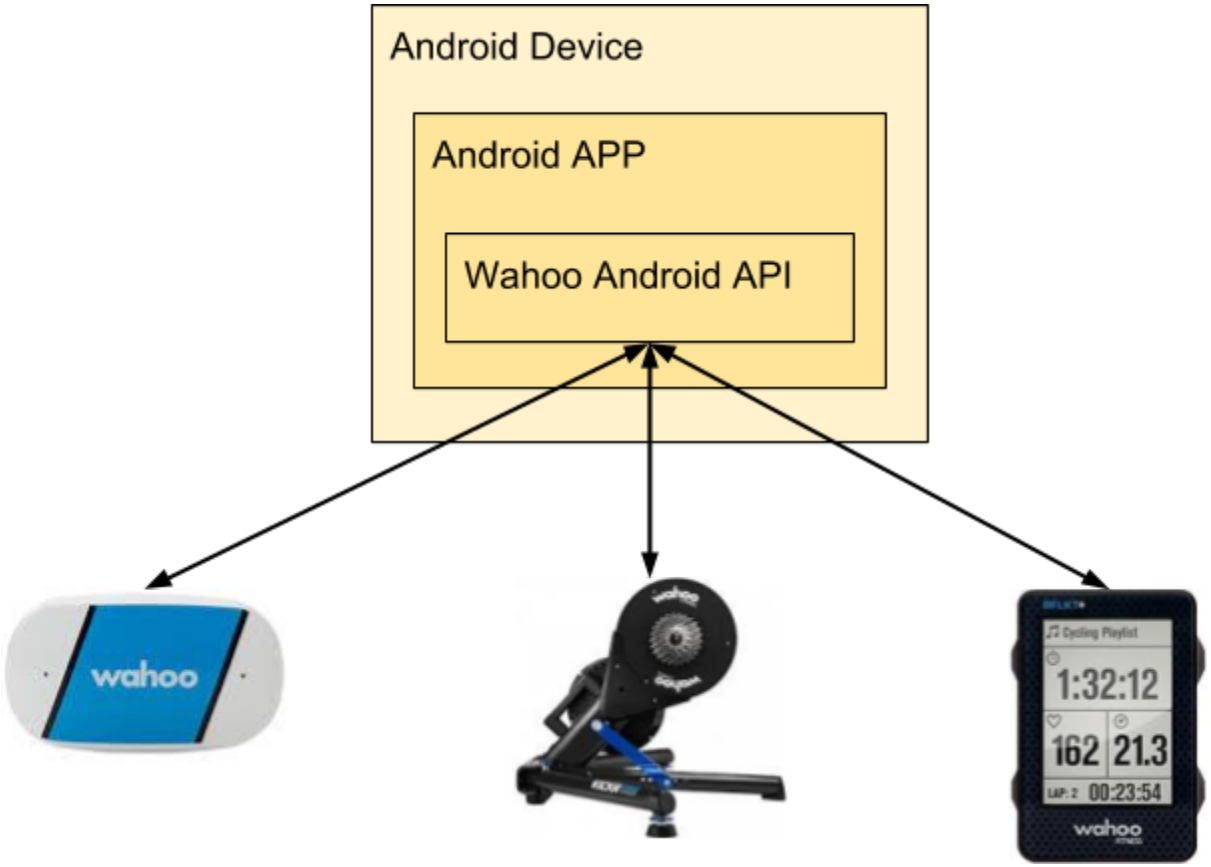
Wahoo Android API Users Manual

1 Introduction

The Wahoo Android API is an aar file that can be included in your Android app, giving you the ability to connect and interact with Wahoo Fitness devices as well as other fitness equipment, over BT or ANT+.



2 Using the API in your Android APP

2.1 Setup the project

To import the .aar library file, in Android Studio

1. Go to File > New > New Module
2. Select "Import .JAR/.AAR Package" and click next.
3. Enter the path to .aar file and click finish.
4. Go to File > Project Structure
5. Under "Modules," in left menu, select your app
6. Go to the "Dependencies" tab.
7. Click the green "+" in the lower left corner.
8. Select "Module Dependency"
9. Select the new module from the list.

