

In-Sight HMI Developers Guide

Document Revision 25.2.0.64

Table of Contents

In-Sight HMI Developers Guide.....	1
1 Overview.....	3
1.1 Definitions.....	3
2 In-Sight Camera Configuration.....	4
2.1 In-Sight Firmware Version 6.x+.....	4
2.2 In-Sight Firmware Version 22.2+.....	5
3 Viewing HTML.....	6
3.1 HTML Pages.....	7
4 HMI API.....	10
4.1 Establish an HmiSession and Receive Results.....	10
4.2 Processing Results.....	13
4.2.1 Requesting an Image.....	14
4.2.2 Accessing the Graphics.....	14
4.2.3 Accessing Cell Results.....	15
4.3 Setting a Cell Value.....	16
4.4 Retrieving Camera Information and Settings.....	16
4.5 Monitoring and Changing State.....	18
4.6 Loading/Saving a Job.....	19
4.7 Manual Triggering.....	19
4.8 Loading an Image.....	19
Appendix A: CogSocket Basics.....	20
A.1. Requests & Response Objects.....	20
A.1.1. Type.....	20
A.1.2. Request Identifier.....	20
A.1.3. Path.....	21
A.1.4. Error.....	21
A.1.5. Headers.....	21
A.1.6. Body.....	21
A.2. Message Encoding.....	21
A.2.1. JSON Message Encoding.....	21
A.3. Get Request.....	22
A.4. Put Request.....	22
A.5. Post Request.....	23
A.6. Listen Request.....	24
A.7. Unlisten Request.....	24
A.8. Event Request.....	25

1 Overview

This document describes how to use the In-Sight HMI API to access results, access camera information, or perform camera operations.

1.1 Definitions

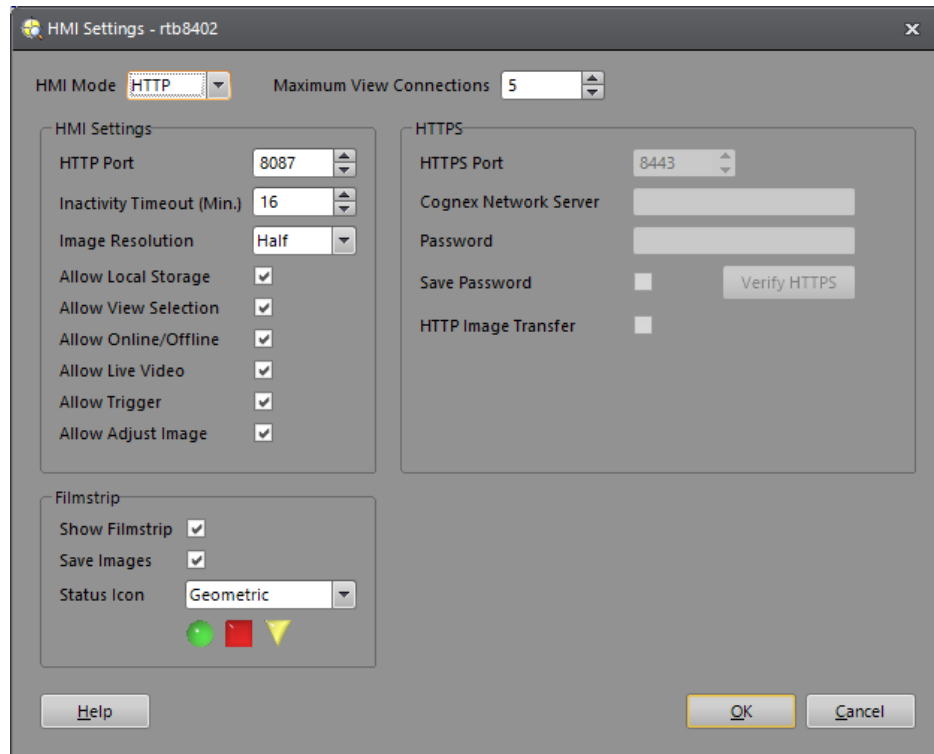
Term	Definition
HTML5	HTML5 is a markup language used for structuring and presenting content.
SVG	SVG stands for Scalable Vector Graphics. It is used to define vector-based graphics that are defined in XML format.
WebSocket	A WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection.
CogSocket	The CogSocket protocol uses a WebSocket connection to transfer messages that are loosely similar to HTTP requests and responses. Each CogSocket message is either a request, or a response to a request. Unlike HTTP, each party may send requests to the other with no distinction between client and server; the term "client" simply identifies the system that establishes the initial connection.
JSON	JSON (JavaScript Object Notation) is a text-based format for encoding objects, and its full specification can be found at http://www.json.org .

2 In-Sight Camera Configuration

2.1 In-Sight Firmware Version 6.x+

The HMI Web Server is configured using In-Sight Explorer (ISE). Support was initially included with firmware version 5.6.0. Revision 6.1.1 or newer is recommended.

