Bellus3D ARC Developer APIs

Overview

The Bellus3D ARC System involves a central Host service connected to one or more ARC cameras to capture and generate a 3D model mesh.

Developers may create their own applications to communicate with the ARC Host (and indirectly through the cameras) via this ARC Developer API, using Websocket as a communication interface.

Glossary

- **Camera**: A Bellus3D ARC (aka B3D4) camera
- **Camera Position**: Position of a camera in a Bellus3D ARC station. Enum values are "c", "l2", "l1", "r1", "r2", "u1", "d1", for "center", "left 2", "left 1", "right 1", "right 2", "top", "bottom" camera respectively.
- **Station**: A grouping of Cameras that forms the ARC capturing system. A station can have from one to seven Cameras. Stations can be created by the Station Configuration APIs.
- **Layout Name**: Name of ARC station layout. Enum values are "single", "horz3", "t4", "cross7", mapping to station names of "ARC1", "ARC3", "ARC4", "ARC7" respectively.
- **Host Server**: A local Websocket server that acts as the Host to ARC-compliant Apps using our ARC **Developer API**. The ARC Software may also create servers for cameras and the ARC Scan App, but Developers may only care about the Developer API server.
- **Host Service**: The ARC Host Server running as an OS service. This runs headless in the background and is automatically started with the OS boots up.
- **ARC Client**: A client of the **Developer API**.
- **Recording Buffer:** A burst recording on a camera, which may be further processed in the camera into depth images, and other processed data. Note that this concept is only used in the multiple camera still capture mode.
- **Scan**: An aggregation of recording buffers across one or more cameras. Once received from the camera(s), the Host can perform additional processing to create a Head Model and other intelligent objects.
- **Head Model**: A 3D head model including surface mesh and texture map.

Host Installation

The Host is distributed as a single windows installer. However, the installer (default name Bellus3D_ARC_Host_version.exe) provides two different ways to install the ARC host.

The Default Install Method

The default installation is triggered by simply double clicking the installer. In this way, the ARC will be installed as a regular application. After the installation, a desktop shortcut to open the ARC ScanApp will be generated.

When a user attempts to open the ScanApp through the desktop shortcut, there will be several things going on under the hood. First of all, the ARC App launcher will be started, then it checks if the ARC Host is in good working order or not, if not, it auto starts the ARC Host and waits for it to be ready. When everything is all set, the launcher will finally open the Scan App.

For developers, the idea is that we do not always assume the ARC Host is in working order at the time we want to use it, for example, when the computer just booted up. The correct thing to do is to execute a command provided by the ARC package to start up the ARC Host before a DEV API connection can be established. **Also note that all the following Host commands should NOT run in system admin permission. This is a change we made since version 1.14.0, before which we explicitly required admin permission.**

Suggest the ARC software is installed at it's default installation path:

```
C:\Program Files\Bellus3D\ARCHost
```

The command to start the ARC Host will be:

```
"C:\Program Files\Bellus3D\ARCHost\Bellus3D ARC Host.exe"
--action=start-host-and-wait
```

The starting of the ARC Host normally takes up to 60 seconds depending on the computer specifications, and as it appears like in the command action name, this command will not finish until the Host is completely ready. If the Host is already ready for use at the time you execute this command, it will simply verify the ARC status and finish in a much shorter time.

Developers may want to restart the host on some other occasions, for example, when the ARC Host configuration file is changed, a restart is required to make the changes live. To restart the ARC host, we need to stop the host first, followed by a fresh start with the command which we just introduced above.

The command to stop the current running ARC Host:

```
"C:\Program Files\Bellus3D\ARCHost\Bellus3D ARC Host.exe"
--action=stop-host
```

Install the ARC Host as a Windows Service (deprecating)

The old method is to install the ARC Host as a windows service. This was the default method in the past, but we are deprecating it due to some of the drawbacks of this architecture. Note that this method will continue to be supported for our old developers for some extended time to be backward compatible, but we do not recommend it if you are a new developer. Due to complicated system environments, there is no guarantee that this method will work for all new developers. If this method does not work for you, please switch to the default install method.

To install ARC Host in this way, we need to run the ARC Installer with an additional argument:

```
<path-to-arc-installer>/Bellus3D_ARC_Host_Installer.exe /W
```

In the above command line, the argument `/W` makes everything work in the old way.

With this method, the Host becomes a windows service registered in the operating system. The service name is "archostservice". The display name is ARC Host Service.

In this architecture, most of the things work identically as with the default install method, except the way we start and stop the ARC Host, more specifically:

We need to use the **net start/stop** command to control the host service.

- start host: **net start archostservice**
- stop host: **net stop archostservice**

If there are any changes in the host configuration, please, run **net stop archostservice & net start archostservice** to restart the host and new configs will be effective.

Also note that the windows service “archostservice” is added to the system as an auto start service, which means it auto starts itself when the system boots up. But just in case, please make sure you check if the service is actually running before any attempts to establish a new DEV API websocket connection.

