# SDK Reference

**Shadow** Version 4.0

www.motionshadow.com

# Contents

# 1 Introduction

The Motion Software Development Kit (SDK) is a collection of classes that provides real-time access to the output of the Motion Service. This includes orientation output as well as the raw and calibrated sensor signals. The SDK is open source and available in the C++, C#, Java, and Python programming languages.

The SDK is not required to access data from Motion Service. It is intended to simplify development of third-party applications. The SDK can also be used as a reference implementation for tighter integration with external application infrastructure.

This document provides an overview of the features of the SDK as well as a common technical reference for all supported programming languages.

# 2 Real-time Data Streams

The Motion Service exports real-time data streams over TCP socket connections. A client application opens a connection to the Motion Service on a specific port. The service will send the current output data to the client as soon as it becomes available. This allows the client application to read the real-time output of the filtering pipeline just as if it were part of the Motion system. The Motion Service exports the combined output from all configured devices as a single associative container.

The SDK classes access data streams from the Motion Service. All device configuration and management is handled through the desktop application or through scripting commands.

There are four separate types, or formats, of data output streams. There are four data *Services* running inside the Motion Service, each listening on its own port for an incoming client connection. The three formats are called *Preview*, *Sensor*, *Raw*, and *Configurable*.

The *Preview* format consists of the orientation output, the final output of the filtering pipeline. The *Sensor* format is the calibrated output of the individual accelerometer, gyroscope, and magnetometer sensors. The *Raw* format is the unprocessed output each individual sensor. The *Configurable* format provides all channels that the client requests.

By default, the *Preview* service listens on port 32079, the *Sensor* service listens on port 32078, the *Raw* service listens on port 32077, and the *Configurable* service listens on port 32076.

## 2.1 Preview

The Preview service provides access to the current orientation output as a quaternion, a set of Euler angles, or a 4-by-4 rotation matrix. The orientation output can be accessed in the global or local coordinate frame. The Preview service also provides a current estimate of linear acceleration in the global coordinate frame.

[global quaternion, local quaternion, local euler, global acceleration]
$$\{ \ ^{G}q_w \quad ^{G}q_x \quad ^{G}q_y \quad ^{G}q_z \quad ^{L}q_w \quad ^{L}q_x \quad ^{L}q_y \quad ^{L}q_z \quad r_x \quad r_y \quad r_z \quad la_x \quad la_y \quad la_z \ \}$$

Figure 1: Preview service data format.

The *global* coordinate frame is defined by gravity and the geomagnetic field. The global identity orientation is `Y` pointed up in the direction of gravity and `X` pointed towards the geographic pole. The global quaternion is the primary output of the filtering pipeline and it is used to compute the other *Preview* elements. Where applicable, orientations are specified in a right-handed coordinate system.

The *local* coordinate frame is expressed relative to:

- an arbitrary start orientation,

- a user defined rest pose orientation,

- and with respect to a parent orientation.

By default, the *local* orientation is a rotation about the global axes. There are two additional local rotation modes for convenience. Use the Lua command `node.system.set_local_mode()` to switch between the rotation modes.

- *Sensor*, rotate about the local axes of the sensor defined at the start time.

- *World + Heading*, rotate about the global axes with the `X` axis pointing forward. The forward direction is defined by the rotation about the vertical axis at the start time.

The configuration defines the rest pose and the parent relationships. The Lua `node.define_identity_pose()` command or the User Interface `Node > Set Pose` menu command set the start orientation. Note that starting a take always defines a start orientation.

Each quaternion is specified as a four element array in `q = [w, x, y, z]` order where $q = w + x\hat{i} + y\hat{j} + z\hat{k}$ and $\sqrt{w^2 + x^2 + y^2 + z^2} = 1$. The Euler angle set is specified in radians assuming an `x-y-z` rotation order and a right handed coordinate system. The linear acceleration vector is specified in $g$.

## 2.2 Sensor

The Sensor service provides access to the current individual accelerometer, gyroscope, and magnetometer signals in real units. The accelerometer signals are specified in $g$, where $1\,g = 9.80665\,\mathrm{meter}/\mathrm{second}^2$, and are on the domain [-2, 2] or [-6, 6] based on the current configuration. The gyroscope signals are specified in $\mathrm{degree}/\mathrm{second}$ and are on the domain [-500, 500] or [-2000, 2000]. The magnetometer signals are specified in $\mu T$, microtesla. The domain of the magnetometer signal varies based on geographic location, but expect values on the domain [-80, 80].

[accelerometer, magnetometer, gyroscope]
$\{\quad a_x \quad a_y \quad a_z \quad m_x \quad m_y \quad m_z \quad g_x \quad g_y \quad g_z \quad \}$

Figure 2: Sensor service data format.