Under the **Sensor Menu**, choose the "**HMI Settings...**" item to configure the server. To start the server when the camera is powered-up, be sure that the **HMI Mode** is set to "HTTP" and an acceptable port number is chosen. The default port number is 8087.



ISE HMI Settings

The **Maximum View Connections** setting determines how many concurrent connections may be made to the camera. The connections include Web HMI connections as well as ISE or VisionView connections. See the ISE help file for additional information.

2.2 In-Sight Firmware Version 22.2+

The HMI Web HMI page is configured using ISVS (In-Sight Vision Suite). In the **Utilities**, choose **HMI Settings**. The web server uses a fixed HTTP port of 80. The number of Web HMI connections is not adjustable and is set to 5.

HMI Settings - Cognex2800

Configure the settings for the HMI on Cognex2800

Image Resolution

Full

Allow Online/Offline

☒

Allow Live Mode

☒

Allow Trigger

☒

Allow Job Load

☒

Allow Job Save

☒

Allow Adjust Image

☒

Allow View Selection

☒

Allow Browser Local Storage

☒

Show Filmstrip

☒

Allow Save Filmstrip Images

☒

Status Icon

Geometric

Enable Inactivity Log Out

☐

Inactivity Log Out

15

minutes

OK

Cancel

ISVS HMI Settings

3 Viewing HTML

Once the server is running, the default web page may be accessed on the camera by entering the camera IP address and port as the address in a supported browser. For example, for a camera at 10.28.0.1 and HMI port 80, this would be "http://10.28.0.1:80". This will load the default HMI page configured on the camera. See the ISE or In-Sight ISVS help file for additional information.

To use the default web page on the camera, a modern browser should be used that supports HTML 5, CSS, WebSocket connections, JavaScript, and SVG. It is recommended that Chrome (or Chromium-based browser) with at least revision 80 is used. Support for other browsers or versions of Chrome/Chromium is not guaranteed.

A web page may also be embedded into a client application by mean of the WebView2 control. For additional information, see: <https://docs.microsoft.com/en-us/microsoft-edge/webview2/>.

3.1 HTML Pages

The default Web HMI page shows a single sensor and provides all the common operator capabilities. This includes an Image and Graphics Display, EasyView table, HMI custom view, Filmstrip, etc.



Default Web Page

You can optionally append URL query parameters to the vision system or sensor's web address to control the appearance of the display and to suppress the credentials dialog. (Note: Query string support requires at least In-Sight 6.1.1 firmware.)

The URL to access the web page has the following syntax:

`http://{address}:{port}?{query option 1}&{query option 2}...`

In a user application that uses an embedded browser, the additional controls may not be necessary. In this case, display of the image and graphics are most useful. To display an image and graphics page, use a URL query string as follows:

`http://127.0.0.1:80/?page=ImageAndGraphics`



Image and Graphics Display

The Image without graphics may also be displayed:

<http://127.0.0.1:80/?page=Image>

The default page also has options to select what View should be displayed. By default, the last view is displayed.

Query Option	Description
view=ImageOnly	Display just the image without graphics.
view=ImageWithGraphics	Display the image and overlay graphics.
view=EasyViewWithImage	Display the EasyView, image, and overlay graphics.
view=EasyViewWithoutImage	Display just the EasyView.
view=CustomView	Display the Custom View, image, and overlay graphics.
view=default	Display the default page (i.e. not the last view). (Added with 22.2.1)

The following image-related query options are also available on the Image and ImageAndGraphics pages:

Query Option	Description
fit=true	Fit the image to the display. The image's original aspect ratio is maintained; some empty (grey) space may exist around the image.
fill=true	Fill the entire display with the image.
zoom=[zoom level]	Increase or decrease the magnification of the image. Supported levels are: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, 2.75, 3, 3.5, 4, 4.5, 5, 5.5, 6, 8, 16 and 32.
rotate=[angle]	Supported angles include: 0 = no rotation 90 = rotated clockwise 90° 180 = rotated clockwise 180° 270 = rotated clockwise 270°
pan=[x,y]	Specify the x and y values of the image offset.
viewControls=true	Displays the interactive image manipulation toolbar.
fillScreen=true	Scales the page so that it fills the browser window with scrollbars when needed, instead of the default fitting of the entire page in the browser window. (Added in firmware version 22.2.1)

Designating the username and password with the URL will avoid requiring the user to login via a login dialog. The designated values may be base64 ASCII URL encoded. (Note: Designating a password on the URL is not secure and may not be acceptable for some applications or installations.) For example:

<http://127.0.0.1:80/?page=Image&user=admin&password=mypassword>

Query Option	Description
user	The username to use for the connection.
password	The password to use for the connection.

To avoid the cookie warning, `hideCookieWarning=true` may be designated.