Install the ARC in the background (silently without UI)

Another scenario is that developers may need to install the ARC system silently (in the background) when the user installs the developer’s customized scan applications.

To do this, we provided an argument to the installer to hide the ARC installation UI.

```
<path-to-arc-installer>/Bellus3D_ARC_Host_Installer.exe /S
```

(Deprecating) If you need to install the ARC host in windows service mode together with the silent mode, simply use both installation arguments together:

```
<path-to-arc-installer>/Bellus3D_ARC_Host_Installer.exe /W /S
```

Host Configuration

The Host Config File

The host config file “default.json” is located at “C:\ProgramData\Bellus3D\ARC\config”. If you have a different system drive other than “C”, replace “C” with your system drive label. Note that we changed our config file path since version 1.14.0. Previously the file was located at “<install-dir>/config”. The installation of 1.14.0 will automatically move the old config file to the new location. Please make sure you find your config file at the new location.

Ports

The ARC Host uses Websocket for communication with a 3rd-party developer app. Websocket ports include a port for Developer's API communication and another port for Camera communication. The host also uses a HTTP Port for hosting Scan App, Configure App and Developer's Sample App.

These ports are configurable via a **default.json** file in the Host's config folder:

```
{
    "ports": {
        "httpApp": 3001,
        "websocketCamera": 3002,
        "websocketDevApi": 3003
    }
}
```

Developers will be interacting bi-directionally with the Host via Websocket (Dev API).

The Dev API port is defined by "**websocketDevApi**". The other port "**websocketCamera**" is for communication between the host and the cameras (developers don’t need to worry about it, just make sure the port is not occupied by other programs).

The host also reserves an HTTP port, defined by "**httpApp**". There are no public DEV APIs through http for now, but please still make sure that this port is available for some internal functionalities.

Feel free to change any port that conflicts with your existing environment.

Stations

The first time you start your host service, there are no active stations. You will need to configure a new station for your ARC cameras.

We provide a Configure App to configure your station easily. Please find the App by accessing http://localhost:3001/admin in your browser. Note that if you customized the HTTP port in host post configuration, please change the URL accordingly.

If you have changed your camera positions physically in the frame, you may need to run the Configure App again to update the station.

By design our Host supports multiple stations. However, to simplify the use case, we're using a single station configuration by default.

WebSocket Communication

ARC Dev APIs use JSON for initiating requests. For most requests, Dev APIs will receive JSON responses, except for streamed previews and camera snaps, where we use a GLTF data frame to pack a JSON header, followed by a binary payload.

All JSON requests will include a session ID, established on websocket connection, and then authenticated/activated.

All initiating JSON requests will have an action field; JSON responses will include a status field, which will be 'OK' if performed successfully, otherwise you will have an error code.

Connection

On connection to the Host's websocket port, the ARC Client will receive:

```
{
  status: 'OK',
  response_to: 'connect',
  session_id: 'SESSION_ID'
}
```

The Client must cache the session_id, which will be passed to all subsequent Dev API calls.

Authentication APIs

Once a websocket connection is established, the first API call to the host should be "session_init", which carries the Developer's client_id and client_secrent.

The host will authenticate the developer session to Bellus3D cloud. Developers are allowed to send further API calls only if this authentication is successful.

session_init (Client Request)

This request authenticates the current Developer API session.

```
{
  session_id: 'SESSION_ID',
  client_id: 'your client id',
  client_secret: 'your client secret',
}
```

If successful, the response is:

```
{
  session_id: 'SESSION_ID',
  status: 'OK',
  client: {...}, // SDK client's information
}
```

session_term (Client Request)

This request terminates and clears the current Developer API session.

After this API call, the Developer API session ends. To start any other API calls, developers will need another session_init to authenticate again.

```
{
  session_id: 'SESSION_ID',
}
```

If successful, the response is:

```
{
  session_id: 'SESSION_ID',
  status: 'OK'
}
```

Scan APIs

The following describe ARC Scan APIs.

station_list (Client Request)

This requests an enumerated list of stations currently defined for this Host:

```
{
  session_id: 'SESSION_ID',
  request: 'station_list'
}
```

If successful, the response is:

```
{
  status: 'OK',
  response_to: 'station_list',
  current_station_id: 'CURRENT_STATION_ID',
  stations: {
    'STATION_ID': {
      name: 'STATION_NAME',
      layout: 'LAYOUT_NAME',
      devices: {
        'CAMERA_ID': 'CAMERA_POSITION',
        ...
      }
    },
    ...
  }
}
```

station_select (Client Request)

```
{
  session_id: 'SESSION_ID',
  request: 'station_select',
  station_id: 'STATION_ID'
}
```

This must be successfully called prior to using any following Dev API actions.

