# AUREUS 3D SDK

User Documentation

# 1 CyberExtruder's Aureus 3D SDK

Formed in the spring of 1999, CyberExtruder is the oldest US company in the face recognition arena and one of the very few in the world which has written its own algorithms for face matching. CyberExtruder was conceived to solve some of the hardest computer vision challenges and as our company has evolved and our focus has sharpened, our mission has become quite distinct. CyberExtruder intends to be the world's leading provider of face recognition products needed to protect the privacy and security of individuals and businesses.

The Aureus 3D SDK is the result of CyberExtruder's continuing program of research and development in advanced facial recognition (FR) technologies. We believe we have developed the world's fastest and most accurate face matching engine. Our performance superiority is based on our proprietary 2D to 3D face modeling technology which mitigates the challenges of pose, lighting and expression as well as our use of convolutional neural networks which allow us to train our algorithms to deliver ever increasing performance.

But it's not just about being the best at matching faces, our approach optimizes the competitive difference of 3D over 2D technology and delivers real-time face matching is full frame rate video. Our 3D technology compensates for pose, lighting, facial expressions, and vague images – which are the biggest constraints to conventional FR. Our 3D technology works with any video camera and we provide the most accurate face tracking technology anywhere. Best of all, our software is designed for easy partner integration.

We are humbled that you have chosen our solution as the heart of your own application and we're committed to improving your competitive advantage and bottom line profitability.

CyberExtruder
211 Warren Street
Newark, New Jersey, 07013 USA
T +1 973 623 7900
F +1 973 623 8900
Last updated June 2, 2016

*The main body of this document is a series of brief step-by-step descriptions of a number of common operations one may need to accomplish with Aureus. Note that Aureus is modular, with state machine like properties, with automated features continually running, leaving the end-developer's main task to be 1) understanding Aureus, 2) configuring Aureus for the functionality desired, and 3) run time direction, collection of data, and end-purpose operations of whatever application Aureus has been incorporated.*

## CONTENTS (THE TOPICS MARKED WITH A * ARE CONSIDERED REQUIRED READING BEFORE WRITING CODE)

# 2 A NOTE ON OUR USAGE OF TERMS

**Biometric:** Refers to the identification of humans by their inherent physical characteristics. It should be noted that in the context of security no single biometric will meet all the requirements of every possible application.

**Enrollment:** Is the process by which an individual supplies a biometric sample (in our case a facial image) to a system which then stores the information for later comparison.

**Facial Features**: Aureus identifies 7 key facial features on each detected human head. These facial features are identified with a series of connected points, which we call facial annotations. The identified facial features are:

1. Face outline
2. Eyebrows
3. Left eye
4. Right eye
5. Nose
6. Outer lips
7. Inner lips

**Facial Recognition**: This term refers to the comparison of one facial image to another with a resulting confidence score which indicates how similar the two faces are, i.e. the more similar the faces, the higher the score and therefore the greater the probability the two images represent the same person.

**Frame Callback**: when processing video streams, Aureus passes control to end-developer logic through a function installed into a CX_VIDEO object or a CX_ENROLL_VIDEO object. This end-developer function installed into these objects is the Frame Callback, so named because Aureus calls it each video frame. It is how end-developer software receives the video frames to display within their application (should they wish to display). As the video stream is consumed, the installed Frame Callback function is called frame after frame, enabling end-developer logic to display the frames, and in the case of a CX_VIDEO object, respond to information contained in the CX_HEAD_LISTs of detected, tracked and processed heads.

**Gallery**: Aureus creates and maintains an SQLite database containing image information about people. A known person consists of a numeric identifier, a name string, and at least one facial image. Additionally, Aureus will generate and maintain additional information describing a person, such as additional facial images, statistical and geometric information describing the facial features of each image, additional end-use supplied text information about each image, as well as facial recognition templates generated by Aureus from one or multiple supplied images.

**Head Pose:** Discussed in angular degrees with the concept of 'pose' being the sum of pitch roll and yaw. 'Pitch' is the nodding up and down motion; 'Roll' is the cheek to shoulder motion, 'Yaw' is the side to side (turning) motion

**Identification:** When a system performs a one-to-many comparison against a biometric database in attempt to establish the identity of an unknown individual.

**Match Score**: This is when comparisons take place between templates, and the result is summarized in a value called the match score. The value is normalized between 0.0 and 1.0. The lower end of the scale is indicative of little to no similarities between faces while higher values point to the images as belonging to the same person.

**Pose Correction**: Aureus identifies when a detected human head is not directly facing the camera, and via CyberExtruder's patented algorithms, performs a 3D Reconstruction of the human head, which is rotated in 3D to create a tokenized image of the human head as if it were directly facing the camera, with the camera positioned at eye level. Multiple human head images may be combined during a pose correction. When combining multiple facial images, perspective distortion correction is also applied, removing lens distortion to create more accurate tokenized image.

**ROC Curves**: Receiver Operating Characteristic (ROC) curves are the graphical plot of how a biometric system performs at various operating settings and are the standard framework by which a biometric system's performance can be judged. It is important to remember that when two facial images are compared to one another there are four possible outcomes – not just right and wrong. Those four possibilities are True Positive (TP - Correctly affirms identity), True Negative (TN– Correctly indicates the person is unknown to the system), False Positive (FP – Incorrectly identifies a person as a different person) and False Negative (FN– Incorrectly indicates a person is unknown to the system) responses. These four outcomes can be distilled in to 'right' and 'wrong' and graphically represented. **TP and TN responses are good** in that they are 'correct' determinations. **FP and FN responses are bad** in that they are 'incorrect.'

**SQLite database**: is an embedded relational database management system. In contrast to many other database management systems, SQLite is not a client/server database engine. Rather, its logic is embedded directly into Aureus. SQLite is a popular choice as embedded database software for local/client storage in application software. It is arguably the most widely deployed database engine, as it is used today by several widespread web browsers, operating systems, and embedded systems. SQLite has bindings to many programming languages.

**Tokenized Image**: A tokenized image is a facial image cropped to the head, and conforming to the ISO/IEC 19794-5 international standard for facial image data and biometric data interchange formats. This means all tokenized images have the face and eyes in consistent locations, as is necessary for high speed comparison for facial recognition. A passport photo is an example of a tokenized image.

**Template**: A template is the compiled information generated from analysis of one or more images and associated data such as their facial annotations. Ultimately, a template is just a cx_byte array. The template's cx_byte array can be considered compiled data, only meaningful to the FR engines loaded by Aureus. A template is the specific data used to perform the high speed comparisons that is facial recognition.

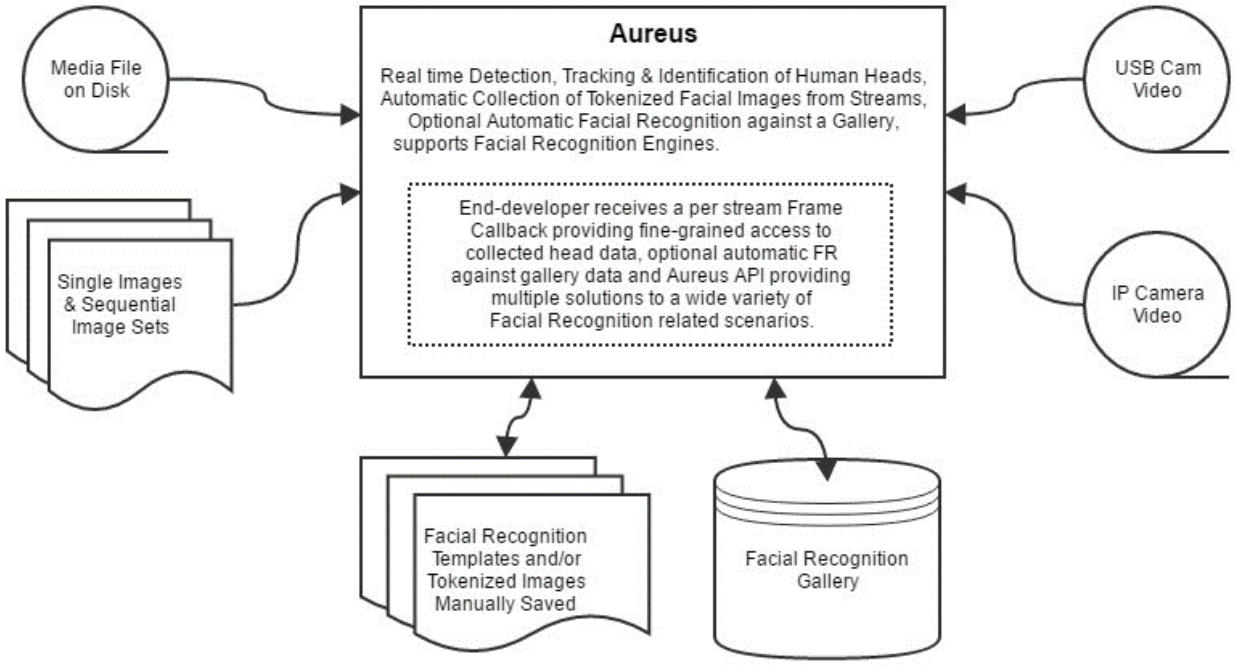
**Verification:** When a system performs a one-to-one comparison of a captured biometric with a specific template stored in a biometric database.

**Video Stream**: video originating from a media file, such as a MPEG or AVI file, video originating from a USB camera, or video originating from an IP camera.

**Unexpected Stream Termination Callback:** when processing video streams, unexpected stream termination can be caused by power loss to one's camera, someone unplugging the camera, or file corruption within the file on disk that is a media file. When this occurs, Aureus passes control to end-developer logic by calling this callback function. Such callback functions may be installed to a CX_VIDEO or CX_ENROLL_VIDEO object for notification of this event. End-developer logic is expected to Stop the active Video or Enroll Video object's playing video stream, and perform any necessary cleanup and end-user notification of video stream loss.

# 3 SDK OVERVIEW

The Aureus SDK provides everything you need to incorporate real time detection, tracking and identification of people (human heads), across multiple simultaneous video streams and sequential image sets into your own applications. Aureus allows you to provide 1:1 matching of faces, matching an image against a gallery with one or more facial recognition engines, as well as the underlying video stream load, configure, play, pause and stop features.




As video and sequential image sets are streamed through Aureus, Aureus is detecting heads, tracking them across frames. Additionally, each frame, Aureus maintains a list of quality sorted portrait format captures of the tracked heads in the stream, and, if requested, facial recognition gallery matches against the tracked heads.

Enrolling new heads into the facial recognition gallery is supported with multiple methods, each of which triggers Aureus to generate facial recognition templates from single or multiple unconstrained images. Likewise, facial recognition templates may be generated directly, or generated automatically and stored in a facial recognition gallery.

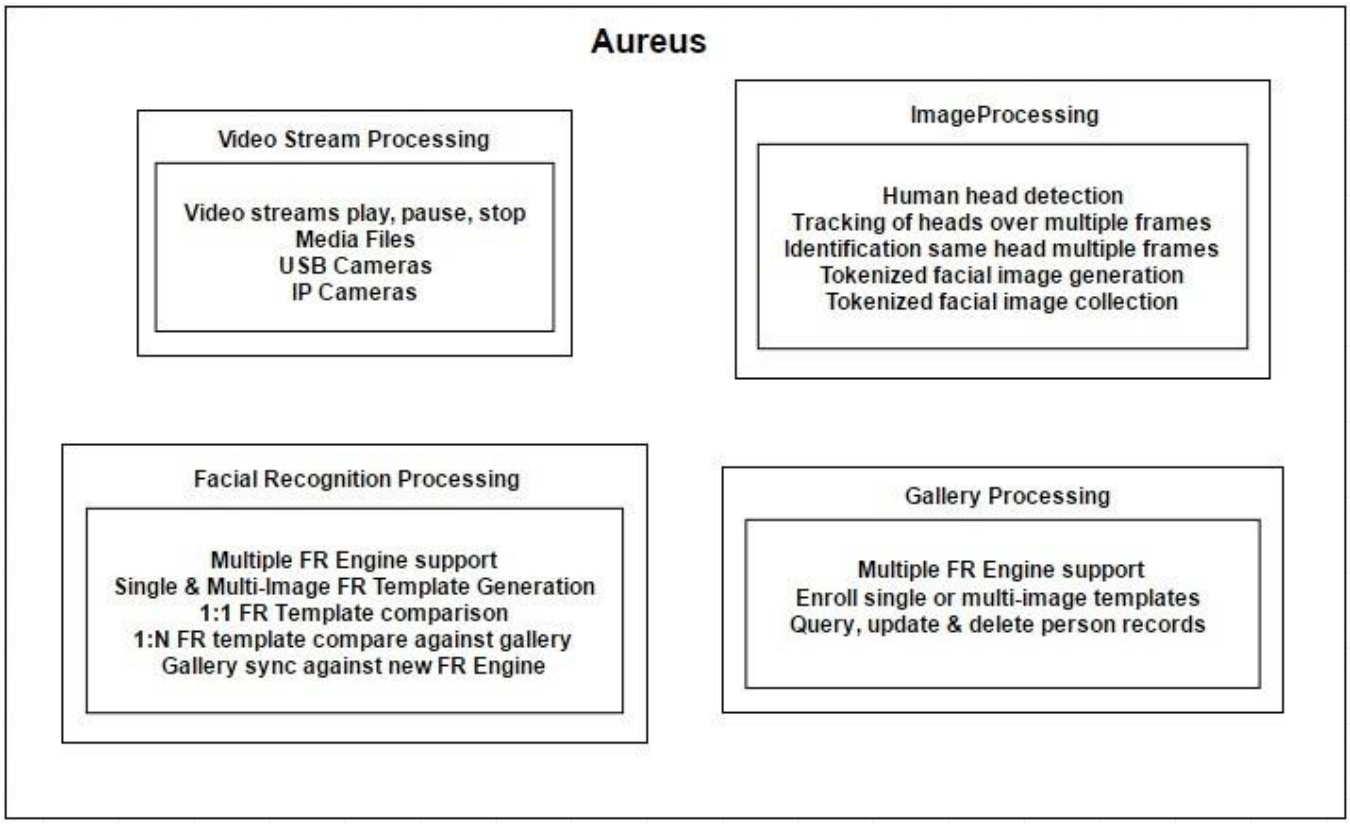
The Aureus SDK interface utilizes the standard C language calling mechanism, allowing Aureus functionality to be available from any programming language that interfaces with standard C. All Aureus features are delivered via a DLL (dynamic link library) located in the Aureus installation directory. Any programming language that can load and access DLL functionality may incorporate Aureus, including Java, C/C++, Python, Perl, C#, Scala, Go, Swift, Objective C, R, PHP, JavaScript (Node.js), Ruby, Haskell, Matlab, Visual Basic and more. Additionally, many applications, such as the Microsoft Office family of applications as well as many CAD, 3D graphics, statistical, and analysis packages provide embedded versions of Visual Basic, JavaScript, Lua, as well as proprietary languages that include DLL loading and functional access. How one goes about running Aureus within Excel is beyond the scope

of this document, but it is entirely possible. Using Aureus in languages other than C/C++ requires the exact same steps and data as detailed in this document, just translated to the language in use.