4 HMI API

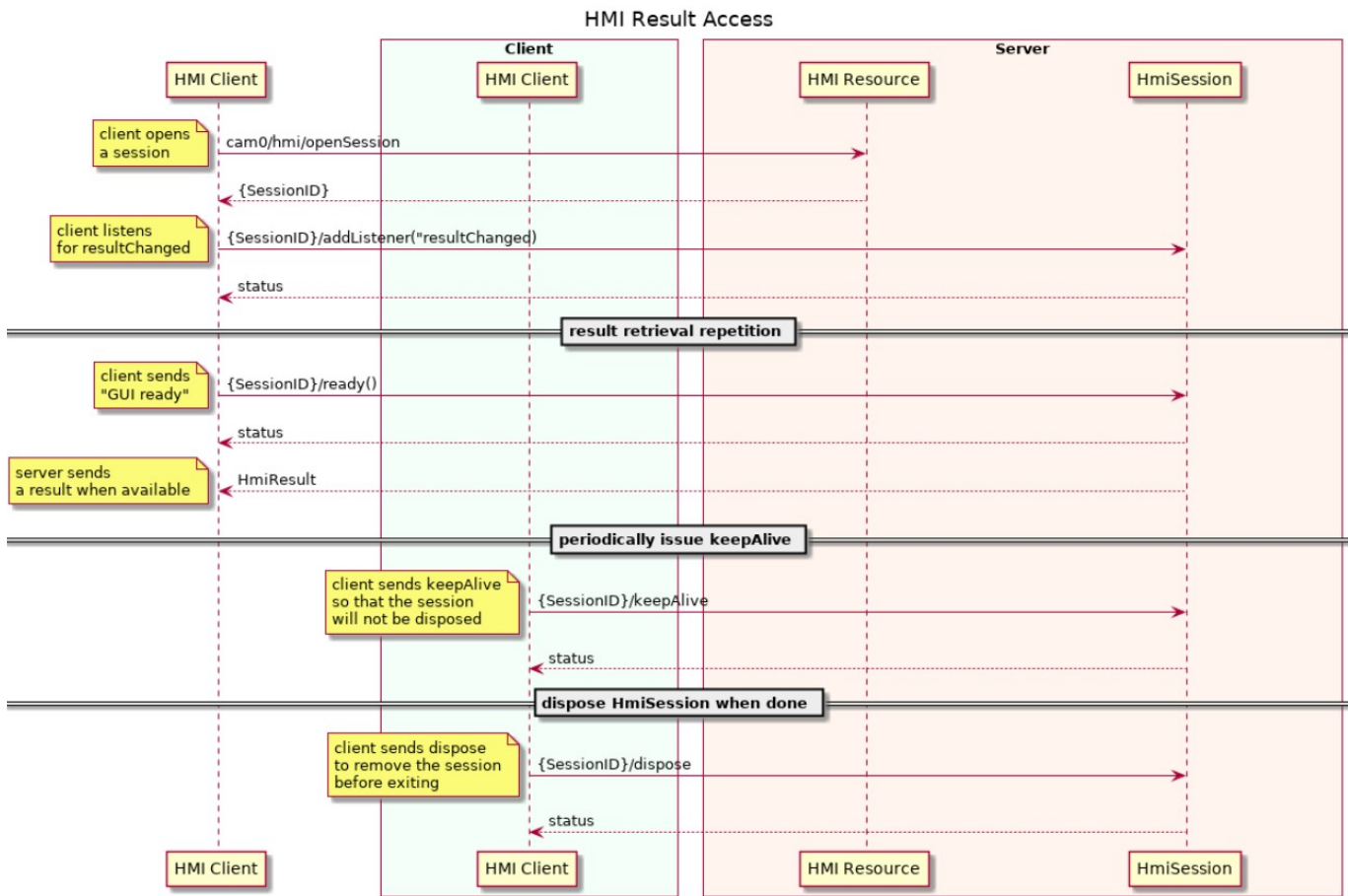
Sometimes it might be necessary to just access the camera directly and not use the web pages that are hosted on the camera. In this case, it is possible to use the In-Sight HMI API directly from the server on the camera.

To use the HMI API, a WebSocket connection must be made to the camera. The In-Sight HMI API is built on top of a Cognex-specific WebSocket protocol called CogSocket. (The `cogsocket.js` sample code may be used as a reference or see [Appendix A](#) for details about CogSocket.)

For additional information regarding the API used in the sections below, refer to the In-Sight HMI Protocol specification.

4.1 Establish an HmiSession and Receive Results

The following sequence diagram shows the necessary steps to receive result data from the camera via the In-Sight HMI Protocol.



To establish a connection to the camera via CogSocket, the client must POST to the **openSession** resource. Here is an example that uses JavaScript (See the **display_results.html** sample file for the complete source):

```

cogsocket.connect("ws://10.28.0.1:80/ws", outputText, onConnectHandler);
...
function onConnectHandler()
{
  // Request that all the cells in the range A0:Z399 are included in the results
  // Note: The sessionInfo is JSON formatted.
  let sessionInfo = '{"cellNames\\": [\\"A0:Z399\\" ]}';
  this.sessionInfo = sessionInfo;
  cogsocket.post(postResult, rootPath.value + "/openSession", this.sessionInfo, onOpenSessionComplete);
}
  
```

}

In this example, `outputText` is the id of an HTML section that can display the messages that are sent and received. This field is optional and may be null.