## 2.3 Raw

The Raw service provides access the current unprocessed accelerometer, gyroscope, and magnetometer signals. The values of all signals are on the domain of the analog to digital conversion on the MotionNode device. The Raw data signals are specified as 16-bit signed integers.

MotionNode modules with 500 $\mathrm{degree}/\mathrm{second}$ gyroscopes measure all channels on the domain [0, 4095]. This range corresponds to 12-bit unsigned raw measurements.

MotionNode modules with 2000 $\mathrm{degree}/\mathrm{second}$ measure all channels as signed values centered about zero. The range varies by sensor type. The accelerometer measures in 15-bit integers, the gyroscope measures in 16-bit integers, and the magnetometer measures in 12-bit integers.

Shadow modules with 4000 $\mathrm{degree}/\mathrm{second}$ measure all channels as signed 16-bit integer values centered about zero.

[accelerometer, magnetometer, gyroscope]
$\{\quad A_x \quad A_y \quad A_z \quad M_x \quad M_y \quad M_z \quad G_x \quad G_y \quad G_z \quad \}$

Figure 3: Raw service data format.

## 2.4 Configurable

The Configurable service provides access to all outputs of the Motion Service in a single container. The client application must send a list of channels when

it connects to the Configurable data service. The data service will write the channels in the order they are specified in this list.

The list of channels is specified as an XML document. See the current list of available channels in the example Configurable Service Definition file.

# 3  Classes

The SDK classes are available in multiple programming languages. The differences in usage are due mostly to the different semantics of each language. This section provides a common technical reference for all supported languages. For more language specific information, refer to the example source files.

## 3.1  Client

The Client class is responsible for communication with the Motion Service. The Client class opens a connection to one of the real-time data services. This connection is simply a TCP socket link with a binary message protocol. The Client class implements data streaming over the socket connection and the message protocol.

Use the Format class to interpret the binary message.

**`Client(String host, Integer port)`**

  **Postcondition**    Open connection to the service running on host:port.

**`close()`**

    **Precondition**    Open connection to a service.
  **Postcondition**    No open connection.

**`isConnected()`**

      **Return**    a true value if the current connection is open and active

**`readData([Integer time_out_second])`**

    **Precondition**    Open connection to a service.
  **Postcondition**    Returns the current incoming binary message.
      **Parameter**    `time_out_second` specifies a time out for this call, 0 denotes blocking call, -1 denotes default value (1 second)

```
try {
  using Motion::SDK::Client;

  // Open connection to a Motion Service on the
  // localhost, port 32079.
  Client client("", 32079);

  // This is application dependent. Use a loop to
  // keep trying to read data even after time outs.
  while (true) {
    // Wait until there is incoming data on the
    // open connection, timing out after 5 seconds.
    if (client.waitForData()) {
      // Read data samples in a loop until we time out or
      // the connection closes.
      Client::data_type data;
      while (client.readData(data)) {
        using Motion::SDK::Format;

        // Do something useful with the current binary
        // message, maybe use the Format class.
        Format::preview_type preview =
          Format::Preview(data.begin(), data.end());

        // Iterate through the list of
        // [id] => PreviewElement objects.
        Format::preview_type::iterator itr=preview.begin();
        for (; itr!=preview.end(); ++itr) {
        }
      }
    }
  }

} catch (std::runtime_error & e) {
  // The Client class with throw std::runtime_error
  // for any error conditions. Even if the connection
  // to the remote host fails.
}
```

Example 1: Client class example usage (C++).

**waitForData([Integer time_out_second])**

| | |
|---|---|
| **Precondition** | Open connection to a service. |
| **Postcondition** | Incoming data is available. |
| **Parameter** | time_out_second specifies a time out for this call, 0 denotes blocking call, -1 denotes default value (5 seconds) |

**writeData(Array data, [Integer time_out_second])**

| | |
|---|---|
| **Precondition** | Open connection to a service. |
| **Postcondition** | Successfully wrote outgoing binary message. |
| **Parameter** | data array of bytes, binary message |
| | time_out_second specifies a time out for this call, 0 denotes blocking call, -1 denotes default value (1 second) |

## 3.2   File

The File class provides an interface for reading Motion binary take files. The File class reads a single sample at a time into an array of typed elements. It provides access analagous to reading data over a real-time Client connection, but only for a single MotionNode device.

   The File class is only intended for use with the Sensor and Raw data files. The entire Preview data format is not available from a Motion take file.

```
# Open take data file in the Sensor format.
# Print out the calibrated gyroscope signals.
take_file = File("sensor.bin")
while True:
    # Requires that we specify:
    # 1. Number of elements in a single sample
    # 2. True if the elements are real valued
    data = take_file.readData(9, True)
    if None == data:
        break

    # Use the Format class to access the data.
    print Format.SensorElement(data).getGyroscope()
```

Example 2: File class example usage (Python).