Aureus provides facial recognition gallery enrollment, so that human heads of interest may be retained for later identification. Aureus is a modular system, capable of separate use to:

a. Real time detection, tracking and identification (either 1:1 or 1: many) across multiple video streams
b. Generating facial recognition templates (which can be then used for 1:1 facial recognition)
c. Performing ranked facial recognition against a gallery
d. Live enrolling of new templates for future facial recognition

The number of video streams and the size of a facial recognition gallery is limited only by the hardware running Aureus.




# 4 How to Use Aureus

Aureus provides a series of SDK objects for passing information to and from the SDK. Regardless of the language one uses, the Aureus objects and their related data are the same. All Aureus objects are runtime checked to insure the correct object is passed when expected. While the Glossary of Terms for this document provides descriptions of these objects, as well as most technical items important to Aureus, please see the header files Aureus.h, AureusAnnotations.h, AureusEnrollVideo.h, AureusGallery.h, AureusHeads.h, AureusImage.h, AureusVideo.h, and

CX_Aureus.h for the final words on Aureus objects, the functions that can be used, and related supporting types, classes, and utilities available for working with them.

Aureus is re-entrant thread safe. All error messages are provided as function parameter strings. Care should be taken to insure the error message string storage one passes to Aureus is large enough to contain any generated error message. In most cases an error message string with a length of 1024 bytes will be fine, except where explicitly documented by a few functions.

Aureus provides a series of standardized variable types for operating system independence, as well as a series of C structures defining parameter sets and structures used by Aureus.

- cx_real — single precision float
- cx_byte — an 8-bit byte
- cx_int — signed 32-bit integer
- cx_uint — unsigned 32-bit integer
- CX_DetectionParams — struct holding face detection parameters
- CX_RAM_Image — struct holding an image in RAM
- CX_HeadInfo — struct holding info of a tracked/detected head
- CX_PersonInfo — struct holding gallery info about one person
- CX_RankItem — struct holding one facial recognition match result
- CX_ResultsSettings — struct holding Aureus results output configurations
- CX_FRTemplate — struct holding one facial recognition template
- CX_MultiFRTemplate — struct holding a multi-template of one person

Please see the CX_Aureus.h and related header files for more details about this information.

The installation of Aureus provides several example projects with source code. All example projects use Microsoft Visual Studio 2013. The installation provides C++ and C# projects that demonstrate how to use Aureus for detection, tracking and 1:N Facial Recognition for sequential images, media file streams and both USB and IP camera streams. In addition to the console example programs the installation provides a more complicated GUI based example project utilizing wxWidgets. This project demonstrates multiple video stream detection, tracking and FR along with gallery display and manipulation and live enrollment.

# 5 AUREUS LICENSING*

To use Aureus in one's own software, a valid license file must be provided within the Aureus installation directory. To obtain a license file, the host system's machine ID must be sent to CyberExtruder for license generation. There are two methods to obtain a host system's machine ID:

1. The command line executable program named MachineID may be executed. The output is a string that is that system's machine ID.

2. The Aureus SDK function **CX_GetMachineID(char *msg)** may be called. This routine does not require an initialized Aureus instance, and may be called prior to creating and initializing Aureus. The single parameter is string storage for the machine ID. The only possible error is the passed string storage is not large enough to hold the machine ID; in that case, the passed message storage will not be large enough to hold the error message, so the error is printed to std out. The pre-built C++ example console program Aureus_Tracking.exe will also print out the machine ID string.

Once the host system's machine ID string is obtained, please send it to CyberExtruder for license generation. Please also provide information regarding the number of video streams you wish to process on the licensed machine and whether you wish to utilize facial recognition. After receiving the license file (named ***AureusSDK_License.txt***) it should be stored in the installation directory.

# 6 DEVELOPING SOFTWARE INCORPORATING AUREUS

Aureus enables developing software applications with visual intelligence capabilities. Nothing within Aureus requires a graphical interface, enabling both command line fully automated applications as well as interactive GUI based applications. The demo applications provide both pure command line and GUI based examples. An example command line Aureus based application might be batch processing thousands of video files for anonymous recognition and grouping. Example GUI based applications range from enrollment kiosks, to gallery lookup, to automated access/entry, to mobile device authentication. Through this range of environments, this document describes incorporating Aureus into your software with these types of requirements:

- Creating and initializing an Aureus object
- Creating one or more Video objects
- Streaming video and utilizing the processed data
- Processing sequential images.
- Adding, reviewing, deleting of single and multi-image person records
- Performing 1:1 and 1: N facial recognition searches
- Updating galleries with new FR engines
- Optional gallery queries for GUI generation

# 7 CREATING AND INITIALIZING AUREUS*

To create and initialize Aureus after a valid Aureus SDK license file has been obtained, the following steps should be taken:

1. **CX_CreateAureus(char *msg)** is the routine that will create an instance of Aureus. This routine should be called before any other Aureus routines. (The only exception is the calling of **CX_GetMachineID()**.) The single parameter is storage for an error message, should Aureus fail to create. The return value will be NULL if an error occurs, or the return value will be a pointer to the Aureus object upon success. This pointer is a required parameter to other Aureus routines.
    a. Remember to free the Aureus object at program termination with a call to **CX_FreeAureus()**.

2. Optionally, one may want to enable verbose error messages with a call to **CX_SetVerbose(CX_Aureus p_aureus, int use_verbose, char* msg)**. This can be helpful identifying if the installation has become corrupted.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter - set to 0 turns off verbose error messages, and set to 1 turns on verbose error messages.
    c. The 3rd parameter is storage for an error message.
    d. A return value of 0 indicates an error occurred and in this situation one should check msg for more information. A return value of 1 indicates success.

3. Optionally, one may want to obtain the Aureus SDK version string. **CX_GetVersion(CX_Aureus p_aureus, char* msg)** will return 1 indicating success with msg containing the Aureus version string. A return value of 0 indicates error and msg contains the error message.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter will be the Aureus version string upon success, or the error message upon failure.

4. **CX_Initialize(CX_Aureus p_aureus, int load_gallery, char* aureus_directory, char* msg)** is the next required routine one needs to call. This will initialize Aureus, returning 0 if an error occurs, or returning 1 to indicate success. Aureus 5.3 and higher uses an SQLite database to create and maintain a facial recognition gallery of known people. SQLite is fast, and rather than operating in a client/server context, SQLite's processing is embedded in Aureus itself, enabling very large galleries, and fast performance with galleries in excess of 1M people.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter is an int; set to 1 asks Aureus to load **AFR_Gallery.gal** located inside the **FR** directory within the directory specified by the third parameter. Set to 0 requests Aureus to ***not*** load the gallery.
    c. The 3rd parameter is the Aureus installation directory where the Aureus data files, dynamically linked libraries, and the Aureus license file is located.
    d. The last parameter is storage for an error message. This will only contain valid information when the return value is 0.

When passing **load_gallery = 1,** the default gallery located at **install-directory/FR/AFR_Gallery.gal** is loaded.

5. Optionally, one may want to retrieve one's Aureus license information after successful initialization of Aureus. The license information is the time remaining which the active Aureus license will be valid. **CX_GetLicenseInfo(CX_AUREUS p_aureus, char* msg)** will place this information into msg. Success is indicated by a return value of 1, and failure will have a return value of 0.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter will be the Aureus license information upon success, or an error message upon failure.

At this point Aureus is initialized and ready for use. Note that if a gallery load was requested, the default gallery is active. Aureus can handle different galleries, as well as different facial recognition (FR) engines. Changing the active gallery or the active FR engine is a global operation. The following section describes changing the active gallery and the active FR engine. After that begins the discussion of the Video object and the Aureus features they provide. It is worth noting that while multiple Video objects may be created, each having different features enabled, the active gallery and the active FR engine will be used by all Video objects.

# 8 GALLERY AND FR ENGINE OPERATIONS

When initializing Aureus on step 4, if the gallery load is requested, the default gallery named **AFR_Gallery.gal** inside the **FR** directory will be loaded. A call to **CX_ChangeGallery(CX_AUREUS p_aureus, const char* gal_name, char* msg)** can be used to change the facial recognition gallery. Returning 0 to indicate failure, or 1 to indicate success, users should be aware that any active video stream processing is stopped. Any previously running video streams will need to be restarted.

a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
b. The 2nd parameter is the full path to the gallery. The gallery is required to be inside the **install-directory/FR** directory.
c. The last parameter is storage for an error message. This only contains valid information when the return value is 0.

To get the number of FR engines installed, a call to **CX_GetNumFRengines(CX_AUREUS p_aureus, char* msg)** will return some positive number upon success, 0 if no FR engines are installed or -1 for failure. Aureus is installed with a default FR engine, so please check one's parameters, and the error message should this fail.

a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
b. The last parameter is storage for an error message. This only contains valid information when the return value is -1.

Granted the number of FR engines available is one or more, a **char*** is returned by **CX_GetFRname(CX_AUREUS p_aureus, int index, char* msg)** that is a string of the name of a requested FR engine. This can be used to collect the names of all the available FR engines.

a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
b. The 2nd parameter is the index of the desired FR engine. Must be within the range given by **CX_GetNumFRengines()**, if not then the return value will be NULL.
c. The last parameter is storage for error message. This only contains valid information when the return value is NULL.

Desiring a different active FR engine, calling **CX_SelectFRengine(CX_AUREUS p_aureus, int index, char* msg)** will return 0 upon failure, or a 1 to indicate success changing the active FR engine.

a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
b. The 2nd parameter is the index of the desired FR engine. Must be within the range given by **CX_GetNumFRengines()**, if not then the return value will be 0.

c. The last parameter is storage for an error message. This only contains valid information when the return value is 0.

Once the FR engine has changed, or the active gallery changed, all further Aureus operations will incorporate those active facilities. If any Video objects existed prior to changes of active gallery or active FR engine, the data generated and written by Aureus using these Video objects is altered.

In each of the following sections, a Video object is required. Aureus allows multiple video streams to be processed simultaneously, limited only by the host system's processing capability and the licensed number of video streams. The creation and initial configuration of a Video object is common to each of these operations. The following section describes that common logic for creation and initial configuration of an Aureus Video object. After the common logic section, a section itemizing the steps necessary to process a series of sequential individual images, is presented. After that is a brief overview of the common aspects of processing video streams, and finally sections detailing how to process media files, how to process USB video, and how to process IP video.

# 9 COMMON LOGIC OF CREATING A VIDEO OBJECT AND INITIAL CONFIGURATION OF A VIDEO OBJECT*

Once an Aureus object instance has been created and initialized, one needs to create a Video object for processing of media. The following steps are common logic for creating initializing a Video object.

1. Create an Aureus Video object by calling **CX_CreateVideo(CX_AUREUS p_aureus, char* msg).** Upon success a CX_VIDEO object will be returned; the returned CX_VIDEO object is a required parameter for future Aureus calls using a Video object. NULL is returned upon failure; in the case of failure, msg will contain explanatory information. In the case where multiple video streams, multiple sequential image sets, or any combination of media types are to be processed simultaneously, each media stream requires its own Video object. One may create as many Video objects as one is licensed.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter will only be valid if an error occurs, containing an error message upon return.
    c. The return value upon success is a new Video object; this Video object is a required parameter for future Aureus routines. The returned Video object should not be freed or deleted.

To perform facial recognition against the gallery, four key conditions must be met: (a) one's Aureus must be licensed for facial recognition, (b) when initializing Aureus, the **load_gallery** parameter must be set to 1 (true), (c) the **generate_templates** flag must be set to 1 (true) (step 2, this section), and (d) the **perform_gallery_fr** flag must be set to 1 (true) (step 3, this section).

Additionally, when processing video streams, automated results saving can be enabled, and a verification threshold may be used to filter the results. These aspects are described by steps 7 thru 9, below.

2. To enable template generation, and ultimately any facial recognition, **CX_SetGenerateTemplatesFlag(CX_VIDEO p_video, int generate_templates, char* msg)** must be called. It is best practices to always call this function, regardless of one wanting template generation or not;

meaning one's logic should call **CX_SetGenerateTemplatesFlag()** with the **generate_templates** parameter set to 0 when no template generation is desired, as well as one should call **CX_SetGenerateTemplatesFlag()** with the **generate_templates** parameter set to 1 when template generation is desired. Remember that Aureus is modular, and may be used purely for template generation if needed.

   a. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
   b. The 2nd parameter is an int; set to 1 requests template generation, and set to 0 requests no template generation through this Video object.
   c. The last parameter is storage for error messages. This will only contain valid information upon failure.