The **SessionInfo** will designate which cells to include in the results that will be sent in the **resultChanged** event.

\$type	"HmiSessionInfo"
cellNames	An array of cell names or range of cells that designates what cell results to include in the results. In version 2.4 of the protocol, a range of cells may also be designated in the array. For example, adding "A0:Z599" would select all cells in the sheet.
enableQueuedResults	A Boolean flag that designates whether the queued results should be provided by the HmiSession when the system result queue is frozen. This is false by default.
includeCustomView	A Boolean flag that designates whether to include the custom view range of cells in the cells array in the results.

Once the session has been created, then the user supplies credentials using the login method. The username and password must be designated as arguments. The username and password should be 64-bit ASCII encoded before transmission. There is an optional third parameter, a boolean, that may be used to designate whether to encode the arguments. Set this third argument to false when the arguments will not be encoded. There is also an optional fourth parameter, a boolean, that may be used to designate to return the **UserAccessInfo** object instead of the string access level.

```
function onOpenSessionComplete(resp)
{
  if (!resp || resp.number) {
    return; // Unable to establish a session
  }

  sessionID = resp;
  cogsocket.post(postResult, sessionID + "/login", ["\admin\", \"mypassword\", false]);
  cogsocket.addListener(postResult, sessionID + "/resultChanged", onResultChanged);
  sendReady(); // Accept the first result
  timerID = setTimeout(onKeepAlive, 20000);
}
```

After logging in, the listener was added for the **resultChanged** event, a "ready" is issued to accept the next result, and a timer is initiated to keep the session alive when there is no other communication occurring.

```
function onResultChanged(resp)
{
  hmiResult.innerText = JSON.stringify(resp[0],null,2);
  sendReady();
}

function sendReady()
{
  cogsocket.post(postResult, sessionID + "/ready", "");
}

function onKeepAlive()
{
  cogsocket.post(postResult, sessionID + "/keepAlive", "");
  setTimeout(onKeepAlive, 10000);
}
```

When the window is closed, don't forget to stop the keep-alive timer and dispose the session:

```
function onClose()
{
    if (timerID) {
        clearTimeout(timerID);
        timerID = null;
    }
    if (sessionID) {
        cogsocket.post(postResult, sessionID + "/dispose", null);
        sessionID = null;
    }
    cogsocket.close();
}

window.onClose = onClose;
```

4.2 Processing Results

An **HmiResult** is included in the payload of the **resultChanged** event. Here is an example:

LISTEN hs/~033b6e6c/resultChanged

Event Response Body:

```
{
  "$type": "HmiResult",
  "acqImageView": {
    "$type": "ViewRecord",
    "id": 10000000000076.0,
    "url": "/cam0/app/views/001000000000076",
    "layers": [{
      "$type": "ImageLayer",
      "url": "/cam0/img/000000000000001",
      "height": 1080,
      "image": {
        "$type": "Image",
        "id": 1.0,
        "url": "/cam0/img/000000000000001",
        "frozen": true,
        "height": 1080,
        "mask": null,
        "offsetX": 0, "offsetY": 0,
        "staticTransform": null,
        "transform": null,
        "width": 1440,
        "acquisitionInfo": {
          "$type": "AcquisitionInfo",
          "acquisitionDuration": 48.881000000000122,
          "acquisitionTimestamp": 37708.36
        },
        "bitsPerPixel": 8,
        "imageFormat": 8,
        "isColor": false,
        "name": "",
        "orientation": 0,
        "readOnly": true
      },
      "mask": null,
      "staticTransform": null,
      "transform": null,
      "width": 1440
    }, {
      "$type": "GraphicsLayer",
      "url": "/cam0/app/views/001000000000076/layers/2/graphics"
    }
  ],
  "source": "Inspection",
  "viewport": {
    "$type": "Viewport",
    "height": 1080,
    "width": 1440
  }
}, {
  "cellTagVer": 0,
  "cells": [{
    "$type": "HmiStringResult",
    "location": "A0",
    "data": "□Image"
  }, ...],
  "id": 3,
  "jobStatus": 1,
  "jobTagVer": 5,
  "logicVer": 0,
  "queuedResult": false,
  "rq": null
}
```

4.2.1 Requesting an Image

When an **HmiResult** is received, each **ImageLayer** in "acqImageView" will have a url that may be used to request the image. For example, the following JavaScript will get the image from the first layer (the main image):

```
var cameraUrl = "http://10.28.0.1:80";

// get the relative path to the main image
var imageUrl = hmiResult.acqImageView.layers[0].url;

// load the image in an image control
image.src = cameraUrl + imageUrl;
```

By default, the image will be returned as a full resolution bitmap image. To scale the image, an optional "sz" argument maybe added to the request. The argument takes the form "sz=dw, dh, sx, sy, sw, sh".