If successful, the response is:

```
{
  status: 'OK',
  response_to: 'station_select'
}
```

station_start (Client Request)

This request makes sure that all cameras in a station are ready for use. After receiving this request, the host will perform any necessary actions e.g. waking up a sleeping camera.

The design of the ARC system is, before actually starting each scan, Dev API users should always send this request to make sure the cameras are ready, otherwise there is no guarantee that a scan can be initiated properly.

Dev API users should NOT send any other requests during the fulfillment of this request (before receiving a response with status "OK" from this request).

```
{
  session_id: 'SESSION_ID',
  request: 'station_start',
  restart_cameras: true|false (optional)
}
```

If successful, the response is:

```
{
  status: 'OK',
  response_to: 'station_start'
}
```

Otherwise:

```
{
  status: 'ERROR',
  response_to: 'station_start',
  error: {message: 'some error message'}
}
```

Also note that, there is an optional argument in this request: `restart_cameras`. The default value is false. The station is supposed to be ready for use after a successful regular station_start call. However, in case any camera is in any unexpected wrong status which prevents the whole station from working properly, please call the station_start API again with the argument `restart_cameras` being true, which will force restart all camera clients to make sure they are back in working order.

station_stop (Client Request)

Dev API users should send this request if the station will be idle for a long time.

This request will put all cameras into sleeping mode.

```
{
  session_id: 'SESSION_ID',
  request: 'station_stop'
}
```

If successful, the response is:

```
{
  status: 'OK',
  response_to: 'station_stop'
}
```

Otherwise:

```
{
  status: 'ERROR',
  response_to: 'station_stop',
  error: {message: 'some error message'}
}
```

station_release (Client Request)

This requests that the ARC Host release the station for other use.

```
{
  session_id: 'SESSION_ID',
  request: 'station_release'
}
```

If successful, the response is:

```
{
  status: 'OK',
  response_to: 'station_release'
}
```

preview_start (Client Request)

This requests that the ARC Host begin streaming from the ARC cameras. If called when the camera is already streaming, this request is ignored.

```
{
  session_id: 'SESSION_ID',
  request: 'preview_start',
  source: 'COLOR',
  format: 'JPEG',
  dimension: '408x544',
  camera: CAMERA_POSITION | null,
  frames: 0,
  tracking: ['FACE']
}
```

source must currently be 'COLOR'.

format: must currently be 'JPEG'.

dimension must be EXACTLY one of the following: '408x544', '612x816', '1224x1632'.

camera is the camera position to stream frames from.

Important Note:

In the case where "camera" is "null", the host will turn on preview for all cameras in the current station. This API call with "camera" being null is highly recommended before starting a scan using the other API "scan_record", giving all the cameras a chance to adjust auto white balance. Note that only the frames from the center camera will be received. And do not forget to call "stream_stop" right before calling "scan_record".

** An exception is that for ARC1 operator mode, you should NOT stop streaming until all angles are captured, see the ARC1 operator mode section for more information.*

frames indicate the number of frames requested. If 0 or undefined, the Host will continue streaming until it receives a preview_stop request.

tracking is currently an experimental feature and subject to change in future. In the current version, the host may return a "FACE" field when available, and under which, you may find another two objects "distance" and "facerect". The "distance" object holds properties x, y and z, which are offsets, measured by millimeters, from the center camera to the center of the head in front of the camera. The "facerect" object holds properties x, y, width and height, which locates the detected face in the streaming frame. Please keep in mind that the definition of this field is not final and subject to change, if you decide to use it, please make sure you check back to this section for updates in future versions.

If initiation is successful, the response for each camera is:

```
{
  status: 'OK',
  response_to: 'preview_start',
  camera: CAMERA_POSITION
}
```

camera is the camera position.

Once initiated, the Host will begin to preview streaming JPEG frames in **GLTF** format.

Each frame will be comprised of a JSON header, followed by a binary image payload of the format with a dimension specified:

```
{
  camera: 'CAMERA_POSITION',
  filesize: FILE_SIZE,
  dimension: '240x320',
  index: FRAME_INDEX,
  tracking: {
      FACE:
      {
        distance: {x, y, z},
        facerect: {x, y, width, height}
      }
  }
}
```

camera is the camera position.

filesize is the size in bytes of the binary payload (JPEG buffer size).

format is the format requested.

index is the 0-based index of the streamed frame.

The **tracking** object contains keys to the returned tracking types (currently 'FACE'), which return objects containing values specific to that type:

distance (z offset, y offset, and depth distance in mm),

facerect (x, y coordinate of the rect's top left corner, width and height of the rect, all values in range [0, 1], as a norm to the image's width and height).

This object may be absent if we are unable to retrieve it from the stream.

preview_stop (Client Request)

This requests that the ARC Host stop preview streaming from the ARC cameras. If the camera has already stopped due to completing the requested frames, through a previous **preview_stop** request, or if the camera is not otherwise streaming, this will have no effect.