3. To enable facial recognition against the gallery, **CX_SetPerformGalleryFRFlag(CX_VIDEO p_video, int perform_gallery_fr, char* msg)** must be called**.** Similar to **CX_SetGenerateTemplatesFlag()**, it is considered best practices to call this routine with **perform_gallery_fr** set to 0 when no gallery matching is desired, as well as being required to call this function with **perform_gallery_fr** set to 1 when gallery matching is needed. *Calling these two routines even when the feature is not requested is an issue of clarity; it is not required by any means, but makes one's code intention obvious.* **CX_SetPerformGalleryFRFlag()** will return 1 upon success and 0 upon failure.
   a. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
   b. The 2nd parameter is an int; set to 1 requests facial recognition against the gallery, and set to 0 requests no facial recognition gallery matching with this Video object.
   c. The last parameter is storage for error messages. This will only contain valid information upon failure.

Next one needs to set the head detection parameters to be used when examining images. Processing sequential images or processing a video stream requires one to set where in the images or frames examination should occur (i.e. a region of interest) as well as the minimum and maximum size heads (i.e. a head size range). There are also two additional properties that should be set at this time which are considered "detection parameters": (a) frame reduction: Aureus can reduce the size of the frame which can significantly speed up detection without effecting tracking or FR as long as the heads would still fall in the min/max size range, and (b) frame interval: Aureus allows you to automatically skip frames if you wish to, this can also significantly speed up processing as there are many situations where one does not need to process all frames.

4. To set the region of interest and the min/max head size range, one calls **CX_SetDetectionParameters(**`CX_VIDEO p_video, float top, float left, float height, float width, float min_height_prop, float max_height_prop, char* msg`**).** `The return value is 1 for success, or 0 for failure.`
   a. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
   b. The 2nd, 3rd, 4th, and 5th parameters define the region of interest. These are normalized numbers, meaning valid values are in the range 0.0 to 1.0. The origin is top left, meaning the location at 0.0, 0.0. To examine the entire frame, use top=0.0, left=0.0, height=1.0 and width=1.0. To only examine the upper right quarter of each frame, use top=0.0, left=0.5, height=0.5, and width=0.5.
      i. The defaults are top=0.0, left=0.0, height=1.0 and width=1.0, the entire frame.

c. The 6th and 7th parameters define the minimum head height and the maximum head height. These are also normalized numbers.
    i. The defaults are min_height_prop=0.2 and max_height_prop=0.4.
d. The last parameter is storage for an error message, should an error occur. This only contains valid information when an error occurs.

5. To set the frame reduction, call **CX_SetFrameReduction(CX_VIDEO p_video, int frame reduction, char* msg)**. Frame reduction is only used during head detection. Frame reduction reduces the size of the image during detection, which can significantly improve detection performance. Aureus successfully detects heads as small as 25 pixels from chin to top of head. Therefore, use of frame reduction can significantly increase performance. **CX_SetFrameReduction()** returns 1 on success or 0 on failure. Frame reduction has three possible modes of operation:
    a. The 1st parameter is the Video object returned by CX_CreateVideo().
    b. The 2nd parameter controls the frame reduction:
        i. Setting to -1 or 0 enables automatic frame reduction, meaning Aureus will internally determine an appropriate frame reduction based upon the frame size and the min_height_prop set by calling CX_ SetDetectionParameters().
        ii. Setting to 1 disables frame reduction.
        iii. Setting to any value greater than 1 defines the denominator to use for frame reduction. Meaning if set to 2, then the frames' height and width will be divided by 2, producing a detection frame half its original size. Likewise, if set to 3, the frames' height and width will be divided by 3, producing a detection frame 1/3rd its original size.
    c. The last parameter is storage for an error message, should an error occur. This only contains valid information when an error occurs.

6. To set a frame interval, use **CX_SetFrameInterval(CX_VIDEO p_video, int frame_interval, char* msg)**. In some situations, it is beneficial to set the frame interval to a value of 2, 3 or even higher. Aureus can successfully detect, track and identify with frame intervals larger than 1 in situations of high frame rates, such as 60 fps, slow moving individuals, such as people shuffling in a line, as well as Aureus operating on a relatively low powered system.
    a. The 1st parameter is the Video object returned by **CX_CreateVideo().**
    b. The 2nd parameter controls the frame interval:
        i. Setting to 0 or 1 causes every frame to be examined.
        ii. Setting to 2 causes every 2nd frame to be examined, setting to 3 causes every 3rd frame to be examined.
    c. The last parameter is storage for an error message, should an error occur. This only contains valid information when an error occurs.

With or without facial recognition active, Aureus can be directed to save/post results for each detected head. The CX_ResultsSettings structure is used to communicate this per Video object configuration. Please examine the CX_ResultsSettings structure to learn more details about the configuration settings.

7. One may call **CX_GetResultsParameters(CX_VIDEO p_video, char* msg)** to receive a pointer to a filled out CX_ResultsSettings structure holding the current configuration for this Video object. This returns NULL on error. When successful, do not free/delete the returned pointer.
    a. The 1st parameter is the Video object returned by **CX_CreateVideo().**
    b. The last parameter is storage for an error message, should an error occur. This only contains valid information when an error occurs.
8. One may call **CX_SetResultsParameters(CX_VIDEO p_video, CX_ResultsSettings results, char* msg)** to set the Results Saving configuration for this Video object. This returns 0 on error. When successful this returns 1.
    a. The 1st parameter is the Video object returned by **CX_CreateVideo().**
    b. The 2nd parameter is a filled out CX_ResultsSettings conveying the desired Results Saving Configuration
    c. The last parameter is storage for an error message, should an error occur. This only contains valid information when an error occurs.
9. When saving results via one of the configurations in step 8, and facial recognition is active, one may filter the amount of information saved by setting the *verification threshold*. Consider verification threshold a match score threshold that a detected head must match or exceed against the gallery before being saved by the automated Results Saving Configuration. Valid values lie between 0.0 and 1.0, with typical values around 0.7 – indicating a tracked head needs a match score >= 0.7 to trigger automated Results Saving. Calling **CX_SetVerificationThreshold(CX_VIDEO p_video, cx_real verification_threshold, char* msg)** will return 0 on error, or 1 when the verification threshold sets successfully.
    a. The 1st parameter is the Video object returned by **CX_CreateVideo().**
    b. The 2nd parameter is the desired 0.0 to 1.0 verification threshold.
    c. The last parameter is storage for an error message, should an error occur. This only contains valid information when an error occurs.

# 10 USING AUREUS TO PROCESS SEQUENTIAL IMAGES*

Once the Aureus object has been created and initialized, as well as a Video object has been created and configured, the following steps may be taken to process a series of sequential frames. It is recommended one read this section before reading future sections describing how to process video streams, as the process is very similar. Processing video streams consists of receiving each video frame individually via launching a video process with a callback to one's own logic. As the video plays, the callback function receives each frame one after another. With this architecture, the logic of one's own code is very similar to processing a sequential set of images.

Two variations are presented here: (a) a version that performs simultaneous detection and tracking, and (b) a version that performs head detection separate from tracking. Note that neither of these sections include template generation or facial recognition, they only demonstrate detection and tracking. Template generation and/or facial recognition could be included, but are described in the video stream handling sections for the sake of keeping these initial descriptions simple and easy to follow. Please also refer to the Aureus_Tracking C++ example project.

When processing sequential image sets, the images are expected to be sequential in nature, as if a video stream were separated into individual images and each saved to disk in a numbered series of individual files. The same Video object may be used to process different sets of sequential images, but should call **CX_ResetProcessing(CX_VIDEO p_video, char* msg)** between each different set of sequential images, to prevent images from one set being combined with the new sequential set. This returns 0 if an error occurs, or it returns 1 to indicate success.

1. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
2. The last parameter is storage for an error message. This only contains valid information when an error occurs.

# 11 Simultaneous Head Detection and Tracking in Sequential Images

To perform simultaneous detection and tracking in a sequential series of images, one must have previously created and initialized an Aureus object, created a Video object, and defined the detection parameters to be used with that Video object. After these prior steps have been successfully completed, follow these steps to perform simultaneous detection and tracking within a sequential series of images:

1. Using one's own methods, or via the provided cxutil.cpp utility function **CX_FindFiles(std::vector<std::string>& kFnames, const char* szPath, const char* szWildCards)**, collect a file path list to the sequential set of images to be processed.

2. Looping over each image path:
    a. Read the current image into RAM. The cxutils.cpp utility function **LoadImageFromDisk(const char* path, CX_RAM_Image& im)** will read a jpeg image into a CX_RAM_Image structure. The CX_RAM_Image structure is used to pass images to Aureus. One does not have to use **LoadImageFromDisk()**, but one does need images intended for Aureus processing to be stored in a CX_RAM_Image.
        i. The 1st parameter is the path to the image to be loaded
        ii. The 2nd parameter is a reference to a CX_RAM_Image to load the image into.

    b. Call **CX_ProcessFrame(CX_Video p_video, CX_RAM_Image* im, int frame_number, int use_face_detector, char* msg)**. This will return a CX_HEAD_LIST object on success, or NULL on failure. DO NOT free or delete the returned pointer. The CX_HEAD_LIST and all its internal values are valid until the next call to **CX_ProcessFrame()** or **CX_ProcessFrameFromDetectedHeads()** (described next section)**.** Use of a previously returned CX_HEAD_LIST after another call to **CX_ProcessFrame()** or **CX_ProcessFrameFromDetectedHeads()** will result in undefined behavior.
        i. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
        ii. The 2nd parameter is a pointer to the CX_RAM_Image to be processed.
        iii. The 3rd parameter is the frame number, and in the case of a sequential set of images, that is the image's ordinal location within the set. This parameter is required for tracking. Aureus expects repeated calls to this function to have incrementing frame numbers.

Frame numbers are expected to be positive integers, typically beginning with 0, but do not have to start at 0.

iv. The 4th parameter controls the type of detection; set to 1 causes only face detection to occur. Set to any other value causes head and shoulders detection prior to face detection.

v. The last parameter is storage for an error message. This only contains valid information when an error occurs.

c. Granted that **CX_ProcessFrame()** returned a valid CX_HEAD_LIST, **CX_GetHeadListSize(CX_HEAD_LIST p_head_list, char* msg)** may be called to retrieve the number of heads detected in the image. This returns 0 for no heads detected, a value greater than 0 when heads are detected, and -1 upon error. When the value is greater than 0, that value is the number of detected heads from the prior call to **CX_ProcessFrame().**

i. The 1st parameter is the CX_HEAD_LIST returned from the last call to **CX_ProcessFrame().**

ii. The last parameter is storage for an error message, and only contains valid information if an error occurred.

d. Granted that **CX_GetHeadListSize()** returns a value greater than 0, one may call **CX_GetHead(CX_HEAD_LIST p_head_list, int head_number, char* msg)** to retrieve individual CX_HEAD objects from the head list. This returns NULL upon error, or the requested CX_HEAD upon success.

i. The 1st parameter is a CX_HEAD_LIST object, and in this context is the head list returned from the last call to **CX_ProcessFrame()**.

ii. The 2nd parameter is the head index to return. The first head in a head list has an index number of 0, the second head in the list is index number 1 and so on.

iii. The last parameter is storage for an error message. This only contains valid information when an error occurs.

e. A call to **CX_GetHeadInfo(CX_HEAD p_head, CX_HeadInfo* p_head_info, char* msg)** will fill the passed CX_HeadInfo structure with detailed information about the passed CX_HEAD object. This will return 0 upon failure, or 1 to indicate success.

i. The 1st parameter is a CX_HEAD object retrieved from a call to CX_GetHead().

ii. The 2nd parameter is a pointer to a CX_HEADInfo structure to fill with information.

iii. The last parameter is storage for an error message. This only contains valid information when an error occurs.

f. At this point, one's logic has had the opportunity to examine detailed information about the existence of detected and tracked heads in the current frame. If you used the utility function **LoadImageFromDisk** you should now free the associated memory with a call to **DeleteImagePixels(CX_RAM_Image& im)**.

3. Once the sequential set of images are all processed, within the context of this example, one is finished. However, one must be sure to call **CX_FreeAureus (CX_AUREUS p_aureus, char* msg)** to terminate any thread processing, and free any allocated memory held by Aureus before ending one's program.
    a. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
    b. The last parameter is storage for an error message, should an error occur. This only contains valid information when an error occurs.

The above steps are illustrated by a code example near the end of this document.

# 12 Two pass: Head Detection in the First Pass and Head Tracking in the Second Pass in Sequential Images

To perform separate detection of heads in one pass and tracking of the detected heads in a second pass over a set of sequential images, one must have previously created an Aureus object, initialized it, created a Video object, and defined the detection parameters to be used with that Video object. After these prior steps have been successfully completed, follow these steps to perform separate detection of heads in one pass and tracking of the detected heads in a second pass over a set of sequential images.

Note that unlike the previous section, where each image is allocated, processed, and freed in a loop, this example loads all the frames at once. Loading them all at once facilitates the two pass logic of this example.

1. Using one's own methods, or via the provided cxutil.cpp utility function **LoadFrames(const char* dir, std::vector<CX_RAM_Image>& frames, bool print_details = true)**, load into RAM the sequential set of images to be processed.

2. Using std::vector<CX_HeadInfo*>, or your own methods, define a dynamic array of CX_HeadInfo pointers. This dynamic array will be used to maintain the detected heads collected during the first pass through the sequential image set. After step 5, below, each element of this dynamic array will point to an array of CX_HEADInfo structures for each processed frame.

3. Using std::vector<int>, or your own methods, define a dynamic array of integers. This dynamic array of integers will be used to maintain the number of detected heads found in each image of the sequential image set. After step 5, below, this dynamic array of integers will contain the number of heads detected in each processed frame.