Parameter	Description
dw	The width of the destination bitmap in pixels.
dh	The height of the destination bitmap in pixels.
sx	The X location (in pixels) to be sampled from the source image.
sy	The Y location (in pixels) to be sampled from the source image.
sw	The number of columns of pixels to be sampled from the source image.
sh	The number of rows of pixels to be sampled from the source image.

For example, to request a 640x480 image in half resolution:

```
image.src = cameraUrl + imageUrl + "?sz=320,240,0,0,640,480";
```

4.2.2 Accessing the Graphics

Typically, the graphics will be rendered in the browser using the HMI page. The graphics, however, can be programmatically accessed via the URL designated in the graphics layer. For example:

<http://10.28.0.1/cam0/app/views/001000000000136/layers/2/graphics>

Result:

```
[
  {"$type": "Text", "color": -1, "font": "Arial", "fontSize": 24, "graphicId": -1,
  "lineThickness": 2, "runtimeEditable": false, "source": "A8", "bgColor": -65536, "text": "NG", "x": 221.0, "y": 298.0},
  {"$type": "Region", "color": -65536, "font": "Arial", "fontSize": 9, "graphicId": -1,
  "lineThickness": 2, "runtimeEditable": false, "source": "A9", "angle": 0.0, "curve": 0.0, "h": 183.30557250976563,
  "showAxesLabels": true, "showScanLine": false, "showXArrow": true, "showYArrow": true, "w": 1083.51953125, "x": 216.8609008
  7890626, "y": 293.9119873046875}, ...
]
```

The In-Sight coordinate system has the origin in the top, left corner. On In-Sight devices prior to the IS-2800, The Y-axis is horizontal and increasing to the right. The X-axis is vertical and increasing downward.

4.2.3 Accessing Cell Results

The **HmiResult** is obtained either via the payload of the **resultChanged** event or by getting the **result** property of the **HmiSession**.

Each cell result for the session is included in the "cells" array in the **HmiResult**. Each cell result will have these common properties:

Property	Description
\$type	The specific type of the cell. The following specific cell types may be received depending upon the contents of the job: HmiFloatResult, HmiStringResult, HmiEditFloatResult, HmiEditIntResult, HmiEditStringResult, HmiButtonResult, HmiCheckBoxResult, HmiListBoxResult, HmiStatusResult, HmiStatusLightResult, HmiMultiStatusResult, HmiColorLabelResult, HmiErrorCellResult, HmiUnsupportedResult.
data	The value of the cell. This varies by cell type and may be a simple value or complex object.
disabled	Designates that the cell is not enabled. This typically occurs when a cell has been explicitly disabled via the Cell State properties. A web HMI should draw an item that is disabled in "grayed-out" text and should not allow it to be edited. Note: This is false , by default, and thus not part of the object unless true .
editable	Designates that the cell is an editable type (e.g. EditInt, EditRegion, etc.). This was added for version 2.0 of the protocol. Note: This is false , by default, and thus not part of the object unless true .
error	Designates that the cell is in the error state. This occurs when a tool or expression in a job cannot be evaluated or fails. A web HMI should draw an item that is in the error state as "Error" or "#ERR" and not allow it to be edited. Note: This is false , by default, and thus not part of the object unless true .
location	The cell location will be defined for every cell.
name	The name of the cell. Depending upon how the session results were requested, not every cell may have a name. (NOTE: All cells in an EasyView will have cell names.)

Note that even if a result is not received via the payload of the **resultChanged** event, any cell in the sheet can be obtained via the **queryCellResults** method. This method takes an array of cell range strings or cell names and returns an array of **HmiCellResults** for the result held by the session. For example:

```
POST hs/~033b6e6c/queryCellResults
[["A2:A7", "BlobCell"]]
Result:
[
  {
    "$type": "StringResult",
    "error": false,
    "location": "A2",
    "name": "BlobCell",
    "data": "\u0007Blobs"
  },
  {
    "$type": "StringResult",
    "location": "A7",
    "name": "A7",
    "data": "test"
  }
]
```

4.3 Setting a Cell Value

The value of a cell is set by posting to the **setCellValue** method on the **HmiSession** resource. For example:

```
POST hs/~033b6e6c/setCellValue
["MyEditInt", 10]
```

The first argument should be a cell name or location of a cell that contains an edit function (or function that allows user editing). The second argument is the JSON-encoded value for the cell.