Note: there will be a delay between the time that a stop action is requested, and the camera actually receiving the action, so expect that some frames will continue to stream in. One approach is to simply ignore incoming frames once you've initiated a **preview_stop**.

```
{
  session_id: 'SESSION_ID',
  request: 'preview_stop',
  camera: CAMERA_POSITION
}
```

camera is the camera position to stop preview.

If successful, the Host will respond with:

```
{
  status: 'OK',
  response_to: 'preview_stop',
  camera: CAMERA_POSITION
}
```

camera_snap (Client Request)

This requests that the ARC Host send a single image frame.

```
{
  session_id: 'SESSION_ID',
  request: 'camera_snap',
  source: 'COLOR',
  format: 'JPEG',
  dimension: '240x320',
  camera: CAMERA_POSITION
}
```

source must currently be 'COLOR'.

format: must currently be 'JPEG'.

dimension must be EXACTLY one of the following: '240x320', '480x640', '1200x1600', '2448x3264'.

camera is the camera position.

If initiation is successful, the response for each camera is:

```
{
  status: 'OK',
  response_to: 'camera_snap',
  camera: CAMERA_POSITION
}
```

camera is the camera position.

Once initiated, the Host will begin to preview streaming frames in GLTF format. Each frame will be comprised of a JSON header, followed by a binary image payload of the format and dimension specified:

```
{
  request: 'buffer_capture',
  camera: 'CAMERA_POSITION',
  filesize: FILE_SIZE,
  dimension: '240x320'
}
```

`request` will be a constant value of "buffer_capture". Note the GLTF frames from preview_start (streaming) do NOT have this property. This may be used to tell camera_snap frames from streaming frames.

filesize is the size in bytes of the binary payload.

format is the format requested.

dimension is the dimension requested.

scan_record (Client Request)

This instructs the ARC Host to request a buffer record from all connected cameras, which can be processed in-camera, and further multi-camera process when aggregated on the Host.

```
{
  session_id: 'SESSION_ID',
  request: 'scan_record',
  scan_id: 'SCAN_ID',
  scan_mode: 'FACE' | 'FULLHEAD',
  capture_mode: 'SINGLE' | 'MULTI',
  operator_mode: true | false,
  position: 'c' | 'l1' | 'r1' | 'd1'
}
```

scan_id must be cached, and used in subsequent scan actions like **scan_process** and **scan_release**. All output files will include this **scan_id** as part of their path and/or filename/URLs.

scan_mode must be specified in **ARC1, ARC3 and ARC4 stations.** The value should be either ‘FACE’ or ‘FULLHEAD’. For ARC7, this value is not supported and will be ignored (ARC7 always uses ‘FULLHEAD’).

capture_mode can be either ‘SINGLE’ or ‘MULTI’. Default value is ‘SINGLE’. **The option ‘MULTI’ is only supported by ARC3 and ARC4 stations.** Please see “multiple capture mode” below for more details.

operator_mode can be either true or false. Default value is false. **This argument is for ARC1 only.** Please see the “ARC1 Operator Mode” section below for more information.

position specifies an operator mode capture position. This argument is for ARC1 operator mode only. Values will be ignored for other scan modes. Please see the “ARC1 Operator Mode” section below for more information.

If the buffer records succeeds, the Host will respond with:

```
{
  status: 'OK',
  response_to: 'scan_record',
  scan_id: 'SCAN_ID'
}
```

(For ARC3,4,7 only) The host will also send the first frame during the capture back to the Dev API client, using the same `GLTF frame` format as **camera_snap**.

(For ARC1 only) If the params head_pose is true, the Host will respond with

```
{
  notification: 'head_pose',
  head_rotation: {
      index: 1,
      head_pose:{
       yValue:0 (-90~90)
      }
  }
}
```

head_pose: yValue indicates the rotation of head around y degree (head turning left and right).

Important Note:

Before calling this API, it is highly recommended to call "preview_start" and "preview_stop" with the "camera" param being null to allow all the cameras in the current station to adjust their auto whilte balance state. Please see more details from the API "preview_start".

scan_process (Client Request)

This requests that the ARC cameras process the buffer, and the Host aggregates the data into requested output type/formats, which is sent to the client as a binary GLTF frame.

```
{
  session_id: 'SESSION_ID',
  request: 'scan_process',
  scan_id: 'SCAN_ID',
  type: 'HEADMODEL',
  dental_bite_plate: 'DENTAL_BITE_PLATE_ID',
  position: 'c' | 'l1' | 'r1' | 'd1'
}
```

scan_id identifies which scan request the cameras/Host will process.

type indicates the type of processing; for now, only **HEADMODEL** is supported.

dental_bite_plate is the ID of the Bellus3D dental bite plate used in the scan. **This argument is currently only supported by ARC3, ARC4 FACE scan mode.** Bellus3D dental bite plates are stored under the **/config/dental-bite-plates** folder. Each plate is stored in an individual folder, with the folder name being it's plate ID.

position specifies an operator mode capture position. **This argument is for ARC1 only.** Please see the "ARC1 Operator Mode" section below for more information.

