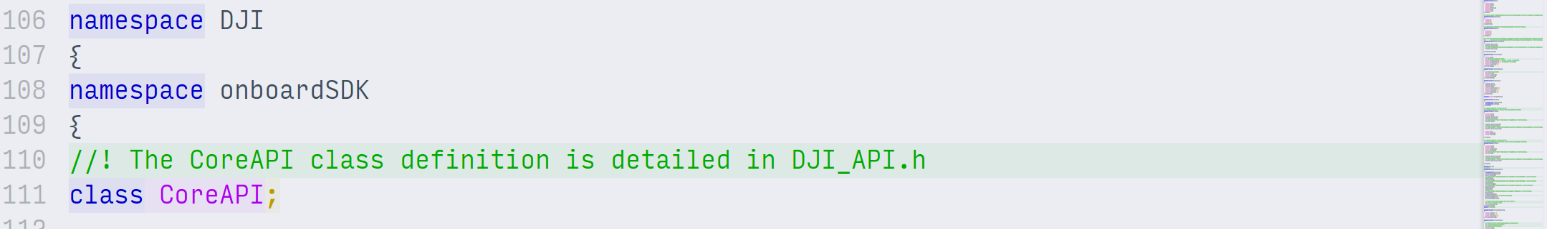
2016-10-14v3.1.9



**Introduction**

This guide helps users understand Onboard SDK programming paradigms and correct usage of the APIs provided by the DJI Onboard SDK library. The guide walks through the two asynchronous programming paradigms that are supported by the SDK, and provides information about DJI Onboard SDK workflow.

All structures and functions are declared in DJI\_API.h, DJI\_Type.h and DJICommonType.h. For more details, please refer to the source code - Doxygen documentation for the API is also provided.

For good example usage of the APIs and the onboard SDK in general, we encourage you to take a look at the sample programs ([Linux](https://developer.dji.com/onboard-sdk/documentation/M100-Docs/github-platform-docs/Linux/README.html), [ROS](https://developer.dji.com/onboard-sdk/documentation/M100-Docs/github-platform-docs/ROS/README.html), [STM32F4](https://developer.dji.com/onboard-sdk/documentation/M100-Docs/github-platform-docs/STM32/README.html) and [Windows (on Qt)](https://developer.dji.com/onboard-sdk/documentation/M100-Docs/github-platform-docs/PureQT/README.html)).

**Software Components**

**Serial Device Driver**

Since the Onboard SDK communicates with the Flight Controller on the aircraft entirely via serial communication, it is necessary to have an efficient serial device driver that implements I/O, memory locking/unlocking and ways for the serial channel to talk to or interrupt other threads in the program.

We provide implementations of serial drivers in our Qt sample, in the ROS sample and a generic Linux platform device driver in the platform folder, used by the new (3.1.8) Linux sample.

**Threading/Interrupts**

The serial communication described above must always run - and we usually want to execute other processes in the foreground while I/O goes on. To make this happen, the preferred method is to spawn a read thread, and let the main flow of the program continue in the main thread. In order to allow inter-thread communication, a method of signaling also needs to be implemented, or the main thread must continuously poll for changes from the read thread.

We provide threading implementations for all our x86 and ARM-based samples. For embedded systems where threading is not an option, an interrupt-based mechanism must be implemented. We provide one such mechanism as part of the STM32F4 sample.

**Application-Layer API calls**

This is the part of the system where functionality for controlling the drone is implemented. In the core Onboard SDK library, we provide a number of APIs that provide interfaces for all functionality made available through the [Open Protocol](https://developer.dji.com/onboard-sdk/documentation/protocol-doc/open-protocol.html). The sample programs provided execute combinations and compositions of the library API calls together with some logic.

**Programming Paradigms**

The core of the Onboard API is a command (CMD)-acknowledgement(ACK) communication for executing actions; the OES sends CMD packets to the Flight Controller containing information about the action it is requesting. The flight controller, in return, replies with an ACK packet confirming or rejecting the CMD. SDK-side code must therefore implement some mechanism of relaying this ACK information back to the user or the CMD-sending function, so that the caller can take further action based on the ACK response.