The cell names for the items that are included in the EasyView table (that is stored in the job) can be obtained via **GET cam0/hmi/job/easyView/names**. For example:

```
GET cam0/hmi/job/easyView/names
Result:
["Job.Pass", "MyEditInt"]
```

4.4 Retrieving Camera Information and Settings

The **GET cam0/hmi/info** request will return a JSON payload with a **CameraInfo** type. For example:

```
GET cam0/hmi/info
Result:
{
  "$type": "CameraInfo",
  "acq": {
    "$type": "CameraAcqInfo",
    "isColor": false,
    "nativeHeight": 1080,
    "nativeWidth": 1440
  },
  "capabilities": ["customView", "resultsQueue", "xyCoordinates"],
  "firmwareVersion": "22.2.0",
  "hmiProtocolVersion": "3.0",
  "httpsEnabled": false,
  "ipAddress": "10.28.0.1",
  "jobExtension": "jobx",
  "macID": "00:d0:24:6c:f3:da",
  "model": "2800",
  "name": "Cognex2800",
  "serial": "1A1947PP470950",
  "supportsCustomView": true,
  "supportsDialogs": false,
  "supportsHttps": false,
  "supportsQueue": true,
  "usesXYCoordinates": true
}
```

The **GET cam0/hmi/settings** request will return a JSON payload with a **BaseHmiSettings** type that includes the settings for a camera (Note: settings are not job-specific). For example:

GET cam0/hmi/settings

Result:

```
{
  "$type": "BaseHmiSettings",
  "hmi": {
    "$type": "HmiSettings",
    "allowAdjustImage": true,
    "allowFilmstrip": true,
    "allowFilmstripSaveImage": true,
    "allowFocus": true,
    "allowJobLoad": true,
    "allowJobSave": true,
    "allowLocalStorage": true,
    "allowProcessedImages": true,
    "allowSideMenu": true,
    "allowSoftOnline": true,
    "allowSwitchView": true,
    "allowTrigger": true,
    "defaultColorScheme": "Default",
    "enableHttpImages": false,
    "imageResolution": 1,
    "inactivityTimeout": 15,
    "statusStyle": 0
  },
  "skipLogin": true,
  "userAccessList": [{
    "$type": "UserAccessInfo",
    "access": "full",
    "accessLevel": 0,
    "name": "admin",
    "privileges": [
      "IS.CFG", "IS.IMAGE", "IS.MNT", "IS.OPS", "IS.FILE", "IS.OPENFILE", "IS.WRITEFILE", "IS.JOB", "IS.OPENJOB",
      "IS.EDITJOB", "IS.2AUTH", "IS.CFGJOB", "IS.CSTMALL", "IS.SAVE", "IS.SAVEJOBAS", "IS.ONLINEOFFLINE"
    ]
  }, {
    "$type": "UserAccessInfo",
    "access": "locked",
    "accessLevel": 2,
    "name": "monitor",
    "privileges": ["IS.CSTMALL"]
  }, {
    "$type": "UserAccessInfo",
    "access": "protected",
    "accessLevel": 1,
    "name": "operator",
    "privileges": ["IS.IMAGE", "IS.OPS", "IS.OPENJOB", "IS.CFGJOB", "IS.CSTMALL", "IS.ONLINEOFFLINE"]
  }]
}
```

4.5 Monitoring and Changing State

The **GET cam0/hmi/state** request will get the state of the camera:

```
{
  "$type": "HmiState",
  "discreteOnline": true,
  "ffpOnline": true,
  "liveMode": false,
  "nativeOnline": true,
  "online": false,
  "softOnline": false
}
```

In this response, **online** is the current online/offline state of the system. The **liveMode** flag is **true** when offline and live mode acquiring. The other flags, **discreteOnline**, **ffpOnline**, **nativeOnline** and **softOnline** all must be **true** for the system to go online.

To go online, the HMI should set the **softOnline** flag on the session. For example:

```
PUT hs/~033b6e6c/softOnline
true
```

To go into live mode, the HMI should set the **liveMode** flag on the session. For example:

```
PUT hs/~033b6e6c/liveMode
true
```

Some API will not be accessible when a job editor (e.g. ISVS EasyBuilder Connect or Spreadsheet Connect, ISE) is attached to the camera. The **GET cam0/hmi/editorAttached** request will get a flag (i.e. true or false), that designates whether an editor is attached.

Some API will not be accessible when a job is loading. The **GET cam0/hmi/jobLoading** request will get a flag (i.e. true or false), that designates whether a job load is in progress.

Whenever the state related values change there is a corresponding event. The body of the **stateChanged** event is 5 boolean values: **online**, **softOnline**, **nativeOnline**, **discreteOnline**, and **ffpOnline**. For example:

```
LISTEN cam0/hmi/stateChanged
Event Response:
{"$type": "event", "body": [true, true, true, true, true], "path": "cam0/hmi/stateChanged"}
```

```
LISTEN cam0/hmi/editorAttachedChanged
Event Response:
{"$type": "event", "body": [true], "path": "cam0/hmi/editorAttachedChanged"}
```

```
LISTEN cam0/hmi/jobLoadingChanged
Event Response:
{"$type": "event", "body": [false], "path": "cam0/hmi/jobLoadingChanged"}
```

4.6 Loading/Saving a Job

A job that is stored on the camera may be loaded by posting to the **loadJob** method on the **HmiSession** resource. The job name to load should be passed in as the argument. For example:

```
POST hs/~033b6e6c/loadJob
["MyJob.jobx"]
```

When the job begins to load, a **jobLoading** event will be received with a payload of **true**. When the job load is complete, a **jobLoading** event will be received with a payload of **false**.