Ensure your app's **manifest** has the following permissions

- android.permission.BLUETOOTH
- android.permission.BLUETOOTH_ADMIN
- android.permission.WAKE_LOCK
- android.permission.INTERNET
- android.permission.ACCESS_NETWORK_STATE
- android.permission.ACCESS_FINE_LOCATION

2.2 Using the HardwareConnector class via a Service

The main class for interacting with the Wahoo Android API is the **HardwareConnector** class. It is recommended to interact with this class via an Android **Service** component to ensure the same **HardwareConnector** object instance persists through the life-cycle of your app.

- Create an **android.app.Service** component.
    - See http://developer.android.com/guide/components/services.html for more information on Services
- In your **Service** class, declare a **HardwareConnector** and initialize a **HardwareConnector.Listener** member

```
private HardwareConnector mHardwareConnector;
private final HardwareConnector.Listener mHardwareConnectorListener = new HardwareConnector.Listener() {...}
```

- In your **Service.onCreate()**, instantiate your **HardwareConnector**

```
@Override
public void onCreate() {
        super.onCreate();
        mHardwareConnector = new HardwareConnector(this, mHardwareConnectorListener);
}
```

- In your **Service.onDestroy()**, shutdown your **HardwareConnector**

```
@Override
public void onDestroy() {
        super.onDestroy();
        mHardwareConnector.shutdown();
}
```

- Your Activities and Fragments can then gain access to the **HardwareConnector** by binding to the your service

2.3 API LogCat Logging

The API has 5 levels of logcat logging; VERBOSE, DEBUG, INFO, WARNING, ERROR. By default, the log level is set to WARNING.

During development of your app, or when reporting issues to Wahoo Fitness, it is important to increase the log level to VERBOSE.

To set the Wahoo API logcat log level to VERBOSE, add the following line before instantiating your **HardwareConnector**

```
private void initWahooHardwareConnector()
{
        com.wahoofitness.common.log.Logger.setLogLevel(android.util.Log.VERBOSE);
        mHardwareConnector = new HardwareConnector(getContext(), mHardwareConnectorListener);
}
```

2.3.1 Sending Logs to Wahoo

When sending support requests to Wahoo, please attach a log file, set to log level VERBOSE, of the scenario.

1. Enable VERBOSE logging as shown above
2. Kill the app
3. Start the app
4. Connect your device and reproduce the issue
5. Disconnect the device
6. Save & send logs
7. Clearly describe the issue and the time it occurred (the time helps us find the issue in the logs)

2.4 Discovering Devices

- Before we can connect to devices, we need to discover them. To discover devices, call `mHardwareConnector.startDiscovery();`
- This will start an asynchronous discovery procedure which will return **ConnectionParams** objects (via the **DiscoveryListener**) as new devices are found.
- **ConnectionParams** represent a 'key' for connecting to devices. They are used to connect to devices using `mHardwareConnector.requestSensorConnection(...);`
- Do not stay in discovery mode for longer than needed as this is a drain on the battery. To stop discovery, call `mHardwareConnector.stopDiscovery();`
- You can serialize and save **ConnectionParams** for later use. This means you do not need to perform a discovery every time you app starts, you simply call **requestSensorConnection(...)** with your deserialized **ConnectionParams**

2.5 Connecting to Devices

- A connection with a physical device is modeled using a **SensorConnection** object. To request a **SensorConnection** from the API, call `mSensorConnection = mHardwareConnector.requestSensorConnection(...);`
- To disconnect your **SensorConnection**, call `mSensorConnection.disconnect();`