4. Define the maximum number of heads to be detected in each frame through the sequential image set. Consider the situation where the set of images contain 50 or more people per image, all of varying distance from the camera. Setting the maximum number of heads to 20, for example, will cause Aureus to return the first 20 heads detected in each image, which is not necessarily the 20 people closest to the camera. As illustrated below, that situation is identifiable via a special return code from **CX_DetectHeads()**, enabling one's logic to catch the situation and increase the maximum number of heads, if necessary.

5. Looping over each image:

    a. Allocate storage for one's maximum desired CX_HEADInfo structures for the current image. Allocate this within the dynamic array of CX_HEADInfo pointers created in step 2 above, where the element within the array of pointers is equal to the current frame number.

    b. Call **CX_DetectHeads(CX_VIDEO p_video, CX_RAM_Image* p_im, int frame_number, CX_HeadInfo* p_head_info_array, int n_array_size, int use_face_detector, char* msg)** to perform head detection on the current image. This will return -1 upon failure; it will return -2 if more heads are detected than the size of the CX_HEADInfo* array; or it will return the number of detected heads when the number of detected heads is able to fit within the CX_HEADInfo* array. Returning 0 is valid, indicating no detected heads.
        i. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
        ii. The 2nd parameter is a pointer to the current CX_RAM_Image
        iii. The 3rd parameter is the current image number within the sequence (required for tracking)
        iv. The 4th parameter is the storage for the maximum desired CX_HEADInfo structures allocated in the prior step. This should be the CX_HEADInfo storage allocated in step "a" just prior to this step.
        v. The 5th parameter is the maximum number of desired heads (the number of elements in the CX_HEADInfo array)
        vi. The 6th parameter controls the type of head detection. If set to 1, only face detection will occur, otherwise head and shoulders detection occurs first followed by face detection.
        vii. The last parameter is storage for an error message. This only contains valid information when an error occurs.

    c. Within the std::vector<int> defined at step 3, save the number of detected heads for this image.

6. Having completed the loop in step 5, the logic now has a dynamic array of CX_HEADInfo pointers populated with the detected heads for each image within the sequential set of images. Now is the time for the second pass, so create another loop over the image set and do the following inside that loop:

    a. For each loop iteration, call **CX_ProcessFrameFromDetectedHeads(CX_VIDEO p_video, CX_RAM_Image* p_im, int frame_number, CX_HeadInfo* p_head_info_array, int n_heads, char* msg)**. This will return NULL upon failure, or it returns a pointer to a **CX_HEAD_LIST** object when successful. This is the same type of CX_HEAD_LIST object returned by **CX_ProcessFrame()** described in the previous section. DO NOT free or delete the returned pointer. The CX_HEAD_LIST and all its internal values are valid until the next call to **CX_ProcessFrameFromDetectedHeads()** or **CX_ProcessFrame()** (described previous section)**.** Use of a previously returned CX_HEAD_LIST after another call to **CX_ProcessFrameFromDetectedHeads()** or **CX_ProcessFrame()** will result in

undefined behavior. Once in possession of the CX_HEAD_LIST, the same processes with that object as described in the previous section may be applied.

i. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
ii. The 2nd parameter is a pointer to the CX_RAM_Image to be processed.
iii. The 3rd parameter is the frame number, and in the case of a sequential set of images, that is the image's ordinal location within the set. This parameter is required for tracking. Aureus expects repeated calls to this function to have incrementing frame numbers. Frame numbers are expected to be positive integers, typically beginning with 0, but do not have to start at 0.
iv. The 4th parameter is the pointer to the CX_HEADInfo array populated during the previous pass through the images, and should be the CX_HEADInfo pointer array equal to this same image used to acquire that CX_HEADInfo array.
v. The 5th parameter is the number of heads in the array, NOT the array size.
vi. The last parameter is storage for an error message. This only contains valid information when an error occurs.

7. At this point in the logic, two passes have been accomplished. Before one is complete, some cleanup needs to occur. The dynamic arrays created during steps 2 and 3, as well as the set of frames loaded into RAM during step 1 should be deleted.

The above steps are illustrated by a code example near the end of this document.

Note that not everything one could do with the processes described by these two examples has been detailed. For simplicity, only detection and tracking has been described. In the next sections processing of video streams is described. The same operations as detailed above can be performed on video frames, but rather than detailing that information again, the next sections will detail how to generate facial recognition templates, how to enroll people into the facial recognition gallery, and how to request facial recognition. Aureus is modular, enabling tracking, detection, template generation, enrollment, and facial recognition simultaneously or in sequential passes, across multiple video streams and sequential image sets simultaneously. If interested and ambitious, even more can be accomplished than is detailed by this document (such as face swapping or people monitoring).

# 13 Common Aspects of Consuming Video Streams*

In many respects, the processing of video streams with Aureus is trivial, as long as one understands the processes described by the previous sections. If one has not read the previous sections, it is strongly advised one read them first. This document incrementally builds one's understanding of Aureus, and the previous sections contain critical information one is required to understand before this section.

Processing of different types of video streams, meaning a media file, or a USB camera's video output, or an IP camera's output, consists of the same creation and initializing of an Aureus object, creating a Video object, and defining the desired detection parameters to be used with that Video object. Working with video streams varies from sequential image sets in three key respects:

- A **Frame Callback** must be installed into one's Aureus Video object. The Frame Callback will be called by Aureus, sending each frame of the video stream to the Frame Callback one after another. This architecture enables one's logic to be similar to the logic used to process sequential image sets, because within one's Frame Callback the situation is just like the handling of one image from a sequential image set. The situation varies from sequential images in the respect that a USB or IP camera's video frames are being processed in real time. If the application has human operators, the Frame Callback is how the end-developer receives the video frame images to display to the user. If one's Frame Callback takes longer than the display time of a USB or IP camera frame, frame dropping will occur. When one's Frame Callback returns control to Aureus, the next frame available in real time will be sent to one's Frame Callback.

    - A Frame Callback to a Video object is a function in one's code fitting this calling signature:

```
typedef void(*CX_FRAME_CALLBACK) (CX_HEAD_LIST  p_head_list,
                                  cx_uint       head_list_size,
                                  cx_byte*      p_pixels,
                                  cx_uint       rows,
                                  cx_uint       cols,
                                  cx_uint       frame_num,
                                  void*         p_object);
```

      A Frame Callback is a function receiving a CX_HEAD_LIST object, the size of that CX_HEAD_LIST, an array of RGBA pixels, the number of rows and columns in the RGBA data, the frame number and a void pointer to a data structure from one's own logic.

      The CX_HEAD_LIST contains the active set of detected and tracked heads, which may be used for operations such as new person enrollment into the gallery or setting an alarm if a person is identified, or anything you choose to do.

      The RGBA data will always have the origin located bottom left, and is always organized row major.

      The last parameter can be a pointer any data one desires, as this is a pointer from one's own logic passed to the Video object when setting the Frame Callback. The last parameter is not altered or accessed by the Video object; it can be NULL if no private data passing is desired. An example Frame Callback is provided near the end of this document.

- An Unexpected Stream Termination Callback can be installed into the Video object. The stream termination callback will be called by Aureus in the event of unexpected video stream termination. An unexpected stream termination can occur when someone unplugs or power loss occurs with the USB or

IP camera, or the media file on disk is corrupted, rendering it unplayable at some point after initial stream playback.

- A stream termination callback of a Video object is a function in one's code fitting this calling signature:

```
typedef void(*CX_STREAM_TERMINATED_CALLBACK) (cx_int      stream_type,
                                              const       char*
connection_info,
                                              void*       p_object);
```

A stream terminated callback is a function receiving the type of stream terminated, with valid values of 0 for media files, 1 for a USB video stream, and 2 for an IP camera video stream.

The connection info parameter is the same connection info used to initiate this stream's processing, i.e.:

- the path to the media file
  - typically, a string like "C:/securityCams/cam0/01_06_2016/daily.wmv"
- a string containing the USB video pin
  - typically, a string like "0"
- a string containing the URL to the IP Camera,
  - typically, a string like "rtsp:admin:admin@192.168.1.108"
  - please consult your IP camera documentation for the specifics of the URL required to initiate video stream reception from one's IP camera.

The last parameter is void pointer to any data one wants; this is similar to the void pointer to any data as is received by one's Frame Callback, but an independent void pointer. Therefore, one may have specific custom data received by the termination callback as is received by the Frame Callback. This last parameter may also be NULL. An example stream terminated callback is provided near the end of this document.

- A Media File Finished callback function can be provided to an Aureus Video object. If provided and the video object is processing a media file the callback function will be called after the last frame have been processed. The callback signature is:

```
typedef void(*CX_MEDIA_FILE_FINISHED_CALLBACK) (cx_uint frame_num, void* p_object);
```

  - The `frame_num` parameter provides the last frame number of the processed media file.

- The last parameter is void pointer to any data one wants. This last parameter may also be NULL.

- Aureus's stream processing routines are used to initiate processing of the video stream, to pause the video stream, resume playback, and stop the video stream when completed.

As can be seen, the Aureus architecture for handling of video streams provides a logical flow allowing for both sequential image sets and any of the three types of video streams to be handled by nearly identical logic. One can easily model their sequential image loading and handling such that a video stream Frame Callback is used to handle their processing, as well as the same Frame Callback could be used to process any of the three video stream types. These common aspects are further illustrated in the next section, the processing of a media file.

# 14 Using Aureus to Process Media Files*

Once an Aureus object instance has been created and initialized, and a Video object has been created and the desired detection parameters have been set for that Video object, the following steps may be taken to process a media file.

1. First one needs to define the Frame Callback to be used for the processing of each video frame. A Frame Callback, in addition to fitting the required calling signature for Frame Callbacks, must contain any processing necessary in response to the detected heads passed in the CX_HEAD_LIST. Because a Frame Callback is called during video playback by Aureus, what occurs inside the Frame Callback is not detailed in this step, but lower down in this same section. For the purpose of creating code while following this section, one should define an empty function fulfilling a Frame Callback's calling signature.

2. Next one must install the Frame Callback into the Aureus Video object which will process the media file. This is accomplished by calling **CX_SetFrameCallBack(CX_VIDEO p_video, CX_FRAME_CALLBACK p_func, void* p_object, char* msg)**. The return value is 1 upon success, or 0 if an error occurs. Note that regardless of one processing a media file, a USB video stream or an IP camera's video stream, this same routine is used to install one's Frame Callback into the Video object.
    a. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
    b. The 2nd parameter is a pointer to one's Frame Callback.
    c. The 3rd parameter is a void pointer to any data one desires, or NULL.
    d. The last parameter is storage for an error message. This only contains valid information when an error occurs.

3. Next one can install the unexpected stream termination callback into the Aureus Video object which will process the media file. This is handled by calling **CX_SetStreamTerminatedCallBack(CX_VIDEO p_video, CX_STREAM_TERMINATED_CALLBACK p_func, void* p_object, char* msg)**. The return value is 1 upon success, or 0 if an error occurs. Note that regardless of one processing a media file, a USB video stream

or an IP camera's video stream, this same routine is used to install one's Unexpected Stream Termination Callback.

a. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
b. The 2nd parameter is a pointer to one's Frame Callback.
c. The 3rd parameter is a void pointer to any data one desires, or NULL.
d. The last parameter is storage for an error message. This only contains valid information when an error occurs.

4. Next one can install the media file finished callback by calling **CX_SetMediaFileFinishedCallBack(CX_VIDEO p_video, CX_MEDIA_FILE_FINISHED_CALLBACK p_func, void p_object, char* msg);**
   The return value is 1 upon success and zero up[on error (consult msg for more informative error message).

5. Now one's media file (or USB/IP) video stream can be opened with **CX_ProcessStream(CX_VIDEO p_video, cx_uint stream_type, const char* connection_info, char* msg)**. This will return 0 if an error occurs, or 1 to indicate success. After this call, the video stream begins playback, and the Frame Callback installed at step 2 will begin to receive calls. Likewise, if the stream terminates unexpectedly, the stream termination callback installed during step 3 will be called. Note that within the context of the logic that calls **CX_ProcessStream()**, a call to Sleep() or some other means of delaying program termination is required, because **CX_ProcessStream()** returns immediately. Alternatively, one can set up a media file finished callback so that your code will know when processing has finished.
   a. The 1st parameter to **CX_ProcessStream()** is the Video object returned by **CX_CreateVideo()**.
   b. The 2nd parameter is the type of video stream to be processed. Valid values are 0 for a media file, 1 for USB video, and 2 for an IP camera video stream.
   c. The 3rd parameter is the "connection info" string, which varies depending upon the stream type:
      i. If a media file, the string is the path to the media file
         1. typically, a string like "C:/securityCams/cam0/01_06_2016/daily.wmv"
      ii. If the stream is from a USB camera, the string contains the USB video pin
         1. typically, a string like "0"
      iii. If the stream is from an IP camera, the string contains the URL to the IP Camera,
         1. typically, a string like "rtsp:admin:admin@192.168.1.108"
         2. please consult your IP camera documentation for the specifics of the URL required to initiate video stream reception from one's IP camera.
   d. The last parameter is storage for an error message. This only contains valid information when an error occurs.

6. While a video stream is being processed, one may call **CX_PauseStream(CX_VIDEO p_video, char* msg)** to pause the processing of the video stream, and likewise one may call **CX_UnPauseStream(CX_VIDEO p_video, char* msg)** to resume video stream playback. When pausing and unpausing a camera video stream, this has the same effect as causing frame dropping for the duration the video stream is paused.

a. The 1st parameter to both **CX_PauseStream()** and call **CX_UnPauseStream()** is the Video object returned by **CX_CreateVideo()**.
b. The last parameter is storage for an error message. This only contains valid information when an error occurs.

7. If one wants to terminate playback of a video stream one calls **CX_StopStream(CX_VIDEO p_video, char* msg)**. This returns 0 for failure, or 1 for success.
    a. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
    b. The last parameter is storage for an error message. This only contains valid information when an error occurs.