This request won't be responded to before a model is produced.

Once a model is ready, the host will send low resolution model binary data along with the response.

The binary data will be a Javascript Blob if Dev API is being used in the browser environment.

```
{
  status: 'OK',
  response_to: 'scan_process',
  scan_id: 'SCAN_ID',
  model: { glb: binaryGlbData }
}
```

IMPORTANT NOTE:

The "model" property is deprecating. Currently it contains a low resolution head model in GLB format. In future this property will be removed. To get a preview of the head model, SDK clients will have to send an additional API scan_preview. Please see the scan_preview session right below for more information.

During the process, the Host will keep sending notifications of process progress every 1 second:

```
{
  notification: 'process_progress',
  scan_id: 'SCAN_ID',
  progress: PERCENT_COMPLETE
}
```

scan_preview (Client Request)

Call this API to get a preview of the recently processed scan. If the scan was successful, this API will respond with a binary buffer of the 3D head model in GLB format.

```
{
  session_id: 'SESSION_ID',
  request: 'scan_preview',
  scan_id: 'SCAN_ID',
  resolution: 'LD' | 'SD'
}
```

scan_id has to be the ID of the most recent scan which the host just processed successfully.

resolution should be either 'LD' or 'SD'. Note that getting the preview of 'SD' resolution is a **premium feature**. A token will be consumed to request for any premium features. To be ready to claim a premium feature, please send the "premium_features_enable" request right before "scan_preview". See the "premium_features_enable" section for more information.

If successful,   the Host will respond with:

```
{
  status: 'OK',
  response_to: 'scan_preview',
  scan_id: 'SCAN_ID',
  model: { glb: binaryGlbData }
}
```

scan_cancel (Client Request)

Scan can be cancelled before fully processed. This API can be called at any stage of the scan, no matter if the host is recording (scan_record) or processing (scan_process).

However, please keep in mind that this API is currently only supported by multi-camera scans (i.e. scan with ARC4/ARC5/ARC6/ARC7 stations). It is currently not supported by ARC1 stations.

Another important thing is that the SDK client is not allowed to start a new scan until the response of this API is received.

The request format:

```
{
  session_id: 'SESSION_ID',
  request: 'scan_cancel',
  scan_id: 'SCAN_ID'
}
```

If successful,   the Host will respond with:

```
{
  status: 'OK',
  response_to: 'scan_cancel',
  scan_id: 'SCAN_ID'
}
```

scan_release (Client Request)

Call this when all scan processing is done. This will release the associated recording buffers in the camera, freeing up camera resources.

```
{
  session_id: 'SESSION_ID',
  request: 'scan_release',
  scan_id: 'SCAN_ID'
}
```

scan_id identifies which scan request the cameras/Host will process.

If successful,   the Host will respond with:

```
{
  status: 'OK',
  response_to: 'scan_release',
  scan_id: 'SCAN_ID'
}
```

premium_features_enable (Client Request)

This API enables premium features for current API sessions. Saving model is one of the premium features.

This API will confirm with the Bellus3D cloud to verify the current Developer Client is eligible for a particular request (for example, to check if the current SDK client has enough “tokens” or not). If not, the API will respond with an error.

Premium features e.g. model_save is not allowed before calling this API.

```
{
  session_id: 'SESSION_ID',
  request: "premium_features_enable"
}
```

Note that since version 1.12.0, the ARC will only deduct one token for all the premium features for the same scan. For example, an SDK client may call model_save multiple times with different options and the host will only consume the tokens once. Another example of premium features is scan_preview with ‘SD’ resolution, only 1 token is required in total for both the scan_preview and model_save request.

However, even though only 1 token is consumed for the same scan, SDK clients need to call premium_features_enable again each time before claiming a premium feature.

If successful, the server will respond:

```
{
  status: 'OK',
  response_to: "premium_features_enable",
  model_token: "a token string"
}
```

The "model_token" in the above response is for the Developer's own reference. Developers may discard this information safely.

model_save (Client Request)

This API saves a successful scan to the local disk.

```
{
  session_id: 'SESSION_ID',
  scan_id: 'SCAN_ID',
  format: 'OBJ'|'PLY'|'STL',
  path: 'OUTPUT_PATH',
  filename: 'MODEL_FILENAME',
  resolution: 'LD'|'SD'|'HD'|'MAX',
  smoothing: 'number',
  watertight: true|false,
  face_landmark: true|false,
  ear_landmark: true|false,
  mesh_triangulation: 'random'|'uniform',
  photos: true|false
}
```

scan_id identifies which scan the host is exporting from. Note that currently we only support exporting from the current scan. As soon as the user starts a new scan, the old one is removed from memory.

path should be an absolute path e.g. C:\Models. Network drive is supported in the format starting with double backslash, e.g. \\networklocation\subfolder.

filename specifies the filename for the output model file. The filename is important when exporting in OBJ format.

resolution specifies the output resolution, default value is “SD”.

smoothing is a number in 1-10, default value is 5.

watertight determines if the output model will be watertight.

face_landmark determines if the output will include face landmark files.

ear_landmark determines if the output will include ear landmark files.

mesh_triangulation determines if we output random or uniform mesh. The supported values are “random” and “uniform”. Default value is “random”. Note that “uniform” is only supported if the scan can generate a fullhead model, otherwise the export will fallback to “random”.

photos determines whether to export photo views of each camera for the model.