2.6 Capabilities

- **Capabilities** are used to interact with the connected devices
- A capability is a Java interface provided by the **SensorConnection** which allows the application to interact with the connected device in a particular way
- For example, the API has capabilities for getting heartrate data, crank revs data, wheel revs data, as well as performing functions like firmware upgrade, etc. For a full list of capabilities, see **com.wahoofitness.connector.capabilities.Capability.CapabilityType**
- The application can query the **SensorConnection's** current supported capability types using `mSensorConnection.getCurrentCapabilities(...);`
- The application can then get the actual Capability interface object by calling `mSensorConnection.getCurrentCapability(..);`
- It is important to understand that a **SensorConnection's** capabilities are dynamic and may not be available immediately after creation. For example, the **Heartrate** capability is not published until the **SensorConnection** has received its first heartrate measurement from the physical device.
- The application is notified of new capabilities via the **SensorConnection.Listener.onNewCapabilityDetected()** callback

2.6.1 Heartrate Capability

To register for heartrate updates

```
private final SensorConnection.Listener mSensorConnectionListener = new SensorConnection.Listener() {
	@Override
	public void onNewCapabilityDetected(SensorConnection sensorConnection, CapabilityType capabilityType) {
		if(capabilityType == CapabilityType.Heartrate) {
			Heartrate heartrate = (Heartrate)sensorConnection.getCurrentCapability(CapabilityType.Heartrate);
			heartrate.addListener(mHeartRateListener);
		}
	}
}
```

To query heartrate data

```
Heartrate.Data getHeartrateData() {
	if (mSensorConnection != null) {
		Heartrate heartrate = (Heartrate) mSensorConnection
				.getCurrentCapability(CapabilityType.Heartrate);
		if (heartrate != null) {
			return heartrate.getHeartrateData();
		} else {
			// The sensor connection does not currently support the heartrate capability
			return null;
		}
	} else {
		// Sensor not connected
		return null;
	}
}
```

2.6.2 Crank Revolutions Capability

To register for crank revolution updates

```
private final SensorConnection.Listener mSensorConnectionListener = new SensorConnection.Listener() {
	@Override
	public void onNewCapabilityDetected(SensorConnection sensorConnection, CapabilityType capabilityType) {
		if(capabilityType == CapabilityType.CrankRevs) {
			CrankRevs crankRevs = (CrankRevs)sensorConnection.getCurrentCapability(CapabilityType.CrankRevs);
			crankRevs.addListener(mCrankRevsListener);
		}
	}
}
```

To query crank revolution data

```
CrankRevs.Data getCrankRevsData()
{
	if(mSensorConnection != null) {
		CrankRevs crankRevs = (CrankRevs)mSensorConnection.getCurrentCapability(CapabilityType.CrankRevs);
		if (crankRevs != null) {
			return crankRevs.getCrankRevsData();
		} else {
			// The sensor connection does not currently support the crank revs capability
			return null;
		}
	} else {
		// Sensor not connected
		return null;
	}
}
```

2.6.3 BikeTrainer Capability (KICKR)

To put KICKR into standard mode

```
void setKickrToStandardMode(int level)
{
	if(mSensorConnection != null) {
		BikeTrainer bikeTrainer = (BikeTrainer) mSensorConnection.getCurrentCapability(CapabilityType.BikeTrainer);
		if (bikeTrainer != null) {
			bikeTrainer.sendSetStandardMode(level);
		} else {
			// The sensor connection does not currently support the bike trainer capability...
		}
	} else {
		// Sensor not connected
		return null;
	}
}
```