A note about processing media file streams:

Behind the scenes Aureus loads images whilst processing continues on a separate thread. The end result is a far smoother and faster processing pipeline. However, sometimes this can result in a frame being skipped here and there (depending on processing power and detection set up and frame size and number of people in the frame). Under certain circumstances you may wish to force Aureus to process every single frame, use the **CX_ForceEveryFrame(CX_VIDEO p_video, int flag, char* msg)** function to signal Aureus to force every frame to be processed.

Note that if Aureus is forced to process every frame it effectively waits for the frame to be loaded, then processes it. This results in a slower throughput of frames (often significantly slower) and for any stream type other than media files it will always be significantly slower since the camera will put out frames at a regular frame period and Aureus will load them and process them in one function. If the *flag* parameter is false, then the loading and processing happen in parallel hence the processing of camera frames is significantly faster and smoother.

Setting the *flag* parameter to true really only has relevance for media files where Aureus can get a new frame after finishing processing the previous frame. The frame interval setting is still effective regardless of the setting of this flag.

The default is false, it is highly recommended that it remains false unless you have a need for each media file frame to be processed (e.g. performing detection and tracking without FR on a set of media files for post processing using FR later).

# 15 USING AUREUS TO PROCESS USB VIDEO*

Using Aureus to process USB video is nearly identical to processing a media file saved to disk, with the following exceptions:

- One specifies a stream type of 1 and provides a string containing the USB video pin as the connection info parameter to **CX_ProcessStream()**. (**CX_ProcessStream()** is described in step 4, previous section.)

# 16 USING AUREUS TO PROCESS IP CAMERA VIDEO*

Using Aureus to process IP camera video is nearly identical to processing a USB video stream, with the following exception:

- One specifies a stream type of 2 and the IP camera's URL for the connection info parameter when calling **CX_ProcessStream()**. Because IP cameras vary widely in their URL configuration, one should consult one's specific IP camera documentation and/or one's system administrator for the correct URL configuration.

# 17 FOUR METHODS FOR ENROLLING PEOPLE INTO THE FACIAL RECOGNITION GALLERY

Granted one's Aureus License includes Facial Recognition; one can enroll people into the facial recognition gallery. Four methods exist for enrolling new people into the facial recognition gallery.

- One may create an Enroll Video object and use a prepared enrollment process, with end-user guidance and feedback overlaid on top of the video frames. Using an Enroll Video object follows a similar pattern as playing a USB or IP camera video stream, with the added benefit of the enrollment process handled internally, automatically, by Aureus. This method of enrollment is only appropriate when people are available "live", because only live USB and IP camera streams are supported. This method also expects people being enrolled to see the video stream, because user guidance is overlaid on the video. The guidance is a head placement oval that changes position, forcing the end user to provide a slightly different view of their face for the multi-image enrollment. This method is detailed in the next section.

- One can manually collect one or more facial images for submission to Aureus. This method is detailed in the Manual Enrollment of New People into the Facial Recognition Gallery section.

- When using Video objects and receiving the CX_HEAD_LIST returned by **CX_ProcessFrame()**, and **CX_ProcessFrameFromDetectedHeads()**, or receiving the CX_HEAD_LIST within one's Frame Callback, the CX_HEAD objects contained in the list can be used for direct enrollment. This require some additional logic to use, which is detailed in the Using Video Objects to Enroll New People into the Facial Recognition Gallery section.

- The Aureus installation provides a command line *BatchEnroll.exe* executable program to be used off-line. To use, simply type *BatchEnroll* from the command line prompt and follow the instructions. This is by far the fastest way to populate a gallery as it utilizes multiple threads and database transaction calls.

# 18 Using an Enroll Video Object to Enroll New People into the Facial Recognition Gallery

Slightly different from using Aureus to process sequential image sets or video streams, after creating and initializing one's Aureus object, one does *not* create a Video object, in its place is an Enroll Video object. An Enroll Video object is like a Video object specific to USB and IP camera streams (live video), but its Frame Callback receives slightly different information, related to the additional automated logic performing an enrollment into the facial recognition gallery.

Being very similar a Video object, an Enroll Video object has a Frame Callback, and an Unexpected Stream Termination Callback. The Frame Callback for an Enroll Video object receives similar but not the same data as does a Video object's Frame Callback:

A Frame Callback to an Enroll Video object is a function in one's code with this calling signature:

```
typedef void(*CX_ENROLL_FRAME_CALLBACK) (cx_byte*      p_pixels,
                                         cx_uint       rows,
                                         cx_uint       cols,
                                         cx_uint       frame_num,
                                         cx_int        status,
                                         const char*   msg,
                                         void*         p_object);
```

This Frame Callback is a function receiving an array of RGBA pixels (origin bottom left, always row major), the number of rows and columns in the RGBA data, the frame number, a status integer, an error message (that only has valid data if the status indicates an error occurred) and a void pointer to a data structure from one's own logic.

The status parameter is how Aureus passes the progress of the live enrollment.

The status parameter is key to tracking the automated enrollment process. When status is 0, the enrollment process is active. When status is 1 then the enrollment process is finished. If status is -1, an error has occurred and the msg parameter will contain an error message.

The last parameter can be a pointer any data one desires, as this is a pointer from one's own logic passed to the Video object when setting the Frame Callback. The last parameter is not altered or accessed by the Video object; it can be NULL if no private data passing is desired.

As can be seen from the parameters passed to the Frame Callback, *status* is what is provided to an Enroll Video Object's Frame Callback for monitoring of the live enrollment process. When the Enroll Video object is playing, the Frame Callback will receive frames, which need to be displayed to the person being enrolled because they include guidance to the person. When the status passed to the Frame Callback is 1, the enrollment is complete.

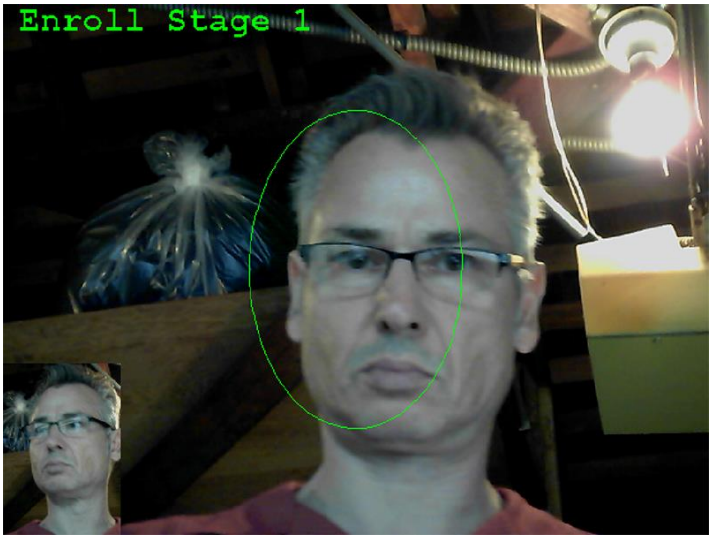
After status is 1, or -1 indicating an error, the Frame Callback will continue to be called, sending new live video frames. As a response to status being or 1 or -1, the end-developer's logic is expected to respond appropriately.

While receiving frames, the Enroll Video object may be paused, which causes the Frame Callback to stop receiving calls, as well as temporarily pause the enrollment process at whatever state it happens to be when paused.

The Unexpected Stream Termination Callback for an Enroll Video object is exactly like its counterpart in a Video Object. They are described in common aspects of video streams, and have a code example near the bottom of this document.

The Enroll Video object expects a specific end-user environment consisting of:

- A USB or IP camera configured for single user eye level viewing/capture
- The viewed/captured user has a view of their live video feed, ideally directly at eye level so they can respond to on screen indicators.
    - Meaning the display and the camera are integrated
    - Or in a kiosk or separate operator setting, the user is directed to look at the camera




While the live enrollment is active, the user is expected to position themselves with their head within the oval. When Aureus detects a face within the oval, a cropped version of the head is automatically created, and a copy is placed along the left edge of the video frame. With each head-in-oval capture, one image is retained, and the oval moves. The user is expected to reposition their head into the oval again. The reason for this is to generate slightly different views, enabling improved facial recognition performance.


Once three images have been captured the enrollment process is completed, and the *status* parameter to the Frame Callback will be 1. The facial recognition gallery will have a new multi-image template generated. The Frame Callback will continue to be called until stopped.

To use an Enroll Video object, one needs to have previously created and initialized their Aureus object. After that:

1. Create the Enroll Video object by calling **CX_CreateEnrollVideo(CX_AUREUS p_aureus, char* msg).** This will return a new Enroll Video object when successful, or NULL if an error occurs.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter will only be valid if an error occurs, containing an error message upon return.

2. Next one starts the Enroll Video playing with **CX_StartEnrollmentStream(CX_ENROLL_VIDEO p_video, cx_uint stream_type, const char* connection_info, char* msg)**. Once the video is streaming, the enrollment processes has begun, and the Frame Callback will start receiving calls.
    a. The 1st parameter is the Enroll Video object returned from **CX_CreateEnrollVideo ()**.
    b. The 2nd parameter is the type of video stream to be processed. Valid values are 1 for USB video, and 2 for an IP camera video stream. Media files are not supported.
    c. The 3rd parameter is the "connection info" string, which varies depending upon the stream type:
        i. If the stream is a USB camera, the string contains the USB video pin
            1. a string like "0"
        ii. If the stream is from an IP camera, the string contains the URL to the IP Camera,
            1. a string like "rtsp:admin:admin@192.168.1.108"
            2. please consult your IP camera documentation for the specifics of the URL required to initiate video stream reception from one's IP camera.
    d. The last parameter is storage for an error message. This only contains valid information when an error occurs.

3. At this point the Frame Callback is receiving frames. One may call **CX_PauseEnrollmentStream(CX_ENROLL_VIDEO p_video, char* msg)** to pause the video and the enrollment process, and afterwards call **CX_UnPauseEnrollmentStream(CX_ENROLL_VIDEO p_video, char* msg)** to resume the video stream, causing the Frame Callback to receive frames once again, and the enrollment process continues where it left off. Both routines return 0 to indicate error, or 1 to declare success.
    a. The 1st parameter to both these calls is the Enroll Video object returned from **CX_CreateEnrollVideo ()**.
    b. The last parameter is storage for an error message. This only contains valid information when an error occurs.
4. When the *status* parameter of the Frame Callback indicates completion or error (or for whatever reason one wants to end the enrollment) one can call **CX_StopEnrollmentStream(CX_ENROLL_VIDEO p_video, char* msg)** to terminate the video stream and enrollment. Note that if *status* is already 1, then the enrollment cannot be abandoned, because it is already completed. This returns 0 to indicate failure, or 1 for success. Once an Enroll Video object has been stopped, to use it again **CX_StartEnrollmentStream()** must be called, which will reinitialize to a new automated enrollment, abandoning any unfinished enrollment left from previous (if any).
    a. The 1st parameter to both these calls is the Enroll Video object returned from **CX_CreateEnrollVideo ()**.
    b. The last parameter is storage for an error message. This only contains valid information when an error occurs.

The above steps 2-4 can be repeated with as many Enroll Video objects as one is licensed.

# 19 Manual Enrollment of New People into the Gallery

One only needs a created and initialized Aureus object for manual enrollment of new people into the facial recognition gallery. Two routines are provided that both generate new enrollments, one for single images and one for multi-image enrollments.

Granted one has one or more images, one may call **CX_AddNewPerson(CX_AUREUS p_aureus, const char* name, CX_RAM_Image* p_im, char* msg)** to do a single image enrollment of a new person. This will return -1 on failure, or the unique ID of the new person enrolled.

- The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
- The 2nd parameter is a string to be used as the "name" or label of the person being enrolled
- The 3rd parameter is the facial image for enrollment.
- The last parameter will only be valid if an error occurs, containing an error message upon return.

If one has multiple images of the same subjects, one may call **CX_AddNewPersonMultiImage(CX_AUREUS p_aureus, const char* name, CX_RAM_Image** p_ims, int n_images, char* msg)** to perform a new multi-image enrollment. In such an enrollment, each image is expected to be the same person, but may represent the individual under different conditions. This returns -1 on failure, or the unique ID of the new person enrolled.

- The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
- The 2nd parameter is a string to be used as the "name" or label of the person being enrolled
- The 3rd parameter is an array of facial images for enrollment.
- The 4th parameter is the number of images to enroll from the array.
- The last parameter will only be valid if an error occurs, containing an error message upon return.

# 20 Using Video Objects to Enroll New People into the Gallery

As Aureus processes video frames, detecting heads, tracking them and so on, it is also maintaining a collection of images for every detected head. This per head collection is sorted by quality. To perform enrollments from Video objects, one should follow these steps:

1. First an Aureus object is created and initialized, and a Video object is created, and the desired detection parameters are set.

2. The Video object is played, which can be used for sequential images, media files, USB or IP camera video streams.

3. With the video stream playing, the Frame Callback is being called. Passed to the Frame Callback of the Video object, the CX_HEAD_LIST contains the CX_HEADs of detected and tracked heads.

4. Still in the Frame Callback, one knows there is at least one active head being processed, and there is a CX_HEAD_LIST holding them. Looping over each CX_HEAD in the passed CX_HEAD_LIST, through a combination of **CX_GetHead()** and **CX_GetNumHeadEnrollmentImages(CX_HEAD p_head, char* msg)** calls, one can track the growth of each CX_HEAD's enrollment suitable image collection. **CX_GetNumHeadEnrollmentImages()** returns -1 on failure, or the current size of the enrollment suitable image collection for the passed CX_HEAD. If this returns > 1, one has the opportunity to allow more images to collect. After 5 images are collected, no more will be added (subject to alteration in future upgrades). Remember the list of each enrollment suitable collection is maintained sorted by quality.
    a. The 1st parameter is the CX_HEAD returned from **CX_GetHead()**.
    b. The last parameter is storage for an error message. This only contains valid information when an error occurs.

