SAFR® Jetson SDK Documentation

Documentation Version = 3.030

Publish Date = October 6, 2021

Contents

1 SAFR SDK Overview

1.1 Purpose

The SAFR SDK is a set of shared libraries that together implement an object tracking mechanism, face recognition and event reporting. The object tracker is used to locate and track different types of objects (for example, human faces and badges) in a video file or live video stream. Face recognition can learn new identities or match existing identities. The event reporter can notify your application of detection and recognition events. Object detection, face recognition, tracking and event reporting are performed in real time.

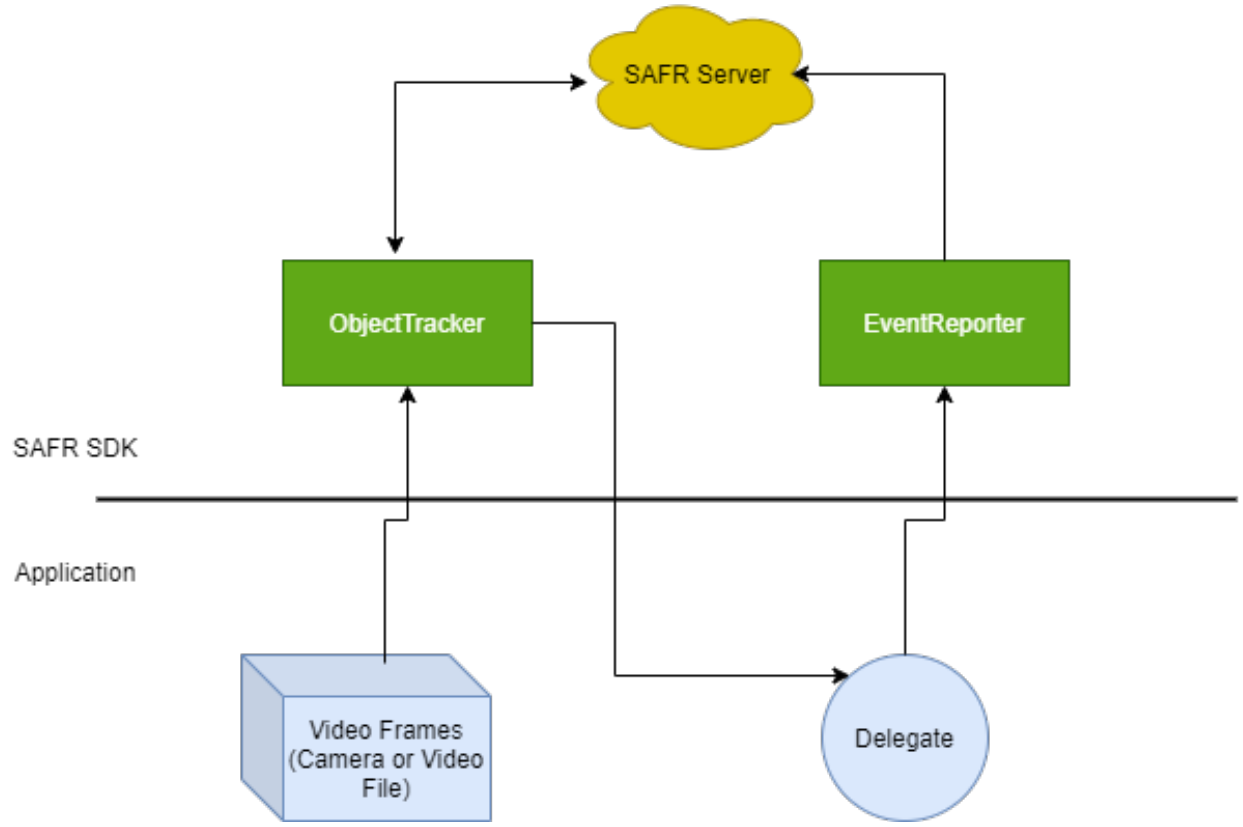
The SDK features a C API that exposes a set of ArgusKit objects. All SDK-defined functions and type definitions are prefixed with the `ARK` namespace qualifier. The SDK is distributed as a `tar.gz` file containing all necessary headers, documentation, and shared libraries. It also includes samples showing how the SDK can be used from your own application, and how the SDK components (shared libraries and auxiliary files) must be packaged with your application in order to ensure ArgusKit works correctly with your application.

While the SDK API design and architecture is the same across all supported platforms, the implementation language of the API may be different. For example, the API may be a C API for some platforms, while it is a Java or C# API for other platforms. Also, the packaging of the SDK is different for each platform to accommodate the technical requirements and limitations of the platform.

