Essential SDK API

This describes the core parts of the API needed to create a minimal application. For a full list of functions and explanations, refer to SDK Complete Function Library.

Introduction

All the function calls regarding connect/disconnect and sending commands to the device are **asynchronous**. This means that as soon as a function call is made, the program will keep on going and the response will later end up in one of the queues to be read from. Only a boolean will be returned from the function call saying if everything went fine with queuing up the command or if something went wrong (check errno below).

The *ConnectionQueue* will handle the responses regarding connecting to the device. If something went wrong, a message is sent over the ConnectionQueue with the type *ConnectionFault*. The specific error code can be read from the *ErrorCode* field in the message. Read more under the Connect call description of the documentation for Connection.

Responses for function calls to the device, such as SetIdleFrequency, GetEnable or SetOperationMode are sent over the DeviceQueue. See example under SensorDevice.h.

For an explanation of errno usage, refer to SDK Complete Function Library.

Most of the structs have function pointers like Destructor etc which are not described in this section. Instead refer to SDK Complete Function Library.

Structures

zForce.h

This is the main structure for the zForce SDK and should always be instantiated at the beginning of execution. On shutdown it´s the last thing that should be uninitialized. Each call to zForce_Initialize() needs to be paired with exactly one call to zForce_Uninitialize(). It´s safe to call zForce_Initialize() after zForce_Uninitialize().

Function declaration	Info
zForce * zForce_GetInstance (void)	Returns a zForce instance, call Initialize first.
bool zForce_Initialize (OsAbstractionLayer * osAbstractionLayer)	Initializes the zForce SDK. If you want to use default OS Functions for memory allocation, mutexes, threads, etc, pass NULL as parameter, otherwise supply a structure.
void zForce_Uninitialize (void)	Uninitialize a zForce instance, freeing all allocated resources. No part of the zForce SDK should be used after this call.

Example

Example for initialize zForce module with default system operations.

```
zForce_Initialize (NULL);
```

Example for uninitialize zForce instance.

```
zForce_Uninitialize ();
```

Connection.h

Handles connections, and binds the devices, protocols and transports together. Also handles two queues, one to notify the software about Connection changes (Connect/Disconnect/Connection errors) and one to send any incoming messages from the device.

Function declaration	Info

Connection * Connection_New (char * connectionString, char * protocolString, char * dataFrameType)	This call creates the binding between the Protocol and the Transport.
bool (* Connect) (Connection * self)	Connect to the unit using the previously specified Protocol and Transport done by Connection_New. This call asynchronously starts the connection process. The call exits immediately and if there is a direct connection problem, for example that the connection is already established, then the call exits with false and errno is set. If there is no immediate error, the call returns true but the connection cannot be considered established until the ConnectionQueue has returned with a message saying status Connected. If connection failed with a status of ConnectionFault, ErrorCode will be set. See documentation under Connection.
bool (* Disconnect) (Connection * self)	Disconnect the connection. This call is asynchronous and cannot be considered complete until the ConnectionQueue has reported status Disconnected.
void (* Destructor) (Connection * self)	Freeing up the memory allocated in Connection_New.
Device * (* FindDevice) (Connection * self, DeviceType deviceType, uint32_t deviceIndex)	Find a Device with a specified Type/Index combination. For Platform and Lighting devices, the Type/Index are searched for directly. For Core, Air and Plus devices, the Type/Index are also searched for directly, but they can also be found using a meta Sensor device type. Sensor is not a real device type, and contains all functionality that is shared between all Sensor types (Core, Air and Plus). Type = Sensor, Index = 0 is the first Sensor in the list, Type = Sensor, Index = 1 is the second Sensor. The "real" De vice Index is not used for this search when searching for a Sensor.

Queues	Info
Queue * Connection Queue	Queue used to receive connection response from device. Response will come after using Connect above. Check Queue in Documentations to see function calls and info.
Queue * DeviceQueue	Queue used to receive Message responses from device. This will be the queue that will hold all responses for the commands.

Examples

Example of setting up a connect to a USB HID device connected on Linux device node using ASN.1 and HidPipeTransport where vid is vendor id and pid is product id. If the computer has multiple connected Neonode sensors, the first one has index 0, the second has index 1, etc. The order is decided by the Operating System.

```
MyConnection = Connection_New ("hidpipe://vid=0x1536,pid=0x0101,index=0", "asn1://", "Streaming");
```

Example of doing the actual connect to the unit using the previously specified Protocol and Transport done by Connection_New.

```
bool connectionAttemptResult = MyConnection->Connect (MyConnection);
```

Example of setting up a connection, connecting, disconnecting and freeing up the memory.

```
MyConnection = Connection_New ("hidpipe://vid=0x1536,pid=0x0101,index=0", "asn1://", "Streaming");
bool connectionAttemptResult = MyConnection->Connect (MyConnection);
MyConnection->Disconnect (MyConnection);
MyConnection->Destructor (MyConnection);
```

This is an example on FindDevice for both sensor device and platform device.

```
PlatformDevice * platformDevice = (PlatformDevice *)MyConnection->FindDevice (MyConnection, Platform, 0);
// Find the first Sensor type device (Core/Air/Plus).
SensorDevice * sensorDevice = (SensorDevice *)MyConnection->FindDevice (MyConnection, Sensor, 0);
```

Example of ConnectionQueue used after issuing a Connect call. The number 1000000 is the number of milliseconds(1000 seconds) we will wait for a response.

```
ConnectionMessage * connectionMessage = MyConnection->ConnectionQueue->Dequeue (MyConnection->ConnectionQueue,
1000000);
```

Example of waiting for an answer to arrive on DeviceQueue with a timeout of 1 second.

```
Message * message = MyConnection->DeviceQueue->Dequeue (MyConnection->DeviceQueue,
1000);
```

Device.h

Device.h is a base class for SensorDevice and PlatformDevice. The calls are asynchronously and will always arrive through DeviceQueue.