If successful, the server simply responds:
```
{
  status: 'OK'
}
```

Note that if the "**path**" argument is missing, the host will send an in-memory zip binary containing all scan outputs, so the response will look like (an example of exporting in OBJ format):
```
{
  status: 'OK',
  model:{
    objzip: ZIP_FILE_BINARY
  }
}
```

camera_status (Client Request)

This requests the status of all cameras in the current layout.
```
{
  session_id: 'SESSION_ID',
  request: 'camera_status',
}
```

Note that this API call will always respond immediately, the response will be:
```
{
  status: 'OK',
  response_to: 'camera_status',
}
```

This does NOT indicate any status of any cameras. Instead, the host will send out additional camera status notifications for each camera shortly.

For each camera, you will receive:
```
{
  session_id: 'SESSION_ID',
  notification: 'camera_status',
  camera: 'CAMERA_POSITION',
  connected: true,
  device_update_required: false,
  host_update_required: false
}
```

camera is the camera layout position.

connected is **true** if the camera is reported as connected.

device_update_required is **true** if the camera's software version is outdated, it will not work with the current host. If you see this, please use the "**camera_update**" API to update cameras. All other Developer APIs will stop working until all cameras get updated.

host_update_required is **true** if the host's version is outdated, it will not work with current cameras. Please visit your Bellus3D Developer Portal to download the latest ARC Host and install the latest host.

camera_status (Host Notification)

This notification may also be initiated by the host to report the latest camera status, if there's any status change in any cameras.

```
{
  session_id: 'SESSION_ID',
  notification: 'camera_status',
  camera: 'CAMERA_POSITION',
  connected: true,
  device_update_required: false,
  host_update_required: false
}
```

Fields are the same as when requesting **camera_status**.

dental_bite_plate_list (Client Request)

This request queries a list of built-in dental bite plates available in the host.

```
{
  session_id: 'SESSION_ID',
  request: 'dental_bite_plate_list’
}
```

The host responds with a dict style list of dental bite plates.

```
{
  status: 'OK',
  response_to: 'dental_bite_plate_list',
  dental_bite_plate_list: {
  PLATE_ID: {
    format: 'PLATE_FORMAT'
    }
  }
}
```

Station Configuration APIs

The following describe ARC Station Configuration APIs (Beta).

camera_list (Client Request)

This API gets a list of all cameras connected to the USB hub.

```
{
  session_id: 'SESSION_ID',
  request: 'camera_list'
}
```

The response will be in the following pattern:

```
{
  status: "OK",
  cameras: {
    'DEVICE-ID1': {
        apk_version_no: '1.0.0',
        pac_version_no: '20200703',
        device_update_required: false,
        host_update_required: false
    },
    'DEVICE-ID2': {...},
    ...
  },
}
```

camera_update (Client Request)

This API updates all cameras connected to the USB hub.

When the host determines that any of the cameras need an update, all other API calls will be rejected until a camera_update is requested.

The API request format:

```
{
  session_id: 'SESSION_ID',
  request: 'camera_update'
}
```

The host won't respond until all cameras finish updating. Once all cameras have been updated successfully, the response will be:

```
{
  status: 'OK',
  response_to: 'camera_update'
}
```

station_configure_start (Client Request)

This API instructs the host to start a temporary station so that the SDK client can finish station configuration based on it.

A temporary station is one which misses the layout and positions of cameras, it can not be used for scanning. However it can be used to connect to cameras and take camera snapshots, through which the App user will be able to figure out the camera positions.

This API accepts an array of camera deviceIDs. The host will start a temporary station with exactly these cameras, and will only be expecting connections from them.

Camera deviceIDs may be obtained from the API camera_list.

Also note that this API requires all cameras to have been already registered, either through SDK or the Bellus3D Scan App. If any camera has not ever been registered, an error will be returned in the API response. To register a camera, please send the `camera_registration_add` API (internet required).,

The API request format:

```
{
  session_id: 'SESSION_ID',
  request: 'station_configure_start',
  cameras: ['DEVICE-ID1', 'DEVICE-ID2']
}
```

If successfully, the host will respond:

```
{
  status: 'OK',
  response_to: 'station_configure_start'
}
```

station_configure_finish (Client Request)

This API creates a new station with all the necessary information.