1.2 SDK Organization

The SAFR SDK defines a number of objects that an application uses to track faces or badges. The application creates these objects, configures them, and then interacts with them. Once the application no longer needs a SAFR SDK object, it is expected to release (*free*) the SAFR SDK objects. All SAFR SDK objects use reference counting to manage the lifetime of the objects.

The following diagram shows the two most important SAFR SDK objects: the `ObjectTracker` and the `EventReporter`, and how an application interacts with them:



The application receives video frames from a camera or a video file and hands them off to an ObjectTracker. The ObjectTracker processes the video frames internally to detect and track objects. The ObjectTacker sends the video frames to a SAFR Server (whether in the cloud or installed on-premise) for recognition. The ObjectTracker continuously informs the application about the current state of the ObjectTracker by invoking the ObjectTracker delegate. This delegate is provided by the application and receives information about objects found in the video input stream, objects that have disappeared, and objects that have changed their state. For example, an object may have been detected in one video frame, and it may have been recognized as a specific person in a later video frame. The ObjectTracker informs the application about this change of state.

An application is free to use the information about a tracked object any way it likes. It could display this information in a video overlay in its UI or it could process the information to generate statistical information or events. The SAFR SDK comes with an EventReporter able to process a stream of tracked objects to detect certain kinds of events that are then automatically reported to a SAFR Server.

Note: For SAFR SDK system requirements, go to SAFR.com.

2 Jetson SDK Installation

The SAFR SDK for Jetson is distributed as a set of shared object (`.so`) and resource files. It is recommended you copy these files into the directory that contains your final application. Place the shared object files inside a directory named `lib` and the resource files inside a directory named `model`. As an example, if your application is called *Unique* and you ship it inside a directory called `UniqueApp`, then the structure of your final application directory would look like this:

```
UniqueApp/
   Unique
   lib/
      ...
   model/
      ...
```

Note: The application must be built as a "64bit" application. "32bit" applications are not supported.

3 Use the Demo App for Jetson

The SAFR Jetson SDK ships with an example application demonstrating how to set up an object tracker, feed it with video frames from a video file, and receive information about detected and recognized persons from the object tracker.

Note: The demo application requires an active SAFR Cloud account in order to function. The demo source does not compile as long as you have not updated it with the SAFR credentials you received. Search for `$USER` and `$PWD` in the source and replace them with your user name and password.

4 Use the Jetson SDK

The C API defines a set of pseudo-objects which the SDK makes available for use by an application. Each object is represented in C by a distinct object type and a set of functions sharing a common function name prefix. The function name prefix reflects the type of object on which the function can be called.

Memory is managed via reference counting. Use the **ARKObjectRetain()** API to increase the reference count and use **ARKObjectRelease()** to decrease the reference count. Once the reference count of an object reaches zero, the object and all the resources managed by the object are freed. Call **ARKObjectRetain()** if you want to keep an object alive and **ARKObjectRelease()** if you no longer care about an object.

5 Recognize and Track Faces

Use an instance of the **ObjectTracker** class to detect, recognize, and track faces in a video stream.

The video stream may originate from a camera or a video file. The object tracker allows you to either process the video stream in real time or as fast as possible. The former mode would be typically used for a video stream that originates from a camera, while the later mode can be used to process the content of a video file at a speed faster than the normal playback speed of the video file.

5.1 Initialize the ArgusKit Framework

You must initialize the ArgusKit framework before you call any of its APIs. You can do this by calling the **ARKInit()** function at the beginning of your application.

Note: You need to call this function only once per application session.

5.2 Create the Video Player

The video player is used to playback video from a file or stream. The video frames are then passed into the object tracker for detection and recognition. The video player is created with a URL, and various callbacks are initialized to obtain information about the video state. The most important callback for tracking is the **didCreateVideoFrame** which is called when a new video frame is available that can be passed to the object tracker.

```
// Create the video player
ARKVideoPlayerCallbacks callbacks0;
callbacks0.context = NULL;
callbacks0.didCompletePreroll = video_player_preroll_completed_callback;
callbacks0.didChangePauseState = video_player_pause_changed_callback;
callbacks0.didCreateVideoFrame = video_player_frame_callback;
callbacks0.didEncounterError = video_player_error_callback;
callbacks0.didReachEndOfStream = video_player_end_of_stream_callback;

gPlayerRef = ARKVideoPlayerCreate(url, &callbacks0);
```

5.3 Create the Object Tracker

The first step is to create an object tracker instance. You do this by creating an instance of an object tracker configuration object that stores all the configuration information an object tracker needs to do its work. Most of the configuration information has sensible default values, but the following information must be explicitly provided:

- The cloud environment.
- Login information that should be used for the cloud-based face recognizer.