**File(String pathname)**

| | |
|---|---|
| **Postcondition** | Open binary data file stream from pathname. |

`close()`

| | |
|---|---|
| **Precondition** | Open binary data file stream. |
| **Postcondition** | No open file stream. |

`readData(Integer length)`

| | |
|---|---|
| **Precondition** | Open binary data file stream. |
| **Postcondition** | Returns the current binary data sample. |
| **Parameter** | `length` specifies the number of samples to read |

## 3.3 Format

The Format class wraps a binary message from the Client class, or an array of typed elements from the File class in an object representation of that message or array. For example, the Sensor data service provides the current calibrated acclerometer signal. The Format.SensorElement class implements a `getAccelerometer` method that returns the value of this set of signals.

`Configurable(Array data)`

| | |
|---|---|
| **Summary** | Convert a binary message into an associative container of ConfigurableElement entries. |
| **Parameter** | `data` array of bytes, binary message from Configurable data stream |
| **Return** | an associative array of `[Integer, ConfigurableElement]` pairs |

`Preview(Array data)`

| | |
|---|---|
| **Summary** | Convert a binary message into an associative container of PreviewElement entries. |
| **Parameter** | `data` array of bytes, binary message from Preview data stream |
| **Return** | an associative array of `[Integer, PreviewElement]` pairs |

`Sensor(Array data)`

| | |
|---|---|
| **Summary** | Convert a binary message into an associative container of SensorElement entries. |
| **Parameter** | `data` array of bytes, binary message from Sensor data stream |
| **Return** | an associative array of `[Integer, SensorElement]` pairs |

```java
try {
  final String Host = "";
  final int Port = 32079;

  Client client = new Client(Host, Port);

  System.out.println("Connected to " + Host + ":" + Port);

  while (true) {
    if (client.waitForData()) {
      while (true) {
        ByteBuffer data = client.readData();
        if (null == data) {
          break;
        }

        Map<Integer,Format.PreviewElement> preview =
          Format.Preview(data);
        for (Map.Entry<Integer,Format.PreviewElement>
            entry: preview.entrySet()) {
          // Read the current orientation as a global
          // rotation matrix.
          float[] matrix_4x4 =
            entry.getValue().getMatrix(false);
        }
      }
    }
  }

} catch (Exception e) {
  System.err.println(e);
}
```

Example 3: Format class example usage (Java).

`Raw(Array data)`

| | |
|---|---|
| **Summary** | Convert a binary message into an associative container of RawElement entries. |
| **Parameter** | `data` array of bytes, binary message from Raw data stream |
| **Return** | an associative array of [`Integer, RawElement`] pairs |

## ConfigurableElement

`ConfigurableElement(Array data)`

| | |
|---|---|
| **Precondition** | Input data is not empty. |
| **Postcondition** | All accessors will return valid data. |
| **Parameter** | `data` array of real valued elements |

`value(Integer index)`

| | |
|---|---|
| **Summary** | Get the real valued element at an index. |
| **Precondition** | `index < size()` |
| **Parameter** | `index` access the element at this index |

`size()`

| | |
|---|---|
| **Summary** | Get the size of the element array. Use the size and the `value` accessors to iterate over the variable length array. |

## PreviewElement

`PreviewElement(Array data)`

| | |
|---|---|
| **Precondition** | Input data has 14 elements. |
| **Postcondition** | All accessors will return valid data. |
| **Parameter** | `data` array of real valued elements |

`getEuler()`

| | |
|---|---|
| **Summary** | Get a set of x, y, and z Euler angles that define the current orientation. Specified in radians assuming an `x-y-z` rotation order and a right handed coordinate system. Not necessarily continuous over time, each angle lies on the domain [`-pi, pi`]. Euler angles are computed on the server side based on the current local quaternion orientation. |

`getMatrix([Boolean local])`

    **Summary**    Get a right handed 4-by-4 rotation matrix from the current global or local quaternion orientation. Specified as a 16 element array in row-major order.

    **Parameter**    `local` set to `true` get the local orientation, set to `false` to get the global orientation

`getQuaternion([Boolean local])`

    **Summary**    Get the global or local unit quaternion that defines the current orientation. Specified as a 4 element array in `q = [w, x, y, z]` order where $q = w + x\hat{i} + y\hat{j} + z\hat{k}$.