The Onboard SDK supports two ways of doing this, described below.

**Asynchronous Programming - Callback Mechanism**

For true asynchronous programming, the SDK needs to allow for a mechanism for the user to execute code upon receiving ACKs from the aircraft independent of the main flow of execution. The Onboard SDK provides this through **callback functions**.

Upon receiving the ACK from the aircraft, the SDK's read function calls the user-provided callback function. The user is also allowed to supply some data (UserData, implemented as a void\*), which is passed as an argument to the user callback. The user callback has access to the ACK and can implement logic based on the ACK. UserData is meant for two purposes: (1) the user can also pass any additional information to the callback that the callback may act on depending on the ACK parsing logic (2) it is a mechanism for the callback to pass data back to user, if the user passes a reference to a variable that can be populated in the callback based on the ACK.

The Linux sample now incorporates Asynchronous programming through the Non-Blocking sample. A separate callback thread runs all the callback functions.

**Example:** Non-Blocking Take Control Linux function.

1. The Take Control command can be called from user code as follows

|  |
| --- |
| takeControlNonBlocking(api); |

1. A default callback is made available with CoreAPI, Protocol Header and UserData passed to the Callback function. Additionally, you can access the ACK for the command in the callback function.

|  |
| --- |
| void takeControlCallback(CoreAPI\* api, Header \*protHeader, DJI::UserData data) {  uint16\_t simpAck = api->missionACKUnion.simpleACK;  switch (simpAck)  {  case ACK\_SETCONTROL\_NEED\_MODE\_F:  std::cout << "Failed to obtain control.\nYour RC mode switch is not in mode F. (Is the RC connected and paired?)" << std::endl;  break;  case ACK\_SETCONTROL\_OBTAIN\_SUCCESS:  std::cout << "Obtained control successfully."<< std::endl;  break;  case ACK\_SETCONTROL\_OBTAIN\_RUNNING:  std::cout << "Obtain control running.."<< std::endl;  takeControlNonBlocking(api);  break;  case ACK\_SETCONTROL\_IOC:  std::cout << "The aircraft is in IOC mode. Cannot obtain control.\nGo to DJI GO and stop all intelligent flight modes before trying this." << std::endl;  break;  default:  {  std::cout << "Error in setControl API function." << std::endl;  }  break;  }  } |

1. The meaning of ACK return values(result) is explained for each command in the [Flight ACK Codes](https://developer.dji.com/onboard-sdk/documentation/M100-Docs/application-development-guides/flight-mission-ack-codes.html) document.

Note: Callbacks, in this implementation is executed in a separate callback thread, but blocks the 'read' thread with mutexes until the callback is complete. Do not execute large pieces of code in callbacks; reading the serial port is prevented until the callback finishes.

**Example:** Activation function in QT sample

1. Declare and define the callback function in your user code.

|  |
| --- |
| static void activationCallback(CoreAPI\* api, Header\* protocolHeader, UserData userData); |

1. Callback functions must have the same function prototype as above. Use UserData to send void\* data and then cast it to appropriate datatypes in your callback function.
2. Pass the callback function when you call activate.

|  |
| --- |
| api->activate(&activateData, DJIonboardSDK::activationCallback, this); |

1. Here, we call the overloaded activate function that takes in a user-specified callback and some user-provided data as the second and third arguments. The prototype for all API functions can be found in [DJI\_API.h](https://github.com/dji-sdk/Onboard-SDK/blob/3.1/osdk-core/inc/DJI_API.h).
2. The meaning of ACK return values(result) is explained for each command in the [Flight ACK Codes](https://developer.dji.com/onboard-sdk/documentation/M100-Docs/application-development-guides/flight-mission-ack-codes.html) document.