The following code snippet shows how to set up the object tracker configuration and how to initiate the object tracker:

```
// Select the appropriate cloud environment. Eg. PROD
const char* environName = get_feature_string_value(argc, argv,
   "--environment");
char environBuffer[16];
strcpy(environBuffer, "com.real.");
strcat(environBuffer, (environName) ? environName : "PROD");
ARKEnvironmentRef envRef = ARKEnvironmentCopyNamed(environBuffer);

// Specify the cloud environment log-in credentials
```

```
ARKUserRef userRef = NULL;

const char* userName = get_feature_string_value(argc, argv, "--user");
const char* userPassword = get_feature_string_value(argc, argv,
   "--password");
const char* directory = get_feature_string_value(argc, argv,
   "--directory");
if (userName && userPassword) {
    userRef = ARKUserCreate(userName, userPassword);

    // Set the user directory in which the cloud should store its data
    ARKUserSetDirectory(userRef, (directory) ? directory : "test");
}

// Create the object tracker configuration
ARKObjectTrackerConfigurationRef configRef =
   ARKObjectTrackerConfigurationCreate();
ARKObjectTrackerConfigurationSetObject(configRef,
   kARKObjectTrackerConfigurationKey_Environment, envRef);
ARKObjectTrackerConfigurationSetObject(configRef,
   kARKObjectTrackerConfigurationKey_User, userRef);
ARKObjectTrackerConfigurationSetString(configRef,
   kARKObjectTrackerConfigurationKey_SiteID, "Building 1");
ARKObjectTrackerConfigurationSetString(configRef,
   kARKObjectTrackerConfigurationKey_SourceID, "Camera 1");


// Create the object tracker
ARKObjectTrackerCallbacks callbacks1;
callbacks1.context = NULL;
callbacks1.willBeginTracking =
   object_tracker_will_begin_tracking_callback;
callbacks1.didEndTracking = object_tracker_did_end_tracking_callback;
callbacks1.didCreateTrackingResult =
   object_tracker_did_create_tracking_result_callback;

gTrackerRef = ARKObjectTrackerCreate(configRef, false, &callbacks1);
ARKObjectRelease(configRef);
configRef = NULL;
```

This example code selects the desired cloud environment and creates a new user object with the required user identifier and password. It also sets the cloud directory where the cloud-based face recognizer should save recognition-related information.

It then creates an object tracker configuration object and sets the cloud environment, cloud user, and some additional information to help identify the camera. After that, it sets up the necessary callbacks the object tracker should invoke as it processes the video stream. Finally, the example code creates the actual object tracker object.

Note: The example code assumes that the input video stream originated from a camera. This is why we pass `true` to the real-time parameter of the **ARKObjectTrackerCreate()** function.

5.4 Start a Tracking Session

All tracking-related activities are done in the context of a *tracking session*. The object tracker uses a tracking session to maintain the necessary state. The following code snippet shows you how you can start a new tracking session:

```
ARKObjectTrackerBeginTracking(trackerRef);
```

Note: Always end the current tracking session and start a new session if the video source or the resolution or frame rate of the video stream has changed. For example, end the current tracking session and start a new one if the user switched cameras or selected a different capture profile.

The object tracker invokes the begin-tracking callback at the start of a new tracking session. Your application can use this callback as a signal that a new tracking session has started. The following code snippet shows an example of such a callback:

```
static void
    object_tracker_will_begin_tracking_callback(ARKObjectTrackerRef
    _Nonnull trackerRef, void* _Nullable context)
{
     printf("willBeginTracking\n");
}
```

5.5 Run a Tracking Session

A new video frame object should be created for every decoded video frame, and this video frame object should then be passed to the object tracker. The object tracker in turn runs a face detector on the video frame and it triggers face recognitions as needed. The object tracker then updates its internal list of tracked object with the result of detectors and recognizers.

The object tracker invokes the application-provided callback with the current state of the tracked object list. The application can inspect this list and trigger actions based on it.