SensorDevice.h

The SensorDevice.h class holds most of the zForce commands in the zForce SDK that is currently available with any sensor device type (Core/Air/Plus). Each command has a Get which when sent to the device will retrieve the current state of that command from the device. The class also contains a set for each command, that will set given values in the sensor device. A Set command will also return current status after the set command has executed, same as Get would.

Only one command can be sent at a time, which means that a command has to be sent and a response then read before another command can be sent.

Function declaration	Info
bool (* GetEnable)(SensorDevice * self)	Ask the Device if it is enabled. Also reports if the device is in continuous mode and number of messages left to receive
bool (* SetEnable)(SensorDevice * self, bool continuousMode, uint32_t numberOfMessages)	Enable the device and set it to either continuous mode or a specified number of messages. Not all protocols handle non-continuous mode
bool (* GetDisable)(SensorDevice * self)	Returns true if the device is disabled.
bool (* SetDisable)(SensorDevice * self)	Disable the device. Messages will stop arriving as soon as the Device pro cesses the request, but Messages already in progress will still arrive as normal.
bool (* GetOperationModes)(SensorDevice * self)	Gets the current Operating Modes of the Device as two bitmasks.
bool (* SetOperationModes)(SensorDevice * self, OperationModes modeMask, OperationModes modeValues)	Sets the current Operating Modes of the Device as two bitmasks. Setting the modeMask bit for a specific Mode to 1 changes it to become the corresponding bit in modeValues but setting the bit to 0 does not change it.
bool (* GetResolution)(SensorDevice * self)	Gets the current Resolution of the Device.
bool (* SetResolution)(SensorDevice * self, uint32_t x, bool xIsValid, uint32_t y, bool yIsValid, uint32_t z, bool zIsValid)	Sets the current Resolution of the Device.
bool (* GetTouchActiveArea)(SensorDevice * self)	Gets the current Touch Active Area of the Device.
bool (* SetTouchActiveArea)(SensorDevice * self, uint32_t lowerBoundaryX, uint32_t upperBoundaryX, bool xIsValid, bool xIsReversed, uint32_t lowerBoundaryY, uint32_t upperBoundaryY, bool yIsValid, bool yIsReversed)	Sets the current Touch Active Area of the Device.
bool (* GetDetectedObjectSizeRestriction)(SensorDevice * self)	Gets the current Detected Object Size Restriction.
bool (* SetDetectedObjectSizeRestriction)(SensorDevice * self, uint32_t minimumSize, uint32_t maximumSize, bool minimumSizeIsValid, bool maximumSizeIsValid)	Sets the current Detected Object Size Restriction of the device.
bool (* GetNumberOfTrackedObjects)(SensorDevice * self)	Get the number of tracked objects.
bool (* SetNumberOfTrackedObjects)(SensorDevice * self, uint32_t numberOfTrackedObjects)	Set the number of tracked objects.

Example

Example of sending GetResolution command to sensor device following by receiving response and destroying message. One for GetResolution and one for SetEnable.

```
// Send the command GetResolution to device.
bool result = sensorDevice->GetResolution (sensorDevice);
// Wait for the answer to arrive, timeout after 1 second.
Message * message = MyConnection->DeviceQueue->Dequeue (MyConnection->DeviceQueue, 1000);
// Freeing up the memory allocated for the message.
message->Destructor (message);
// After reading the response we are ready to send a new command.
result = sensorDevice->SetEnable (sensorDevice, true, 0);
// Wait for the answer to arrive, timeout after 1 second.
message = MyConnection->DeviceQueue->Dequeue (MyConnection->DeviceQueue, 1000);
// Freeing up the memory allocated for the message.
message->Destructor (message);
```

PlatformDevice.h

Holds the commands to communicate with the platform device.

Function declaration	Info
bool(* GetFingerFrequency)(PlatformDevice * self)	Get the current finger frequency.
bool(* SetFingerFrequency)(PlatformDevice * self, uint32_t frequency)	Set the finger frequency.
bool(* GetIdleFrequency)(PlatformDevice * self)	Get the current idle frequency.
bool(* SetIdleFrequency)(PlatformDevice * self, uint32_t frequency)	Set the idle frequency.

Example

Example of sending GetFingerFrequency command to platform device following by receiving response and destroying message. One for GetFingerFrequency and one for SetFingerFrequency .

```
bool result = platformDevice->GetFingerFrequency(platformDevice);
// Wait for the answer to arrive, timeout after 1 second.
Message * message = MyConnection->DeviceQueue->Dequeue (MyConnection->DeviceQueue,1000);
// Freeing up the memory allocated for the message.
message->Destructor (message);

result = platformDevice->SetFingerFrequency(platformDevice, 10);
// Wait for the answer to arrive, timeout after 1 second.
Message * message = MyConnection->DeviceQueue->Dequeue (MyConnection->DeviceQueue,1000);
// Freeing up the memory allocated for the message.
message->Destructor (message);
```

Read More

- SDK System Requirements
- Getting started with SDK for Linux
- Getting started with SDK for Windows
- Example Program Pseudocode
- Essential SDK API
- SDK Complete Function Library
- SDK Function Support Matrix
- Release Notes
- Legal Notice