Note: Callbacks, in the current implementation, are executed on the read thread. Do not execute large pieces of code in callbacks; reading the serial port is prevented until the callback finishes.

**Synchronous Programming - Blocking API Calls**

The SDK provides a clean way to implement blocking API calls through its signaling-based thread synchronization. Blocking calls are very convenient because a blocking call only returns after the CMD-ACK round trip is done. When the blocking call returns, users are assured that the command has executed, and the ACK is available in user code as the blocking call's return value. As a result, users do not need to implement callbacks to take decisions or execute code based on ACKs.

Blocking calls are great for maintaining a linear flow of execution. They are also free from the issues arising executing heavy code in callbacks.

**Example:** Take Control function in new(3.1.8) Linux sample

1. Call the blocking overload for activate.

|  |
| --- |
| unsigned short takeControlAck = api->setControl(true, 1); |

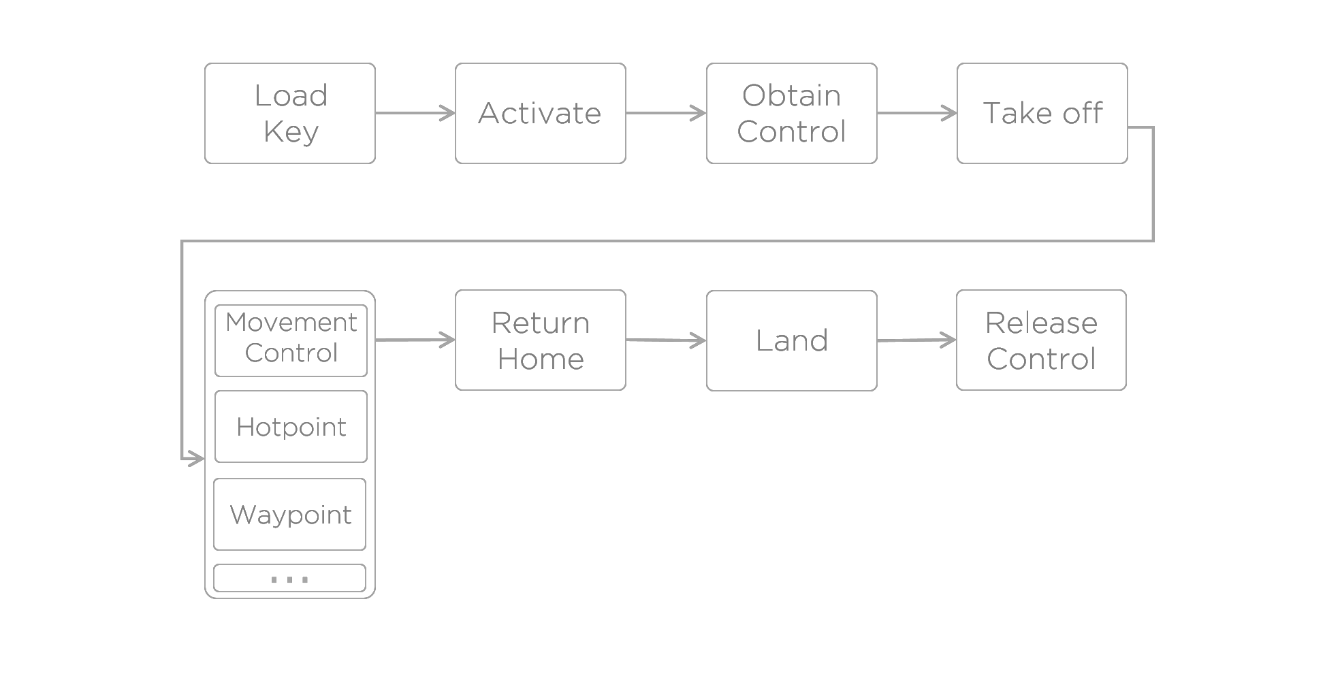
1. This function prototype takes in a timeout as the second parameter - this is to prevent the blocking function from blocking execution indefinitely if the ACK is not received - when there is a packet corruption, for example.
2. Parse the returned ACK according to the [Flight ACK Codes](https://developer.dji.com/onboard-sdk/documentation/M100-Docs/application-development-guides/flight-mission-ack-codes.html) document.