The job name may be retrieved via **GET cam0/hmi/job/name**. For example:

```
GET cam0/hmi/job/name
Result:
"MyJob.jobx"
```

A job may be loaded from a client device by posting to the **loadJobData** method on the **HmiSession** resource. Because a job may be very large, this should be done via HTTP/HTTPS and not via CogSocket. The content type for the body of the request should be "application/json" and must be an **HmiNamedContent** object. The content must be data that is either formatted as a base 64 encoded ASCII string or a dictionary with a "base64" keyed item that holds the data.

```
POST http://10.28.124.13:80/cam0/hmi/hs/~4fd93cc9/loadJobData
Body:
{"$type": "HmiNamedContent", name: "MyJob.jobx" content: "Qk02BBQAAAAA..."}
```

The current job may be saved by posting to the **saveJobData** method on the **HmiSession** resource.

```
POST hs/~3da91e3c/saveJobData
Returns:
{
  "$type": "Byte[]",
  "sz": 5632,
  "base64": "Sm9iLmpzb24AAAAA ... A=="
}
```

4.7 Manual Triggering

A manual trigger is issued by posting to the **manualTrigger** method on the **HmiSession** resource. For example:

```
POST hs/~033b6e6c/manualTrigger
```

4.8 Loading an Image

An image may be loaded by posting to the **loadImage** method on the **HmiSession** resource. Because an image may be very large, this should be done via HTTP/HTTPS and not via CogSocket. This method loads an image from the base 64 encoded ASCII data for an image. The second parameter is an optional Boolean flag that allows the JPG format to be designated (rather than the default BMP format). The third parameter is also optional and allow an image name to be designated for devices that support it. For example:

```
POST http://10.28.124.13:80/cam0/hmi/hs/~4fd93cc9/loadImage
Body:
"Qk02BBQAAAAA..."
```

Appendix A: CogSocket Basics

The CogSocket protocol uses a WebSocket connection to transfer messages that are loosely similar to HTTP requests and responses. Each CogSocket message is either a request, or a response to a request. Unlike HTTP, each party may send requests to the other with no distinction between client and server; the term "client" simply identifies the system that establishes the initial connection. In some applications, only the client will send requests and thus the server sends only responses to those requests, but that is an application choice not dictated by the CogSocket protocol.

Requests and responses can come in any order and at any time, so a party is free to send multiple requests before receiving any responses and it must not assume that responses will come back in the same order, nor must it assume that a response will come back before a new request from the other party. It is expected that every request will eventually get a response (except as noted under [Request Identifier](#) below), but naturally applications must be prepared to handle cases where a request gets dropped due to insufficient resources or coding errors on the other end; how this is handled is an application-specific issue and not specified by the protocol but typically a timeout mechanism would be used to abort pending requests if they don't get a response after some length of time.

The following request types are defined. The first four request types look like HTTP methods and are similar in meaning. But unlike HTTP, CogSocket provides a very direct way for objects to emit and listen to events using the last three request types:

Request	Meaning
GET	Read a value
PUT	Write a value
POST	Invoke a method
DELETE	Delete a resource
LISTEN	Start listening to an event
UNLISTEN	Stop listening to an event
EVENT	Notify a listener when an event is emitted

A.1. Requests & Response Objects

Each request and response is encoded as an object, with only a single top-level request or response object sent in each WebSocket message. There are various request object types as in the table listed above, but only a single Response type for all kinds of requests.

Any request has the possibility of failure, in which case an [Error](#) will be returned in the response as specified below. We won't duplicate this statement as we document specific request types; we'll just document their normal (successful) response with the understanding that an error might be returned instead.

Request and response messages contain the following elements.

A.1.1. Type

The message type indicates a particular kind of request, or a response. The top-level object in each message may only be one of the supported request types or the response type; no other object types are permitted.

A.1.2. Request Identifier

Each request includes a 32-bit sender-generated integer that must be echoed back in the response to that request. It is suggested that a sender simply increment an integer request number for each new request that is sent (rolling over to one rather than zero), but a receiver must assume no particular significance or pattern to the numbers and must simply echo the request ID in the corresponding response so that the sender can associate it with a specific request.

The special value zero indicates a request for which no response should be sent. If a request is received with a RequestID of zero, the receiver MUST NOT send a response even if the request cannot be processed i.e. errors should be silently ignored although they may be logged for diagnostic purposes. For example, events may be emitted by sending an EventRequest having a RequestID of zero, meaning that the sender expects no response and will not wait for one i.e. the event is emitted in a non-blocking way that lets the sender proceed immediately without waiting for the event to be handled.