5. In situations with a human operator, the operator may need to select the person to enroll from a video stream with multiple detected and tracked heads. Supposing the chosen solution is the operator clicks their mouse on the tracked head of the person to enroll. To accomplish this, one can call **CX_GetHeadInfo(CX_HEAD p_head, CX_HeadInfo* p_data, char* msg)** to obtain video frame specific information about this head. This returns 0 to indicate error, or 1 to indicate success.
    a. The 1st parameter is the CX_HEAD returned from **CX_GetHead()**.
    b. The 2nd parameter is a CX_HEADInfo structure to be filled.
    c. The last parameter is storage for an error message. This only contains valid information when an error occurs.

6. With a valid CX_HEADInfo structure, one can validate the mouse click occurs inside by examining either the normalized head location, or the normalized face location within the video frame:

```
struct CX_HeadInfo {
  // an index referring to the tracked head, after performing tracking
  // this index will remain the same throughout tracked frames
  // this value is undetermined upon detection, it only has meaning after tracking
  cx_int   m_head_id;

  cx_int   m_rows;  // the image size
  cx_int   m_cols;

  // the head location in normalized coordinates with origin at bottom left
  bool    m_head_ok;
  cx_real m_head_bl_x, m_head_bl_y, m_head_tr_x, m_head_tr_y;

  // the face location in normalized coordinates with origin at bottom left
  bool    m_face_ok;
  cx_real m_face_bl_x, m_face_bl_y, m_face_tr_x, m_face_tr_y;

  // a confidence measure with respect to face recognition, higher = better
  // this is constructed from measures of internal Aureus model fitting, 3D pose
  // estimation and face resolution
  cx_real m_confidence;

  // a focus measure with respect to face recognition, higher = better
  // this is only available after tracking and is undefined if
  //  m_has_annotation_set is false.
  cx_real m_focus;

  // the 3D pose of the detected head, this is only available after tracking
  // and is undefined if m_has_annotation_set is false.
  cx_real m_rot_x;
  cx_real m_rot_y;
  cx_real m_rot_z;

  // if true, then the head has an associated annotation set
  // if false then it does not
  bool m_has_annotation_set;

  // the frame number relevant to this head info
  cx_int m_frame_number;

  // a flag that denotes whether the head info data is valid and being tracked for the
current frame
  bool   m_valid;
};
```

Additionally, one could get ambitious and filter the heads by looking at more details beyond the m_confidence, m_focus fields which they are already sorted during collection. Ensuring m_head_ok, m_face_ok, m_has_annotation_set, and m_valid fields are all true, selecting X,Y,Z poses closer 0,0,0, and perhaps retrieving the annotations with XXX and seeking more neutral facial expressions. The CX_HEAD images could be collected with calls to CX_GetRankedHeadDataList(), walking that head data list, retrieving their CX_HEAD_DATA via CX_GetHeadData() and annotations via CX_HeadDataAnnotations() for the additional filtering, and finally use CX_HeadDataImage() and CX_ImageData() to get the image(s)

for enrollment. After all this, a process similar to the last section where **CX_AddNewPerson() / CX_AddNewPersonMultiImage()** is demonstrated for performing a manual enrollment would complete a more custom image selection for enrollment.

7. Demonstrating enrollment from the already collected images, ensuring *m_head_ok* is true on a chosen head, calling **CX_AddNewPersonFromHead(CX_AUREUS p_aureus, const char* name, CX_HEAD p_head, ce_int n_images, char* msg)** will enroll the passed CX_HEAD into the gallery using the CX_HEAD's sorted collection of enrollment suitable images. This returns -1 on failure, or the unique ID of the new person added to the gallery. The number of images parameter, n_images, is a request to attempt a multi-image enrollment with the best n_images from the collection of enrollment suitable images. If fewer images are available than requested, they will all be enrolled. If there are more images than requested, the best n_images will be enrolled.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter is a string to be used as the "name" or label of the person being enrolled
    c. The 3rd parameter is the CX_HEAD for enrollment.
    d. The 4th parameter is the number of images from the collection for that CX_HEAD to use for the enrollment.
    e. The last parameter will only be valid if an error occurs, containing an error message upon return.

As can be seen, the goal of these steps is calling **CX_AddNewPersonFromHead()**, preceded by insuring one has correct and valid data. This process may be repeated, steps 3-7, to perform multiple new person enrollments. The Video object continues to call the Frame Callback as long as not paused, stopped or the stream ends / is terminated.

# 21 Adding a New Image to an existing Gallery Person

Each of the three enrollment methods described above return a unique person ID for the enrolled person. Each person in a facial recognition gallery is represented by a unique ID. Consider the situation where one needs to add additional image(s) to an existing enrolled person. Each addition of one or more images results in an addition facial recognition template of the individual, and does not modify existing facial recognition templates of this person. Aureus provides two methods, a single image addition and a multi-image addition to an existing facial recognition record.

To accomplish a single image addition, one first must have that person's unique ID and a filled out CX_RAM_Image containing the new image of the person receiving the update. With this data, calling **CX_AddImageToPerson(CX_AUREUS p_aureus, int person_id, CX_RAM_Image* p_im, char* msg)** will either return 1 to indicate the image was successfully verified and added to the person's record. Or it may return 0 to indicate an error occurred.

1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
2. The 2nd parameter is the person ID receiving the addition
3. The 3rd parameter is the image to add.
4. The last parameter will only be valid if an error occurs, containing an error message upon return.

To do a multi-image addition, one must have that person's unique ID and an array of filled out CX_RAM_Images containing the new images of the person receiving the update. No more than 10 CX_RAM_Images will be enrolled. With this data, calling **CX_AddImagesToPerson(CX_AUREUS p_aureus, int person_id, CX_RAM_Image** p_ims, int n_images, char* msg)** will either return 1 to indicate at least one image successfully verified, and added to the person's record. Or it may return 0 to indicate an error occurred.

1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
2. The 2nd parameter is the person ID receiving the addition
3. The 3rd parameter is the array of images to add.
4. The 4th parameter is the number of images to enroll.
5. The last parameter will only be valid if an error occurs, containing an error message upon return.

# 22 Reviewing and Deleting Images from an Existing Gallery Person

When images are enrolled and/or added to a person ID in the facial recognition gallery, each image receives a unique image ID for use when referencing the image.

Note that a person ID and a person index are two different things. Where a person ID is a unique identifier for each person, the person index is a value used for fast retrieval of a person's information record.

The images associated with a given person ID include those images passed to Aureus. To review the images associated with a person's ID, and if desired, delete an image from a person's gallery records, one needs to begin with a created and initialized Aureus object with the gallery loaded.

Granted one knows the gallery ID of the person having images reviewed and/or deleted, there are two methods for retrieving the gallery image IDs associated with a person by their ID. Both methods require retrieving a CX_PersonInfo structure describing the person's gallery information.

- The first method of getting the image IDs for a given person starts with calling **CX_GetPersonIndex(CX_AUREUS p_aureus, int person_id, char* msg)** to convert from a person ID to a person index. This returns -1 on failure, or the person index upon success. The person index is literally where in an array of all person IDs in the gallery, that person's ID is located.
    1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    2. The 2nd parameter is the person 's unique gallery ID
    3. The last parameter will only hold valid info if an error occurs, containing the error message upon return.

    In possession of a person's gallery index, calling **CX_GetPersonInfo(CX_AUREUS p_aureus, int index, CX_PersonInfo* person_info, char* msg)** fills in the CX_PersonInfo structure passed by pointer.

    1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    2. The 2nd parameter is the person 's gallery index, for example, if the gallery contains 3 people, the index can be 0,1 or 2.
    3. The 3rd parameter is a pointer to a CX_PersonInfo structure to be filled

4. The last parameter will only hold valid info if an error occurs, containing the error message upon return.

- The second method of getting image IDs is by calling **CX_FillPersonInfo(CX_AUREUS p_aureus, CX_PersonInfo* person_info, char* msg)** with the CX_PersonInfo structure containing a valid person ID, fills in the rest of the CX_PersonInfo structure, including the array of image IDs for that person. This returns 0 for failure or 1 indicating success. Note this method does not require converting from person ID to person index; the person ID is used directly by this method,
    1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    2. The 2nd parameter is a pointer to a **CX_PersonInfo** structure to be filled, with m_person_id initialized with the id of the person whose data will be used to complete the **CX_PersonInfo** fields.
    3. The last parameter will only hold valid info if an error occurs, containing the error message upon return.

With a filled out CX_PersonInfo structure, its cx_int array m_image_ids[] field has the gallery IDs of the images of this person, and m_num_images tell the number of images IDs in that array.

To review the images, calling **CX_GetGalleryImage(CX_AUREUS p_aureus, int image_id, char* msg)** will return a CX_IMAGE pointer holding the requested image, or NULL on failure. Note that the CX_IMAGE returned is allocated on the Heap, and **CX_FreeImage()** must be called to free the CX_IMAGE when its use is complete.

1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
2. The 2nd parameter is the image ID to generate an CX_IMAGE object.
3. The last parameter will only hold valid info if an error occurs, containing the error message upon return.

Calls to **CX_GetGalleryImage()** return a full size image. When generating a GUI, thumbnail sized images may be more useful. Calling **CX_GetImageThumbnail(CX_AUREUS p_aureus, int image_id, char* msg)** will return a CX_IMAGE pointer holding the thumbnail version of the same image, or NULL on failure. Note that the CX_IMAGE returned is allocated on the Heap, and **CX_FreeImage()** must be called to free the CX_IMAGE when its use is complete.

1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
2. The 2nd parameter is the image ID to generate an CX_IMAGE object.
3. The last parameter will only hold valid info if an error occurs, containing the error message upon return.

A CX_IMAGE is one of the Aureus objects defined as a void pointer. To get meaningful image data from a CX_IMAGE, one calls **CX_ImageData(CX_IMAGE p_image, cx_uint* rows, cx_uint* cols, char* msg)** and receives back a cx_byte pointer to the pixel array, or NULL to indicate failure. The image pixels are always row major, RBGA order, with the origin at bottom left. Do not free/delete the returned pointer.

1. The 1st parameter is the CX_IMAGE object returned from **CX_ GetGalleryImage ()**.
2. The 2nd parameter is a pointer to a cx_uint that will be set to the number of rows in the image
3. The 3rd parameter is a pointer to a cx_uint that will be set to the number of columns in the image.
4. The last parameter will only hold valid info if an error occurs, containing the error message upon return.

Once one has a cx_byte array of RGBA pixels containing an image, it can be displayed for visual confirmation and/or incorporated into the GUI of one's application.

To remove an image by its ID, call **CX_DeleteImage(CX_AUREUS p_aureus, int image_id, char* msg)**. Returning 0 for failure, or 1 on success, this additionally checks if the removed image is the only image representing a person, and if so removes the person from the gallery.

1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
2. The 2nd parameter is the image ID being deleted.
3. The last parameter will only be valid if an error occurs, containing an error message upon return.

# 23 Deleting a Person from the Gallery

Supposed after reviewing the images associated with a given person ID (described last section), one wants to delete a person from the gallery. In these situations, deleting a person entirely from the facial recognition gallery can be accomplished with a call to **CX_DeletePerson(CX_AUREUS p_aureus, int person_id, char* msg)**. Returning 0 to indicate error, or returning 1 for success, this will delete all images and the person record associated with that person ID.

1. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
2. The 2nd parameter is the person ID being deleted.
3. The last parameter will only be valid if an error occurs, containing an error message upon return.

# 24 Using Aureus to generate Facial Recognition Templates

To generate facial recognition templates, one first needs a version of Aureus licensed for facial recognition. Granted one is licensed, Aureus provides two direct methods for generating templates, one automated and the other manual. Additionally, Aureus indirectly generates templates anytime a new person is enrolled, as well as anytime one or more images are added to a person in the gallery.

The automated method simply needs to call **CX_SetGenerateTemplatesFlag(CX_VIDEO p_video, int generate_templates, char* msg)** with the second parameter set to 1 when initializing the Video object being used. The use of this routine is already described in the section detailing the creation and configuration of an Aureus Video object. When template generation and gallery ranking are both enabled, as heads are detected and tracked through the use of a Video object, templates are being automatically generated from these heads as is necessary to generate the ranked FR results against the templates for people already in the gallery.

1. The 1st parameter is the Video object returned by **CX_CreateVideo()**.
2. The 2nd parameter is the generate_templates flag, set to 1, enabling this feature.
3. The last parameter will only be valid if an error occurs, containing an error message upon return.

Once the Video object is playing with template generation and facial recognition active, Aureus is automatically generating templates and comparing them against the gallery. The remainder of this bullet item is retrieval of those generated templates.

As the Frame Callback receives calls, it is receiving a CX_HEAD_LIST each call. Walking the list of detected and tracked heads in CX_HEAD_LIST with **CX_GetHead(),** each CX_HEAD can have its CX_HEAD_DATA_LIST requested by using **CX_GetRankedHeadDataList(CX_HEAD p_head, char* msg).** This returns NULL on error, or a CX_HEAD_DATA_LIST when successful. A CX_HEAD_DATA_LIST is a list of CX_HEAD_DATA objects. In this situation, the CX_HEAD_DATA_LIST represents the ranked facial recognition results of the CX_HEAD against the gallery.

1. The 1st parameter is the CX_HEAD object returned by **CX_GetHead()**.
2. The 2nd parameter is the generate_templates flag, set to 1, enabling this feature.
3. The last parameter will only be valid if an error occurs, containing an error message upon return.

One can get the size of the CX_HEAD_DATA_LIST by calling **CX_HeadDataListSize(CX_HEAD_DATA_LIST p_head_data_list, char* msg)**. That returns -1 on failure or the number of elements in CX_HEAD_DATA_LIST.

1. The 1st parameter is the CX_HEAD_DATA_LIST returned by **CX_GetRankedHeadDataList()**.
2. The last parameter will only be valid if an error occurs, containing an error message upon return.