The host will remove the temporary station created during station configuration and create a fully functional station with the arguments layout, positions and rotations.

This API will not respond until the new station is created and ready for a scan.

A sample request for creating a ARC4 (t4) station may look like:

```
{
  session_id: 'SESSION_ID',
  request: 'station_configure_finish',
  layout: 't4’,
  positions: {
    'DEVICE-ID1': 'c',
    'DEVICE-ID2': 'l1',
    'DEVICE-ID3': 'r1',
    'DEVICE-ID4': 'd1'
  },
  rotations: {
    'd1': 90
  }
}
```

If successfully, the host will respond:

```
{
  status: 'OK',
  response_to: 'station_configure_finish',
  station_id: 'the newly created station id',
  station: {...}  // the newly created station information
}
```

The definitions for different station layouts are (the layout argument):

ARC1: `'single'`

ARC3: `'horz3'`

ARC4: `'t4'`

ARC6: `'u6'`

ARC7: `'cross7'`

camera_registration_query (Client Request)

This API queries registration information for a particular camera. Note that the host only accepts queries for cameras currently connected on the USB hub.

The API request format:

```
{
  session_id: 'SESSION_ID',
  request: 'camera_registration_query',
  device_id: 'DEVICE-ID1'
}
```

The response will be in the following format:

```
{
  status: 'OK',
  response_to: 'camera_registration_query',
  device_id: 'DEVICE-ID1'
  camera_registration: {
    'registration_date': '2020-10-10T00:00:00.000Z',
    'Client_id': 'test-client'
  }
}
```

camera_registration_add (Client Request)

This API registers a camera under the current authenticated SDK client. Note that the host only accepts queries for cameras currently connected on the USB hub.

Camera registration is required when using the Bellus3D Scan App. For SDK client Apps, the camera registration is currently only for reference purposes. The Host will work well without camera registrations. However, we still recommend SDK developers to call this API to have all the cameras registered if possible, unless there is always no internet connection in the use cases.

The API request format:
```
{
  session_id: 'SESSION_ID',
  request: 'camera_registration_add',
  device_id: 'DEVICE-ID1'
}
```

If successful, the host will respond:
```
{
  status: 'OK',
  response_to: 'camera_registration_query',
  device_id: 'DEVICE-ID1'
  camera_registration: {
    'registration_date': '2020-10-10T00:00:00.000Z',
    'Client_id': 'test-client'
  }
}
```

Special Scan Modes

ARC3/ARC4 Multiple Capture Mode

ARC3 and ARC4 stations support a multiple capture mode when doing FULLHEAD scan.

In the regular (i.e. 'SINGLE') FULLHEAD scan, the capture is for example, 8 seconds of continuous user head motions. However, in the multiple capture mode, the capture is intermittent, the head motions are being captured separately with intervals.

In the API perspective, the ARC client sends multiple **scan_record** requests. Each request is sent with option **scan_mode** being 'FULLHEAD' and **capture_mode** being 'MULTI'. The Host will respond with a **scan_id** after the 1st capture, then, in the following captures, the ARC client must include the same **scan_id** to continue the captures. The ARC client is not allowed to start another capture when the previous capture is still processing. After several captures (for example 4 captures) are done, the ARC client may send a scan_process to generate the scan result.

Note that for each single capture, the Host needs 1.2 - 1.5 seconds to open the camera, starting from the time each **scan_record** request is received. Then, the camera will capture the user for about 2 seconds, at this time, it is ~3.5 seconds from the **scan_record** requests. The capture is finished at this time, however, the ARC client has to wait for another ~3 seconds of processing time. The API **scan_record** will not be responded to before all the above mentioned tasks are completed.

Below is a sample workflow of the entire scan:

1. Send the 1st **scan_record**, with scan_id set to null. After capture is completed, you get a scan_id, say 'xxx', in the response.
2. Send the 2nd **scan_record**, with scan_id set to 'xxx'.
3. Send the 3rd **scan_record**, with scan_id set to 'xxx'.
4. Send the 4th **scan_record**, with scan_id set to 'xxx'.
5. Send the final **scan_process**, with scan_id set to 'xxx'.

As a reminder, all the above **scan_record** must set the **scan_mode** to 'FULLHEAD' and **capture_mode** to 'MULTI'.

ARC1 Operator Mode

ARC1 station supports the “operator” capture mode. In this mode, the operator performs multiple captures at different angles (i.e. snapshot positions), each followed by an individual process of that capture. Once the operator finishes all the individual captures/processes, an additional process will be applied to merge all individual capture results.

In the API perspective, to capture and process in operator mode, we send multiple “**scan_record**” and “**scan_process**”.