|  |
| --- |
| switch (takeControlAck)  {  case ACK\_SETCONTROL\_NEED\_MODE\_F:  std::cout << "Failed to obtain control.\nYour RC mode switch is not in mode F. (Is the RC connected and paired?)" << std::endl;  takeControlData.status = -1;  return takeControlData;  break;  case ACK\_SETCONTROL\_OBTAIN\_SUCCESS:  std::cout << "Obtained control successfully."<< std::endl;  break;  case ACK\_SETCONTROL\_OBTAIN\_RUNNING:  std::cout << "Obtain control running.."<< std::endl;  takeControl(api);  break;  case ACK\_SETCONTROL\_IOC:  std::cout << "The aircraft is in IOC mode. Cannot obtain control.\nGo to DJI GO and stop all intelligent flight modes before trying this." << std::endl;  takeControlData.status = -1;  return takeControlData;  break;  default:  {  std::cout << "Error in setControl API function." << std::endl;  }  break;  } |

1. The enums for the cases above are defined in the SDK.

**Onboard SDK Programming Workflow**

When calling APIs, developers should make sure that the sequence of events follows this chart:



There are some exceptions to this workflow - Virtual RC and arming/disarming are not part of this flow, for example - but for most cases this workflow suffices.

**Receiving Flight Data**

The above workflow talks about the *sending* side of user code; another important part is reading the aircraft's flight data. A full list of all the data that can be received is available on the SDK tab of DJI Assistant 2.

The aircraft will always send this data at the frequencies set here (or set through the Onboard SDK's setBroadcastFreq() API call) over the serial port.

To save this flight data, developers need to declare correct structs. The [DJI\_Type.h](https://github.com/dji-sdk/Onboard-SDK/blob/3.1/osdk-core/inc/DJI_Type.h) file contains sample structs for correctly accepting all kinds of broadcast data.

This is the BroadcastData struct definition in DJI\_Type.h:

|  |
| --- |
| typedef struct BroadcastData  {  unsigned short dataFlag;  TimeStampData timeStamp;  QuaternionData q;  CommonData a;  VelocityData v;  CommonData w;  PossitionData pos;  MagnetData mag;  RadioData rc;  GimbalData gimbal;  FlightStatus status;  BatteryData battery;  CtrlInfoData ctrlInfo;  //! @note these variables are not send from FMU,  //! just a record for user.  uint8\_t controlStatus;  uint8\_t activation;  } BroadcastData; |

**Example:** Get quaternion data:

1. Declare quaternion struction

|  |
| --- |
| typedef struct QuaternionData  {  float32\_t q0;  float32\_t q1;  float32\_t q2;  float32\_t q3;  } QuaternionData;  QuaternionData q; |

1. Get the quaternion

|  |
| --- |
| q = flight->getQuaternion() |

**Controlling the Aircraft's Flight**

**Movement Control**

Movement control is a mechanism through which users can command the drone to execute varied, complex maneuvers and custom missions.

We recommend developers send their movement control commands at 50Hz frequency. Developers can implement that by calling usleep(20000), ros::Duration(1/50) or other ways which depend on the development environment.

For Movement Control, specific meanings of arguments are decided by the control mode byte. For more info about Movement Control, please refer to the [Control mode byte](https://developer.dji.com/onboard-sdk/documentation/appendix/index.html#Control-Mode-Byte) section in the Appendix.

To execute horizontal movement, developers can use the HORI\_POS mode (0x91) for horizontal movement. More details are shown in [Position Control(HORI\_POS)](https://developer.dji.com/onboard-sdk/documentation/M100-Docs/application-development-guides/programming-guide.html#position-control-hori-pos) in this document. In this mode, speed and attitude are controlled by autopilot, thus developers do not need to worry about that.