Then **CX_GetHeadData(CX_HEAD_DATA_LIST p_head_data_list, int index, char* msg)** will return a pointer to the requested CX_HEAD_DATA.

1. The 1st parameter is the CX_HEAD_DATA_LIST returned by **CX_GetRankedHeadDataList()**.
2. The 2nd parameter is the index to CX_HEAD_DATA being requested.
3. The last parameter will only be valid if an error occurs, containing an error message upon return.

With a CX_HEAD_DATA, calls to **CX_HeadDataFRTemplate(CX_HEAD_DATA p_head_data, int* p_template_size, char* msg)** will either return NULL or the cx_byte array of the generated template associated with this CX_HEAD. Do not free/delete the returned pointer.

4. The 1st parameter is the CX_HEAD_DATA object returned by **CX_CreateVideo()**.
5. The 2nd parameter is a pointer to an int, which will contain the size of the returned template byte array.
6. The last parameter will only be valid if an error occurs, containing an error message upon return.

With the template in hand, it may be stored (by copy), and used for facial recognition comparisons either directly or via a copy.

- The manual method of creating facial recognition templates has two variations: creating a single image template, and creating a multi-image template.
    1. When creating a single image template, one first needs to call **CX_GetTemplateSize(CX_AUREUS p_aureus, char* msg)** to get the number of bytes the active FR Engine requires for template storage. This returns -1 on failure, or the number of cx_bytes needed to store a single template for the active FR Engine.
        a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
        b. The last parameter will only be valid if an error occurs, containing an error message upon return.

Next one calls **CX_GenerateTemplate(CX_AUREUS p_aureus, CX_RAM_Image* p_im, CX_DetectionParams fdp, cx_byte* p_template, char* msg)** to generate the template using the active FR Engine. This returns 0 to indicate failure, or it returns 1 on success. When successful, the p_template parameter will contain the generated template. Be sure that m_template points to enough data for the template to be stored.

a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
b. The 2nd parameter is a CX_RAM_IMAGE containing the facial image to use for the template
c. The 3rd parameter is a filled out CX_DetectionParams structure.
d. The 4th parameter is storage for the generated template, with size via **CX_GetTemplateSize()**
e. The last parameter will only be valid if an error occurs, containing an error message upon return.

2. When creating a multi-image template, one needs an array of CX_RAM_Images. The images must all be of the same person. Aureus will generate a unique template for each image given, as well as will create a combined template from the images. During validation of the images, one or more may fail validation; if only one passes validation, a template will be generated, but no combined template. If during validation, more than one image passes validation, a combined template will be generated from those images that passed. Calling **CX_GenerateMultiTemplate(CX_AUREUS p_aureus, CX_RAM_Image** p_ims, int n_images, CX_DetectionParams fdp, char* msg)** will return NULL upon failure, or will return a pointer to a CX_MultiFRTemplate structure when successful. The m_combined_exists field of the CX_MultiFRTemplate will be true when a multi-image combined template has been created. The multi-image combined template will be the last one in the array of templates with the CX_MultiFRTemplate. Note that **CX_FreeMultiFRTemplate()** must be called with the returned CX_MultiFRTemplate when its use is complete.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter is a CX_RAM_IMAGE containing the facial image to use for the template.
    c. The 3rd parameter is a filled out CX_DetectionParams structure.
    d. The 4th parameter is storage for the generated template, with size via **CX_GetTemplateSize().**
    e. The last parameter will only be valid if an error occurs, containing an error message upon return.

# 25 Performing Manual 1:1 Matching of Templates

Given one needs to compare two images for being of the same individual, without a facial recognition gallery. The comparison could be from freshly generated facial recognition templates, or a comparison against a stored template during an identification process with a live video stream. Either way, the comparison is between templates. The previous section details methods for creating facial recognition templates from images either by creating them from images or obtaining them live from processed video streams.

- Granted one has an Aureus environment created and configured, one calls **CX_MatchFRtemplates(CX_AUREUS p_aureus, cx_byte* p_t1, int t1_size, cx_byte* p_t2, int t2_size, char* msg)** to calculate the match score between two templates. This returns negative to indicate failure, or the match score between the two templates. The two templates must have been generated with the same FR Engine, and it must be the active FR Engine selected into Aureus.
    a. The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
    b. The 2nd parameter is a pointer to a cx_byte array containing the first template.
    c. The 3rd parameter is the byte size of the first template.
    d. The 4th parameter is a pointer to a cx_byte array containing the second template.
    e. The 5th parameter is the byte size of the second template.
    f. The last parameter will only be valid if an error occurs, containing an error message upon return.

The match score is a normalized confidence value, 0.0 meaning no confidence the two templates are of the same person, and 1.0 meaning there is high confidence the two templates represent images of the same person. Typicel verified match scores are in the region of 0.75 to 1.0.

# 26 Performing Image Comparisons against the Gallery

Given one needs to compare an image against the active facial recognition gallery, one needs an Aureus environment created and configured with the facial recognition gallery loaded, and a CX_RAM_Image holding the facial image for comparison.

Calling **CX_ApplyFR(CX_AUREUS p_aureus, CX_RAM_Image* p_im, CX_RankItem* p_rank_results, int array_size, char* msg)** will return -1 to indicate error, or the number of CX_RankItem structures written into the p_rank_results array. Before calling, one needs the CX_RankItem array pre-declared, as one is passing the array in to be filled with the best ranked array_size matches.

a) The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
b) The 2nd parameter is a CX_RAM_Image to be compared against the gallery.
c) The 3rd parameter is an array of CX_RankItem.
d) The 4th parameter is the size of the CX_RankItem array.
e) The last parameter will only be valid if an error occurs, containing an error message upon return.

The filled CX_RankItem array contains person ids, image ids, and match scores, delivered in sorted order by match score.

```
struct CX_RankItem {

  // the person ID from the gallery, this means the actual person: so in a list of CE_RankItem's

  // there might be more than one element having the same person_id because there were more

  // than one image per person; alternatively, a list of CE_RankItem's might contain rank info

  // combined from many processed images, hence more than one element might have the same
person_id
```

```
  cx_int  m_person_id;

  // this is the name of the person defined by the unique id m_person_id
  char    m_person_name[1024];

  // this is the image id from the gallery (same rules as person_id above wrt multiple
processed images)
  cx_int  m_image_id;

  // this is the FR match score with range from zero (no match) to 1 (perfect match)
  cx_real m_score;

};
```

# 27 UPDATING A GALLERY WITH A NEW FR ENGINE

From time to time, when developing or having Aureus applications supporting multiple FR engines, the need occurs when a gallery has had a number of enrollments occur while one FR engine was active, and now a different FR engine is active. The enrollments with the other FR engine had FR templates generated with the prior FR engine, but the current active FR engine has no such templates for those newly enrolled people. Or a new FR engine has just been installed, and the gallery needs to generate templates for the people compatible to its requirements.

Calling **CX_UpdateFR(CX_AUREUS p_aureus, char* msg)** accomplishes the necessary template generations such that all FR engines have templates for all the people in the gallery. This returns -1 when the Aureus pointer is bad; it returns 0 when no templates were generated at all; returns 1 to indicate all necessary templates were generated, and it returns 2 to indicate a partial success – some templates were generated, but some failed to generate. **Note that this is one Aureus call where the error message can be very large, as large as one message per image in the gallery!** Due to the possibility of such a large error message, allocating an enormous error message buffer may be recommended. We additionally recommend working with a smaller version of one's gallery for tests of new FR engines for this reason.

a) The 1st parameter is the Aureus object returned from **CX_CreateAureus()**.
b) The last parameter will only be valid if an error occurs, containing an error message upon return.

# 28 AUREUS OBJECTS

**Aureus Objects:** Aureus provides and maintains a series of "objects" – managed data and functions collectively referenced and operated as single conceptual entities. These objects are created with Aureus functions, and returned to the caller as void pointers. Using an Aureus object consists of creating one through the appropriate Aureus function, and then passing it (the void pointer to it) to the Aureus functions one desires operations to be applied through that object.

A list of Aureus objects:

**CX_AUREUS**: this is the top level Aureus instance which must be created and initialized in order to create and operate any other Aureus Object. One can consider this to be Aureus itself, the master root object which all Aureus functionality is derived. Use CX_CreateAureus() to create, CX_Initialize() to activate for use, CX_FreeAureus() to delete.

**CX_VIDEO**: an object representing the controller for a video stream; all video processing, as well as sequential image processing, requires an Aureus Video object. Through this object video streams are played, paused, stopped, and passed to Aureus for facial recognition related operations. Each video stream processed simultaneously requires its own Video object. One may create as many Video objects as one is licensed across multiple processes. The licensed number of video objects is *per machine* not per Aureus process. Use CX_CreateVideo() to create, CX_VIDEO objects should not be freed or deleted.

**CX_HEAD**: an object containing information about a detected/tracked head. Use CX_GetHeadInfo() to retrieve the information contained within a CX_HEAD.

**CX_HEAD_LIST**: an object representing a list of CX_HEADs, typically returned by calls to CX_ProcessFrame(), CX_ProcessFrameFromDetectedHeads() and received by CX_VIDEO Frame Callbacks. Use CX_GetHead() to retrieve individual CX_HEADs from a CX_HEAD_LIST.

**CX_HEAD_DATA**: an object representing one frame's / image's worth of information about a tracked CX_HEAD. These are collected across multiple frames and maintained sorted by quality and facial recognition match score.

**CX_HEAD_DATA_LIST**: an object representing a list of information about a tracked CX_HEAD, spanning multiple frames / images, and sorted by quality and facial recognition match score.

**CX_ANNOTATION**: an object representing a set of points, defining a facial feature's characteristics

**CX_ANNOTATION_SET**: an object representing a list of CX_ANNOTATIONs.

**CX_IMAGE**: an object representing an image internal to Aureus. Use CX_ImageData() to retrieve the image pixels, and the numbers of rows and columns of pixels in the image. CX_IMAGES are always row major, are always RGBA, and have their origin located at the bottom left. Use CX_CopyImage() to perform a fast conversion from one image type to another (including origin placement).

**CX_ENROLL_VIDEO:** an object representing a video stream for USB or IP camera video streams with additional Aureus functions related to the enrollment of new people into the facial recognition gallery.

# 29 AUREUS STRUCTURES

Aureus uses a series of C structs (multiple field structures) to contain information and parameter sets passed to and from Aureus. A list of these structures is itemized below:

**CX_DetectionParams structure:** the structure describing head detection parameters used by Aureus.

```
struct CX_DetectionParams {
  cx_real    m_top;
  cx_real    m_left;
  cx_real    m_height;
  cx_real    m_width;
  cx_real    m_min_height_prop;
  cx_real    m_max_height_prop;
};
```

The top, left, width and height fields describe the region of interest used by Aureus when detecting and tracking heads. All values in a CX_DetectionParams structure are normalized, meaning their valid values range from 0.0 to 1.0. An image or video frame origin is top left, meaning the top, left corner is coordinate 0.0, 0.0. Values for the entire image or video frame would have top=0.0, left=0.0, width=1.0, and height=1.0. The m_min_height_prop and m_max_height_prop fields define the min and max proportions of the image / video frame which a detected head could occupy. This is the range of whole head heights, not face heights.

**CX_FRTemplate structure**: the structure used as a container for a single image facial recognition template.

```
struct CX_FRTemplate
{
  cx_int   m_size;      // the size of the FR template
  cx_byte* mp_template; // the FR template data
};
```

**CX_Head_Info structure:** the structure describing a detected/tracked head. Use **CX_GetHeadInfo()** to fill in a CX_HeadInfo structure with the information associated with a passed CX_HEAD object.

```
struct CX_HeadInfo {
  // an index referring to the tracked head, after performing tracking
  // this index will remain the same throughout tracked frames
  // this value is undetermined upon detection, it only has meaning after
tracking
  cx_int   m_head_id;
```

```
  // the image size
  cx_int   m_rows;
  cx_int   m_cols;

  // the head location in normalized coordinates with origin at bottom left
  bool    m_head_ok;
  cx_real m_head_bl_x, m_head_bl_y, m_head_tr_x, m_head_tr_y;

  // the face location in normalized coordinates with origin at bottom left
  bool    m_face_ok;
  cx_real m_face_bl_x, m_face_bl_y, m_face_tr_x, m_face_tr_y;

  // a confidence measure with respect to face recognition, higher = better
  // this is constructed from measures of internal Aureus model fitting, 3D
pose
  // estimation and face resolution
  cx_real m_confidence;

  // a focus measure with respect to face recognition, higher = better
  // this is only available after tracking and is undefined if
  // m_has_annotation_set is false.
  cx_real m_focus;

  // the 3D pose of the detected head, this is only available after tracking
  // and is undefined if m_has_annotation_set is false.
  cx_real m_rot_x;
  cx_real m_rot_y;
  cx_real m_rot_z;

  // if true, then the head has an associated annotation set
```

```
        // if false then it does not
        bool m_has_annotation_set;

        // the frame number relevant to this head info
        cx_int m_frame_number;

        // a flag that denotes whether the head info data is valid
        // and being tracked for the current frame
        bool   m_valid;
      };
```

**CX_MultiFRTemplate structure:** the structure used to contain multiple facial recognition templates.

```
struct CX_MultFRTemplate {
  int            m_n_templates;     // number of templates in this structure
  CX_FRTemplate* mp_templates;      // the templates
  bool           m_combined_exists; // if true then the last template is a combined
template
};
```

**CX_PersonInfo structure**: the structure used to pass information about the facial recognition gallery entries associated with a person in the gallery.

```
struct CX_PersonInfo {         // this struct is used to pass gallery information
about a single person
  cx_int m_person_id;          // the unique person identifier
  char   m_person_name[1024];  // the name of the person in the gallery
  cx_int m_num_images;         // number of images associated with the person in
the gallery
  cx_int m_image_ids[1024];    // the unique image id's, one for each associated
image
};
```