Each **scan_record** will have to assign the following:

- **scan_id**: All individual captures will share the same **scan_id**. Leave it empty for the 1st capture. The 1st capture will respond with a **scan_id** for future reference used in the subsequent captures.
- **scan_mode**: leave empty or set to ‘FACE’.
- **capture_mode**: has to be a string value ‘MULTI’.
- **operator_mode**: has to be a boolean value ‘true’ (do not pass string).
- **position**: the snapshot position, identified by one of the following: ‘c’, ‘l1’, ‘r1’, ‘d1’

Each **scan_process** will have to assign the following:

- **scan_id**: the **scan_id** in the response of the 1st **scan_record** request.
- **position**: the snapshot position which should be the same as the one in the previous **scan_record** request.

Once all individual captures are done (captured and processed), send an additional final **scan_process** with the current **scan_id**, and an **EMPTY** (null) **position**. Also note that you may capture multiple times for the same position, if you believe your previous capture is not good.

IMPORTANT NOTE ABOUT SNAPSHOT POSITION:

The “position” property in scan_record and scan_process API are defined from the perspective of the entity being captured, NOT from the perspective of the operator. To the human being captured, the camera/operator could be to the **left**, to the **right**, in **front** of, or from **down** below. These positions are defined as ‘**l1**’, ‘**r1**’, ‘**c**’ and ‘**d1**’ respectively.

To be more specific, when the operator moves left from the center, the operator is then standing on the right side of the captured object, from the perspective of the people being captured. In this case, the snapshot position is defined as “**r1**”, rather than “**l1**”. This is very important, please always make sure you use the correct position in the API.

IMPORTANT NOTE ABOUT STREAMING:

For ARC1 Operator Mode, we need to keep the camera streaming active until the end of the entire capture. This is different from any other scan mode where you have to turn the streaming off before starting the captures. That is to say, before starting to send **scan_record**, make sure you send a **preview_start**, and with an additional property **preview_mode** = “**operatorscan**”.

```
{
  session_id: 'SESSION_ID',
  request: 'preview_start',
  camera: null,
  preview_mode: 'operatorscan'
}
```

Below is a sample workflow of the entire scan:

1. Send **preview_start** to turn on streaming, do NOT turn it off until step 10.
2. Send the 1st **scan_record**, with scan_id set to null, **position** set to ‘c’. After capture is completed, you get a scan_id, say ‘xxx’, in the response.
3. Send the 1st **scan_process**, with scan_id set to ‘xxx’, **position** set to ‘c’.
4. Send the 2nd **scan_record**, with scan_id set to ‘xxx’, **position** set to ‘l1’.
5. Send the 2nd **scan_process**, with scan_id set to ‘xxx’, **position** set to ‘l1’.
6. Send the 3rd **scan_record**, with scan_id set to ‘xxx’, **position** set to ‘r1’.
7. Send the 3rd **scan_process**, with scan_id set to ‘xxx’, **position** set to ‘r1’.
8. Send the 4th **scan_record**, with scan_id set to ‘xxx’, **position** set to ‘d1’.
9. Send the 4th **scan_process**, with scan_id set to ‘xxx’, **position** set to ‘d1’.
10. Send **preview_stop** to turn off streaming.
11. Send the final **scan_process**, with scan_id set to ‘xxx’, **position** set to **null**.

As a reminder, all the above **scan_record** must set the **capture_mode** to ‘MULTI’, **operator_mode** to true.

OPERATOR MODE TRACKING:

After you send the **preview_start** with the additional property **preview_mode** = “**operatorscan**”, you’ll start to get **facerect** and **headpose** information from the subsequent GLTF frames.

```
{
  session_id: 'SESSION_ID',
  request: 'preview_start',
  source: 'COLOR',
  format: 'JPEG',
  frames: 0,
  tracking: ['FACE'],
  preview_mode: 'operatorscan'
}
```

This information is located in the **tracking** object inside the GLTF frames, please refer to the **preview_start** API section for more details. Note that the preview in other contexts will not carry the headpose data, it's only for ARC1 operator mode.

```
{
  camera: 'CAMERA_POSITION',
  filesize: FILE_SIZE,
  dimension: '240x320',
  index: FRAME_INDEX,
  tracking: {
      FACE:
      {
        headpose: {x, y, z},
        facerect: {x, y, width, height}
      }
  }
}
```

In the headpose object, you'll find three properties **x**, **y** and **z**. They indicate the head rotation angels (degrees) in each direction. To get a good scan, we need the user's head in the correct pose for each snapshot. You may want to turn the oval red/green in your App UI based on these values.

Below is the range of head rotation angles for each position. x is the vertical degree (head facing up and down) around x axis and y is horizontal degree (head facing left and right) around y axis.

1. Position c: **y: -10° ~ 10°** , **x: -15° ~ 15°** (head facing to the operator)
2. Position l1: **y: -30° ~ -50°** , **x: -15° ~ 15°** (head facing to the left)
3. Position r1: **y: 30° ~ 50°** , **x: -15° ~ 15°** (head facing to the right)
4. Position d1: **x: -15° ~ 15°** , **x: 15° ~ 35°** (head facing up, capturing from down below)