    **Parameter**    `local` set to `true` get the local orientation, set to `false` to get the global orientation

`getAccelerate()`

    **Summary**    Get x, y, and z of the current estimate of linear acceleration. Specified in $g$.

**SensorElement**

`SensorElement(Array data)`

    **Precondition**    Input data has 9 elements.
    **Postcondition**    All accessors will return valid data.
    **Parameter**    `data` array of real valued elements

`getAccelerometer()`

    **Summary**    Get a set of x, y, and z values of the current unfiltered accelerometer signal. Specified in $g$ where $1\,g = 9.80665\ \mathrm{meter}/\mathrm{second}^2$.
                Domain varies with configuration. Maximum range is `[-6, 6]`$g$.

`getGyroscope()`

    **Summary**    Get a set of x, y, and z values of the current un-filtered gyroscope signal. Specified in $\mathrm{degree}/\mathrm{second}$.
                Valid domain of the sensor is `[-500, 500]` $\mathrm{degree}/\mathrm{second}$. Expect values outside of this domain as the system does not crop the sensor outputs.

**getMagnetometer()**

Summary    Get a set of x, y, and z values of the current un-filtered magnetometer signal. Specified in $\mu T$, (microtesla). Domain varies with local magnetic field strength. Expect values on domain `[-80, 80]` $\mu T$, (microtesla).

**RawElement**

**RawElement(Array data)**

Precondition    Input data has 9 elements.
Postcondition    All accessors will return valid data.
Parameter    `data` array of short integer valued elements

**getAccelerate()**

Summary    Get a set of x, y, and z values of the current raw accelerometer signal. Integer format.

**getGyroscope()**

Summary    Get a set of x, y, and z values of the current raw gyroscope signal. Integer format.

**getMagnetometer()**

Summary    Get a set of x, y, and z values of the current raw magnetometer signal. Integer format.

## 3.4   LuaConsole

The LuaConsole class provides an interface to send arbitrary Lua chunks to the Motion Service console. A client application can control all aspects of the Shadow Service through the Lua scripting interface.

For more information and function level documentation refer to the Motion **Scripting Reference** manual.

```
// using Motion.SDK;
// ...

try {
  String chunk =
    // 1. Close all connections.
    // 2. Set the G range for all configured devices
    // 3. Start reading
    "node.close() node.set_gselect(6) node.start()" +
    // Print the result of node.is_reading()
    " print('node.is_reading() == ', node.is_reading())"
    ;

  // Open connection to the Motion Service console
  // on the localhost, port 32075.
  Client client = new Client("", 32075);

  // Send the Lua chunk and receive the response.
  LuaConsole.ResultType result =
    LuaConsole.SendChunk(client, chunk);
  if (LuaConsole.ResultCode.Success == result.first)
  {
    // Print the output from Lua.
    Console.WriteLine(result.second);
  }
  else if (LuaConsole.ResultCode.Continue == result.first)
  {
    Console.WriteLine("incomplete Lua chunk: " + result.second);
  }
  else
  {
    Console.WriteLine("command failed: " + result.second);
  }
} catch (Exception e) {
  System.err.println(e);
}
```

Example 4: LuaConsole class example usage (C#).

```
SendChunk(Client client,
     String chunk,
     [Integer time_out_second])
```

| | |
|---|---|
| **Summary** | Send a Lua chunk, or string of commands, to a Motion Service console and return the results. |
| **Parameter** | `client` client connection to the Motion Service console |
| | `chunk` any valid Lua chunk |
| | `time_out_second` specifies a time out for this call, passed directly to Client::`readData` and Client::`writeData` |
| **Return** | a pair of return code and string results from the Lua chunk |

# 4 Supported Platforms

## 4.1 C++

**Windows**

The SDK is designed for use with Microsoft Visual Studio 2005 (MSVC8) or newer. We do not support older versions.

**Linux**

We tested the GCC compiler versions 3.4, 4.0, 4.1, and 4.2. Use the supplied Makefile to build the library and test files.

**macOS**

We tested the Apple GCC compiler version 4.0.1. Use the supplied Makefile to build the library and test files.

**Borland CodeGear C++**

We tested the SDK with the CodeGear C++Builder 2007. We do not support older versions.

## 4.2 C#

We tested the SDK with Microsoft Visual Studio 2005 (MSVC8). We do not support older versions.

We also support the Mono cross platform, open source .NET development framework. Use the supplied Makefile to build the library and test files.

## 4.3   Java

The SDK was developed on the Java SE 6 platform, with the 1.5 compiler. We do not support older versions.

## 4.4   Python

We tested the SDK with Python versions 2.7 and 3.7

# 5   Changes

The SDK stream format is compatible with all versions of the Motion Service. An older version of the SDK will work with newer software and vice versa.

As of SDK version 2.0, the namespace and module name in the SDK source code is `Motion`. Users upgrading from version 1.x will need to update references to the old namespace and module name `MotionNode`.