We provide higher-level APIs for common control modes - position control, attitude control and velocity control - through simple functions in the new Linux example. We recommend you try these functions out and view their implementations before calling API-level movement control functions.

Please note that certain conditions are required for some control modes to be functional:

* Only when the GPS signal is good (health\_flag >=3)，horizontal position control (HORI\_POS) related control modes can be used.
* Only when GPS signal is good (health\_flag >=3)，or when Gudiance system is working properly with Autopilot，horizontal velocity control(HORI\_VEL) related control modes can be used.

**Example**: Direct API call to setFlightControl()

|  |
| --- |
| FlightData data;  data.flag = flightFlag;  data.x = flightx;  data.y = flighty;  data.z = flightz;  data.yaw = flightyaw;  flight->setFlightControl(&data); |

**Example**: Linux example call to move the aircraft by 10 m in the forward (x) direction:

|  |
| --- |
| int positionControlStatus = moveByPositionOffset(api, flight, 10, 0, 0, 0); |

**Position Control(HORI\_POS)**

The HORI\_POS control mode is a relative offset control mode; the input horizontal arguments are the offset between current position and target position. The unit of offset is meters.

For example, in ground frame, target is target position and current is UAV's current position. The coordinates of these positions are calculated by GPS, Guidance or other sensors. In most cases, GPS is the correct way to make this work.

Because, for the autopilot, the maximum frequency of receiving data is 50Hz, the frequency of calculating offset should be over 50Hz to ensure the closed-loop control is valid; you always want to have a new offset calculated before you send it to the flight controller to execute.

**Example"**: Writing position control procedure:

|  |
| --- |
| void update\_offset()  {  offset\_x = target\_x - current\_x;  offset\_y = target\_y - current\_y;  }  /\* Command thread \*/  FlightData data;  data.flag = 0x90;  data.x = offset\_x;  data.y = offset\_y;  data.z = target\_z;  data.yaw = target\_yaw;  flight->setFlight(&data); |

As mentioned before, we have implemented this checking procedure in the moveByPositionOffset Linux example call.

**Math Utilities**

This information will help you go back and forth between the various coordinate-systems and methods of angle representation used in the code.

**GPS to North-East Coordinate**

Convert GPS to North-East-Down Coordinate. (GPS in radian，North-East Coordinate in meter) For example, origin\_longti and origin\_lati , as the longitude and latitude of original position, are decided by developers. The position of UAV takeoff is recommended to be set as the original position. longti and lati are longitude and latitude of UAV's current position. x and y are offsets to the original position in the local frame North and the East directions. The unit of offset is meter.

|  |
| --- |
| #define C\_EARTH (double) 6378137.0  /\* From GPS to Ground \*/  {  double dlati = lati-origin\_lati;  double dlongti= longti-origin\_longti;  double x = dlati \* C\_EARTH;  double y = dlongti \* C\_EARTH \* cos(lati / 2.0 + origin\_lati / 2.0);  } |

**Quaternion to RPY**

Convert quaternion to roll, pitch and yaw in radian in body coordinate. A fairly good introduction to quaternions can be found [here](http://www.chrobotics.com/library/understanding-quaternions).

|  |
| --- |
| api\_quaternion\_data\_t q;  DJI\_Pro\_Get\_Quaternion(&q);  float roll = atan2(2.0 \* (q.q3 \* q.q2 + q.q0 \* q.q1) , 1.0 - 2.0 \* (q.q1 \* q.q1 + q.q2 \* q.q2));  float pitch = asin(2.0 \* (q.q2 \* q.q0 - q.q3 \* q.q1));  float yaw = atan2(2.0 \* (q.q3 \* q.q0 + q.q1 \* q.q2) , - 1.0 + 2.0 \* (q.q0 \* q.q0 + q.q1 \* q.q1)); |