2.6.4 RFLKT / ECHO / TIMEX Display Devices

ECHO/RFLKT/TIMEX requires what we call a "confirmed connection" to work properly. This involves an additional sequence of handshake messages between the phone and device, just after BT connection. This procedure typically involves in a popup message on the device saying 'connect to phone XYZ?' with a yes/no option. If yes, the connection is deemed confirmed.

```
/**
 * {@link Poller} for managing the connection and display update for the ECHO/RFLKT display
 * devices
 */
private final Poller mDisplayPoller = new Poller(1000) {

	@Override
	protected void onPoll() {

		// Get the capabilities
		ConfirmConnection confirmConnection = getConfirmConnectionCapability();
		Rflkt rflkt = getRflktCapability();

		// If the capabilities are present
		if (confirmConnection != null && rflkt != null && displayConfiguration != null) {

			State confirmationState = confirmConnection.getState();

			// If connection has not yet been confirmed...
			if (confirmationState == ConfirmConnection.State.READY) {

				// Confirm the connection
				boolean ok = confirmConnection.requestConfirmation(
						ConfirmConnection.Role.MASTER, mDeviceName, mAppUuid, mAppName);

				// On failure, the Poller will simply try again later

			} else if (confirmationState == ConfirmConnection.State.ACCEPTED) {

				// The connection has been confirmed

				// If we're not currently loading display config
				if (!rflkt.isLoading()) {

					// If the loaded configuration up to date
					if (displayConfiguration.equals(rflkt.getDisplayConfiguration())) {

						// Update the values
						Date timeNow = Calendar.getInstance().getTime();
						rflkt.setValue(DisplayDataType.TIME_CURRENT_HHMMSS.getUpdateKey(),
								mTimeFormat_hhmmss.format(timeNow));
```

```
                                        } else {
                                                // Config mismatch - need to load the desired config
                                                rflkt.loadConfig(displayConfiguration);
                                        }
                                }
                        }
                }
        }
};

private final SensorConnection.Listener mSensorConnectionListener = new SensorConnection.Listener() {

        @Override
        public void onSensorConnectionStateChanged(SensorConnection sensorConnection,
                        SensorConnectionState sensorConnectionState) {

                if (sensorConnection.isConnected()) {
                        mDisplayPoller.start();
                } else {
                        mDisplayPoller.stop();
                }
        }
};


private ConfirmConnection getConfirmConnectionCapability() {
        ConfirmConnection confirmConnection = null;
        if (mSensorConnection.isConnected()) {
                confirmConnection = (ConfirmConnection) mSensorConnection
                                .getCurrentCapability(CapabilityType.ConfirmConnection);
        }
        return confirmConnection;
}

private Rflkt getRflktCapability() {
        Rflkt rflkt = null;
        if (mSensorConnection.isConnected()) {
                rflkt = (Rflkt) mSensorConnection.getCurrentCapability(CapabilityType.Rflkt);
        }
        return rflkt;
}
```

2.6.4.1 Additional Notes Regarding TIMEX Watches

Because TIMEX watches have a specific requirement to be paired to their **Timex Connected app** before they can be used, TIMEX devices are not scanned (in the traditional sense) by the Wahoo Fitness API, in the same way as other devices.

Instead, when a discovery procedure is started in the API, the API will query the installed Timex Connected app for the address of the locally paired TIMEX watch and return this ConnectionParams in the discovery results.

If the TIMEX Connected app is not installed, or the TIMEX is not paired with the TIMEX Connected app, the Wahoo Fitness API will not return the TIMEX in its discovery results.

3 Firmware Upgrade

When the Wahoo Android API detects that a connected device requires a firmware upgrade, it will notify the app via the callback `HardwareConnector.Listener.onFirmwareUpdateRequired(...)`

It is highly recommended the APP then calls the following code to initiate a firmware upgrade procedure, via the Wahoo Utility APP.

```
private final HardwareConnector.Listener mHardwareConnectorCallback = new HardwareConnector.Listener() {

	@Override
	public void onFirmwareUpdateRequired(SensorConnection sensorConnection,
			String currentVersion, String recommendedVersion) {

		// Request if user wants to perform the firmware upgrade
		// On accept, call performFirmwareUpgrade()
	}
};

public void performFirmwareUpgrade(SensorConnection sensorConnection)
{
	// Launches the Wahoo Utility App and starts a firmware upgrade.
	// If the app is not installed, opens the Google Play Store to download the app
	sensorConnection.disconnect();
	WahooUtilityLauncher.launch(MainService.this, sensorConnection);
}
```

4 Licence Agreement

Please see the EULA here