Note: The application should create a copy of the tracked object list if it wants to hold on to the data. (e.g. if the application wants to process the tracked objects on a different thread)

The following code snippet shows how to create a video frame object and how to pass it to the object tracker:

```
static void video_player_frame_callback(ARKVideoPlayerRef _Nonnull
    playerRef, ARKVideoFrameRef _Nonnull frameRef, void* _Nullable context)
{
     ARKObjectTrackerTrackObjects(gTrackerRef, frameRef);
     ARKObjectRelease(frameRef);
}
```

You must provide a timestamp when you create a video frame object. This is typically the presentation timestamp of the video frame. The **isSceneChange** parameter of the **ARKVideoFrameCreateWithPixelBuffer()** function should be set to `true` if the video frame is the first video frame after a scene change. Scene changes in movies are often indicated by a cut transition from one scene to another. The object tracker uses this information to enhance its ability to disambiguate between persons in different scenes.

Note: Although a video stream from a camera includes key frames, those key frames do not indicate a scene change for tracking purposes. For that reason, those key frames should not be treated as a scene change.

The following code snippet shows an example implementation of the did-track that simply prints a description of the new tracking results callback:

```
static void
   object_tracker_did_create_tracking_result_callback(ARKObjectTrackerRef
   _Nonnull trackerRef, ARKTrackingResultRef _Nonnull resultRef, void*
   _Nullable context)
{
    ARKObjectPrintDebugDescription(resultRef)
}
```

The object tracker invokes the end-tracking callback at the end of the tracking session. Your application code can use this callback as a signal that a tracking session has ended. The following code snippet shows an example implementation of such a callback:

```
static void object_tracker_did_end_tracking_callback(ARKObjectTrackerRef
   _Nonnull trackerRef, void* _Nullable context)
{
    printf("didEndTracking\n");
}
```

5.6 End a Tracking Session

You inform the object tracker about the end of a tracking session by invoking the **ARKObjectTrackerEndTracking()** function. This allows the object tracker to clean up its internal state and lets it know it should execute pending callbacks as soon as possible.

The following code snippet shows how to end a tracking session:

```
ARKObjectTrackerEndTracking(trackerRef);
```

5.7 Detecting Persons

Once you setup and run a basic object tracker configuration object, enable the detection of persons by setting the **kARKObjectTrackerConfigurationKey_ShapeDetector_Enable** property in the object tracker's detector configuration object to `true`. This causes the object tracker to return tracked objects with a detected object type equal to **kARKObjectType_Shape**.

6 Utility and Video Processing Objects

6.1 ARKObject

This is the base class of all ArgusKit objects. The C functions provided by **ARKObject** can be used with all ArgusKit object types. **ARKObject** provides APIs for memory management and to generate a debug description of an object that is printed to the console.

6.2 ARKEnvironment

The **ARKEnvironment** object encapsulates the cloud service to which the ArgusKit should connect to execute cloud-based functions like face recognition.

6.3 ARKObjectTrackerConfiguration

An object tracker configuration object stores the configuration information for an object tracker, the object detector, and the face recognizer. The configuration object is initialized with default values allowing the tracker to detect and recognize faces. You may also instantiate a configuration object based on one of a number of predefined configuration presets. A preset encapsulates all the configuration information needed to use the object tracker for a specific type of task, for example, to recognize faces or to recognize and learn faces.

After you have instantiated a configuration object, set the cloud account, cloud environment and the face recognizer directory name. This is the minimum required information that you need to provide to allow the object tracker to successfully detect and recognize faces.

If you want to detect faces only without recognizing them, you can turn off the face recognizer stage by setting the **Recognizer.Enabled** property to `false`.

6.4 ARKObjectTracker

The object tracker is the heart of ArgusKit. It receives a stream of video frames that it analyzes to find objects. It tracks found objects as long as they remain visible in the video stream. Object detection, recognition and the tracking are executed in real time. An object tracker is connected to a delegate defined by a set of C callback functions you pass to the object tracker at creation time. The object tracker informs its delegate at every frame about the current state of the objects it is tracking. The delegate can then use the tracked object APIs to learn what kind of objects the tracker found and what their current spatial location, size, and state are.

6.5 ARKTrackingResult

A tracking result object contains a snapshot of the current state of the object tracker. The object tracker delivers a new tracking result at every frame boundary. The tracking result contains a list of tracked object which have appeared in the current frame, which have disappeared and which have changed their current state in the current frame. For example, if a tracked object was detected in a previous frame and the object tracker has now been able to recognize the tracked object as a specific identity, then the tracked object is included in the list of updated tracked objects.

iOS: In the sample application, this is wrapped by the Swift object **TrackingResult**. The **CameraViewController.swift** file contains a function called **handleTrackingResult** that illustrates how the tracking result can be used to get the bounding box and other properties for the face.

