>/packages/libARStream2/Includes/libARStream2/arstream2_stream_receiver.h`