**CX_RAM_Image structure:** the structure for passing images in RAM to Aureus. The cxutils.cpp function **LoadImageFromDisk(const char* path, CX_RAM_Image& im)** will read a jpeg image into this structure.

```
struct CX_RAM_Image {
  cx_byte*    mp_pixels;
  cx_uint     m_rows;
  cx_uint     m_cols;
  cx_uint     m_type;    // 0=RGB, 1=RGBA, 2=Gray, 3=BGR, 4=BGRA
  cx_uint     m_origin;  // 0=top left, 1=bottom left
};
```

**CX_RankItem structure**: the structure used to describe a facial recognition comparison result. Typically in arrays, describing a gallery comparison.

```
struct CX_RankItem {
  // the person ID from the gallery, this means the actual person: so in a list of CE_RankItem's
  // there might be more than one element having the same person_id because there were more
  // than one image per person; alternatively, a list of CE_RankItem's might contain rank info
  // combined from many processed images, hence more than one element might have the same person_id
  cx_int  m_person_id;

  // this is the name of the person defined by the unique id m_person_id
  char    m_person_name[1024];

  // this is the image id from the gallery (same rules as person_id above wrt multiple processed images)
  cx_int  m_image_id;

  // this is the FR match score with range from zero (no match) to 1 (perfect match)
  cx_real m_score;
```

```
};
```

**CX_ResultsSettings structure:** the structure used to control how results are saved during the processing of Video objects.

```
#define AUREUS_TAG_SIZE 50

struct CX_ResultsSettings {

  // if set then xml files will be saved to the given folder, one xml file per tracked
head

  bool m_save_xml_results;      // default = false

  char m_save_xml_folder[1024]; // default = InstallFolder/POST


  // if true then any saved xml files will be removed after a

  // certain time period (circa 5s, subject to change)

  bool m_remove_files;          // default = true


  // if set then the xml files will be posted to the given url

  bool m_post_xml;              // default  = false

  char m_post_url[1024];        // default = http://localhost:81


  // if set then the tracked head image and it's highest ranked matched image

  // will be saved to the given folder

  bool m_save_images;               // default = false

  char m_save_images_folder[1024];  // default = InstallFolder/Results/Images


  bool m_save_only_verified;         // if true then only verified matches will be
saved (default=true)


  // if xml files are saved, the contents are controlled by the following flags

  // each one has a tag that will be used in the xml file, you can edit/alter the
tags

  // to suit your own schema

  char m_root_tag[AUREUS_TAG_SIZE]; // defaults to CustomerInfoRequest
```

```
  bool m_person_name;               // the name of the rank1 matched person
  char m_person_name_tag[AUREUS_TAG_SIZE]; // defaults to externalId

  bool m_person_id;                 // the unique DB ID of the rank1 matched person
  char m_person_id_tag[AUREUS_TAG_SIZE]; // defaults to person_id

  bool m_processed_frames;          // the number of processed frames for the tracked
head
  char m_processed_frames_tag[AUREUS_TAG_SIZE]; // defaults to ProcessedFrames

  bool m_head_id;                   // the head id
  char m_head_id_tag[AUREUS_TAG_SIZE]; // defaults to head_id

  bool m_stream_type;               // a string (MediaFile, USBcamera, IPcamera)
  char m_stream_type_tag[AUREUS_TAG_SIZE]; // defaults to stream_type

  // a string containing the connection info:
  // 0 for USB pin zero, or the IP camera url, or the full path and file name of a
media file
  bool m_stream_connection_info;
  char m_stream_connection_info_tag[AUREUS_TAG_SIZE]; // defaults to conn_info

  bool m_stream_index;              // the index of the video stream
  char m_stream_index_tag[AUREUS_TAG_SIZE]; // defaults to stream_index

  bool m_verification_threshold;    // the verification threshold used at the time
  char     m_verification_threshold_tag[AUREUS_TAG_SIZE];     //     defaults     to
VerificationThresh

  bool m_frame_number;              // the frame number at which the result occurred
  char m_frame_number_tag[AUREUS_TAG_SIZE]; // defaults to FrameNumber
```

```
  bool m_utc_time;                  // coordinated universal time
  char m_utc_time_tag[AUREUS_TAG_SIZE]; // defaults to utcTime

  bool m_ranked_results;
  cx_uint m_ranked_results_n;       // the max number of ranked FR results (default
= 1)
  char m_ranked_results_tag[AUREUS_TAG_SIZE]; // defaults to NumOfRankedResults

  // for each ranked result
  bool m_matched_person_name;       // the name of the ranked matched person
  char m_matched_person_name_tag[AUREUS_TAG_SIZE]; // defaults to externalId

  bool m_matched_person_id;         // the unique DB ID of the ranked matched person
  char m_matched_person_id_tag[AUREUS_TAG_SIZE]; // defaults to person_id

  bool m_matched_image_id;          // the unique DB ID of the ranked matched image
  char m_matched_image_id_tag[AUREUS_TAG_SIZE]; // defaults to image_id

  bool m_date_time_stamp;           // a date and time stamp
  char m_date_time_stamp_tag[AUREUS_TAG_SIZE]; // defaults to DateTime

  bool m_confidence_measure;         // confidence measure of the tracked person's
image (higher the better)
  char m_confidence_measure_tag[AUREUS_TAG_SIZE]; // defaults to Confidence

  bool m_focus_measure;              // focus measure of the tracked person's image
(higher the better)
  char m_focus_measure_tag[AUREUS_TAG_SIZE]; // defaults to Focus

  bool m_eye_positions;            // the eye locations in the tracked person's image
  char m_eye_positions_tag[AUREUS_TAG_SIZE]; // defaults to EyePositions
```

```
  bool m_face_box;             // the detected face box in the tracked person's image
  char m_face_box_tag[AUREUS_TAG_SIZE]; // defaults to FaceBox

  bool m_match_score;          // the score between the two matches
  char m_match_score_tag[AUREUS_TAG_SIZE]; // defaults to MatchScore

  bool m_matched_status;        // VERIFIED if the score >=verification threshold,
UNVERIFIED if not
  char m_matched_status_tag[AUREUS_TAG_SIZE]; // defaults to matchStatus

  bool m_tracked_image;            // the tracked person's image
  char m_tracked_image_tag[AUREUS_TAG_SIZE]; // defaults to capturedImage

  bool m_matched_person_thumbnail; // the thumbnail of the person in the DB which
resulted in the match
  char    m_matched_person_thumbnail_tag[AUREUS_TAG_SIZE];    //    defaults    to
MatchedPersonImage

  bool m_matched_thumbnail;        // the DB thumbnail image to which it was matched
  char m_matched_thumbnail_tag[AUREUS_TAG_SIZE]; // defaults to MatchedImage
};
```

# 30 Example Programs

Whilst Aureus contains many sophisticated algorithms under the hood it is simple to use. Simply create Aureus, ask Aureus to create a video stream and tell Aureus to start the video stream. How you choose to use/manipulate the processed results is up to you. The example programs demonstrate various usages.

***A simple camera example:***

Once you have created Aureus and created a video object. You can simply tell Aureus what information you wish to save and where then tell Aureus to start the camera and Aureus will do the rest for you. You can also tell Aureus that you want the results positing to a url of your choice.

***A simple camera example using the frame callback:***

Once you have created Aureus and created a video object you can provide the video object with your own frame callback function. Aureus will then call this function for every processed frame. It will pass in a list of processed CX_HEAD objects as well as the frame image. A CX_HEAD object represents a tracked person; it contains all the data necessary to enable a system to utilize facial recognition for whatever purpose.

Briefly, each CX_HEAD object contains a current CX_HEAD_DATA object and a ranked list of CX_HEAD_DATA objects. Each CX_HEAD_DATA object contains information about the head location, a tokenized head image, a set of landmark points, a confidence score, a focus measurement, a facial recognition template and ranked facial recognition results. The ranked list of CX_HEAD_DATA objects stores the last n *best* results for the tracked CX_HEAD.

# 31 Code Example: Processing Sequential Images, Simultaneous Detection & Tracking

Note: To perform simultaneous detection and tracking in a sequential series of images, one must have previously created an Aureus object, initialized it, created a Video object, and defined the detection parameters to be used.

Note: the below code example is detailed in the section Simultaneous Tracking and Detection of Sequential Images.

Note: the below code is intended to illustrate the process, and will not compile. For example, the routine PrintHeadData() called within the example is not given. The Aureus_Tracking example code project provided with the installation contains this code.

```
void ProcessSequentialFrames(CX_VIDEO p_video, const char* dir)

{

  char msg[1204];


  std::vector<std::string> fnames;

  CE_FindFiles(fnames, dir, "*.jpg");
```

```
printf("Processing %d images from %s\n", fnames.size(), dir);

for (int i = 0; i < fnames.size(); i++)
{
  string& fname = fnames[i];
  CX_RAM_Image im;
  if (LoadImageFromDisk(fname.c_str(), im))
  {
    printf("Loaded frame %d\n", i);

    int use_face_detector = 1;
    CX_HEAD_LIST p_head_list = CX_ProcessFrame(p_video, &im, i, use_face_detector, msg);

    if (!p_head_list)
       printf("Error at frame num %d : %s\n", i, msg);
    else
    {
      int n_heads = CX_GetHeadListSize(p_head_list, msg);
      if (n_heads < 0)
         printf("Frame %d Error: %s\n", frame_number, msg);
      else
      {
         printf("FRAME %d, Tracking %d HEADS\n", frame_number, n_heads);
         for (int j = 0; j < n_heads; j++)
         {
           CX_HEAD p_head = CX_GetHead(p_head_list, j, msg);
           if (!p_head)
              printf("Failed to get head %d of %d heads: %s\n", j, n_heads, msg);
           else
           {
             PrintHeadData(frame_number, j, p_head);
           }
         }
       }
```

```
      }


      // free mem allocate for image pixels
      DeleteImagePixels(im);
    }
    else
    {
      printf("FAILED TO LOAD FRAME %d\n", i);
    }
  }
}
```

# 32 Code Example: Processing Sequential Images in 2 Passes: First Detection, then Tracking

Note: To perform separate detection of heads in one pass and tracking of the detected heads in a second pass over a set of sequential images, one must have previously created an Aureus object, initialized it, created a Video object, and defined the detection parameters to be used with that Video object.

Note: the below code example is detailed in the section Two pass, Detection First Pass and Tracking Second Pass of Sequential Images. The Aureus Tracking example code project provided with the installation contains this code.

```
void ProcessFramesSeparateDetection(CX_VIDEO p_video, const char* dir, bool print_details = true)
{
  int i;
  char msg[1204];
  std::vector<CX_RAM_Image> frames;
  LoadFrames(dir, frames, print_details);

  printf("Processing sequential images frame from folder:\n%s\n", dir);

  printf("Detecting heads in %d images\n", frames.size());

  std::vector<CX_HeadInfo*> heads;
  std::vector<int> num_heads;
  int max_heads = 20;
  CX_HeadInfo* tmp_head = NULL;

  /////////////// Detection ////////////////////////////////////
  // step through all frames detecting heads
  for (i = 0; i < frames.size(); i++)
  {
    // allocate mem for head detections
    heads.push_back(tmp_head);
    heads[i] = new CX_HeadInfo[max_heads];
    int use_face_detector = 1;

    int rval = CX_DetectHeads(p_video, &frames[i], i, heads[i], max_heads, use_face_detector, msg);
```

```
    // catch any errors
    if (rval < 0)
    {
      printf("Image %d error: %s\n", i, msg);
      num_heads.push_back(0); // save zero heads because there was an error
    }
    else
    {
      if (print_details) printf("Image %d detected %d heads\n", i, rval);
      num_heads.push_back(rval); // save number of detected heads
    }
  }

  ////////////////// Tracking ///////////////////////////////
  // now step through them applying tracking
  printf("Tracking heads in %d images\n", frames.size());
  for (i = 0; i < frames.size(); i++)
  {
    CX_HEAD_LIST p_head_list =
                CX_ProcessFrameFromDetectedHeads(p_video, &frames[i], i, heads[i], num_heads[i], msg);

    if (!p_head_list)
         printf("Error at frame num %d : %s\n", i, msg);
    else PrintHeadListData(i, p_head_list, print_details);
  }

  // clean up: delete allocated head info data
  for (i = 0; i < frames.size(); i++) {
    if (heads[i]) delete[] heads[i];
    heads[i] = NULL;
  }

  // cleanup: delete the images
  DeleteFrames(frames);
}
```

# 33 CODE EXAMPLE: A BASIC FRAME CALLBACK

Note: the below code example is described where video stream processing describes Frame Callbacks.

```
void FrameCallBack( CX_HEAD_LIST p_head_list,
                    cx_uint      head_list_size,
                    cx_byte*     p_pixels,
                    cx_uint      rows,
                    cx_uint      cols,
                    cx_uint      frame_num,
                    void*        p_object )
{
  // just printing the number of heads being tracked

  char msg[1204];

  int n_heads = CX_GetHeadListSize( p_head_list, msg );

  if (n_heads < 0)
     printf( "Frame %d Error: %s\n", frame_num, msg );
  else
  {
     printf( "FRAME %d, Tracking %d HEADS rows=%d cols=%d\n",
             frame_num, n_heads, rows, cols );
  }

}
```

# 34 CODE EXAMPLE: A BASIC STREAM TERMINATED CALLBACK

Note: the below code example is described where video stream processing describes unexpected stream terminated callbacks. These are used by both Video objects and Enroll Video objects.

```
void UnexpectedTerminationCallBack( cx_int      stream_type,
                                    const char* connection_info,
                                    void*       p_object)
{
  fprintf( stderr, "Unexpected stream termination, stream type = %d, connection_info = %s\n",
           stream_type, connection_info );
}
```