6.6 ARKTrackedObject

A tracked object represents a single and unique instance of an object the object tracker has been able to detect in the inout video stream and is actively tracking. A tracked object has a type indicating whether it is a badge or the face of a person. A tracking object also has an axis-aligned bounding box informing you where

in the input video frame the tracked object can be found and what its size is. This bounding box can be used with the original frame passed to the object tracker to extract the thumbnail image from the video frame.

6.7 ARKTrackedBadge

A tracked badge is a tracked object that encodes an integer number in the form of a special pattern.

6.8 ARKTrackedFace

A tracked face represents a human face and may be linked to a person. If the object tracker is able to recognize the face as belonging to a specific person, the tracked face provides a reference to the corresponding person object. The tracked face also contains other detected attributes about the face if they are enabled such as center pose quality, yaw, roll, and pitch.

6.9 ARKPerson

A person object provides information about a person who has been registered with the ArgusKit face recognition service. Each person has a unique identifier. You may assign a name and a set of tags to a person with the help of a **ARKPersonChange** object and the **ARKObjectTrackerApplyPersonChange()** object tracker function.

6.10 ARKPersonChange

A person change object stores attributes applied to the person record on file in the ArgusKit face recognition service. This object allows you to change a person's name, tags, age, or gender information.

6.11 ARKVideoPlayer

A **VideoPlayer** allows you to play back a video file or a HTTP/RTSP video stream.

Note: The video player supports video streams only; it does not support audio streams.

Create an instance of a video player and register an event handler with the **DidDecode** event property to receive the decoded video frames. Next, start playback by setting the **Paused** property to `false`. You can pause playback at any time by setting the **Paused** property to `true`.

You can stop playback altogether in preparation of disposing of the video player by calling **Stop()**. You can stop playback and dispose of the player at the same time by calling **Dispose()**.

The video player enforces the video clock if the URL points to a video file, but it does not enforce the video clock if the URL points to a HTTP or RTSP video stream. In this case, the video player decodes video frames as fast as they arrive from the camera. It does this to minimize the video playback latency.

6.12 ARKVideoFrameCache

A video frame cache is a pool of cached video frames allowing you to efficiently create a new video frame.

6.13 ARKVideoFrame

A video frame object encapsulates a single decoded video frame that is passed to an object tracker instance.

7 Image Procesing Objects

These objects allow you to detect and recognized objects in an individual image.

7.1 ARKImage

You create image instances to represent an image that you want to hand off to the image analyzer. An image may be created form a JPEG file or an in-memory buffer of RGBA or BGRA 32bit pixels.

7.2 ARKImageAnalyzerConfiguration

An image analyzer configuration object stores the configuration information for an image analyzer, the object detector, and the face recognizer. The configuration object is initialized with default values allowing the image analyzer to detect and recognize faces. You may also instantiate a configuration object based on one of a number of predefined configuration presets. A preset encapsulates all the configuration information needed to use the image analyzer for a specific type of task, for example, to recognize faces or to recognize and learn faces.

After you have instantiated a configuration object, set the cloud account, cloud environment, and the face recognizer directory name. This is the minimum required information you need to provide to allow the object tracker to successfully detect and recognize faces.

If you want to detect faces only without recognizing them, turn off the face recognizer stage by setting the **Recognizer.Enabled** property to `false`.

7.3 ARKImageAnalyzer

The image analyzer can process an image/photo that it analyzes to find objects. Object detection and recognition are executed in real time. An image analyzer is connected to a delegate defined by a set of C callback functions passed to the image analyzer at creation time. The image analyzer informs its delegate when analysis is completed. The delegate can then use the image analyzer APIs to learn what kind of objects the image analyzer found and what their current spatial location, size, and state are.

7.4 ARKImageAnalysisResult

An image analysis result object contains a snapshot of the what was detected in an image. The image analyzer result contains a list of objects that have appeared in the current image.

7.5 ARKImageAnalyzedObject

An analyzed object represents a single and unique instance of an object the image analyzer has been able to detect in the image. An analyzed object has an axis-aligned bounding box that tells you where in the input video frame the tracked object can be found and what its size is.

7.6 ARKImageAnalyzedFace

An analyzed face represents a human face. An analyzed object may be linked to a person. If the image analyzer is able to recognize the face as belonging to a specific person, the tracked face provides a reference to the corresponding person object.

7.7 ARKImageAnalyzedBadge

An analyzed badge represents a Rhino tags-style badge.

7.8 ARKImageAnalyzedShape

An analyzed shape represents an object such as a car or a person.