A.1.3. Path

Each request includes a resource path that identifies a resource to which the request is directed, and it is a forward slash-separated path leading to a resource such as property or method of an object. The path is similar to the URI portion of an HTTP request, but it does not begin with a leading slash, and it may contain any sequence of printable characters i.e. those with code points 32 and greater. In most cases the time needed to generate and parse path strings should be insignificant but for time critical cases, applications may define very compact top-level identifiers for some resources.

The path is not present in responses.

A.1.4. Error

If (and only if) a request fails, the response contains an error code that is a negative integer. The value -1 indicates a general failure when a more specific error code is not available.

In a response that indicates an error, the [Body](#) is usually empty but may optionally contain an error message string or exception object to provide additional information about why the request failed.

A.1.5. Headers

In a manner similar to HTTP, the CogSocket protocol provides a way to include optional *headers* with additional data concerning how a request or response should be handled or interpreted. At this time, the protocol doesn't define any header types specific to CogSocket, but applications may attach any number of header objects of any kind. This provides a way to extend the protocol in application-dependent ways if needed or to support future extensions to the protocol.

A.1.6. Body

For many requests and responses, a Body is provided. The body may be entirely absent when no additional data is associated with the message.

A.2. Message Encoding

CogSocket supports two different forms of message encoding that are functionally equivalent but have different tradeoffs in terms of performance versus ease of implementation and debugging. A sender may encode each message in either JSON form or binary form, i.e. there is no connection-level requirement that all messages be JSON or binary. However, a particular implementation might not support a given type of encoding, and if it receives a message in a form it cannot handle then it must indicate this by returning a prescribed response as indicated below.

Typically, an application would use JSON encoding for messages that are small and not time-critical because they are easier to debug, and binary encoding for messages that are large and/or time-critical because they can be transmitted and processed more efficiently. It is expected, though not specifically required, that the response to a request will be encoded in the same way (JSON or binary) as the request.

The type of encoding is implied by the 'opcode' in the WebSocket header; a 'text' opcode indicates that the message is encoded in JSON and a 'binary' opcode indicates binary encoding. In JavaScript code running in a browser for example, the data sent or received by a WebSocket will be contained in either a string or an ArrayBuffer so it's easy to differentiate between the two.

A.2.1. JSON Message Encoding

JSON encoding is implied when a system receives a WebSocket text message. If the system does not support JSON encoding, then it must reply with a WebSocket text message containing the following string:

```
{"$type": "err_encoding"}
```

A message is encoded in JSON as an object with the following properties:

A.2.1.1. \$type

This indicates the type of message. For requests, the property value will be the lowercase name of one of the request types listed above, e.g. "get", "put", "post" etc. For responses the property value will be "resp".

Other JSON-encoded objects within the message body may also have a "\$type" property that indicates what type of object they represent. This is not specifically required by the CogSocket protocol, i.e. it can support any valid JSON-encoded content for the message body, but it may be required by some implementations. The naming of types other than CogSocket messages is outside the scope of this specification.

A.2.1.2. id

This property specifies the [Request Identifier](#) as a number within the range of 32-bit signed integers. If this property is absent, it is implied to be zero.

A.2.1.3. path

This property is present only in requests, and its value is the resource path as a string.

A.2.1.4. error

Present only in responses to failed requests, its value is a negative integer status code.

A.2.1.5. headers

This property is optional and typically absent. If present, its value is an array containing one or more [Headers](#).

A.2.1.6. body

This property is optional in requests and responses. Its value can be any valid JSON-encoded value i.e. a quoted string, number, object, array, or the keyword `true`, `false` or `null`.

A.3. Get Request

This is a request to read the value of a property. The path gives the full path to the property e.g. "cam0/hmi/job/name". There is no body sent with the request.

The response body contains the value of the property.

Here is a JSON-encoded example of a Get request:

```
{
  "$type": "get",
  "id": 3,
  "path": "cam0/hmi/job/name"
}
```

The response contains a body indicating the current value of the property:

```
{
  "$type": "resp",
  "id": 3,
  "body": "myjob.jobx"
}
```

A.4. Put Request

This is a request to write the value of a property. The path gives the full path to the property e.g. "cam0/hmi/keepAliveTimeout". The request body is required and contains the new value of the property.

The response is simply an acknowledgement and contains no body.
Here is an example of a Put request to set a value:

```
{ "$type": "put", "id": 4, "path": "cam0/hmi/keepAliveTimeout", "body": 60 }
```

The response simply indicates success:

```
{  
  "$type": "resp",  
  "id": 4  
}
```

A.5. Post Request

This is a request to call a method. The path gives the full path to the method e.g. "cam0/hmi/openSession". The request body is optional and if absent, there are no arguments given to the method. If the request body contains an array, it is interpreted as an array of individual arguments to the method. If the request body contains anything other than an array, it is interpreted as a single argument.

The response body contains the return value from the method. If the method returns nothing (e.g. a 'void' method in C++) then the response will have no body.

The following is an example of a POST request directed to "cam0/hmi/openSession". The single argument is a JSON encoded object.

```
{  
  "$type": "post",  
  "id": 5,  
  "path": "cam0/hmi/openSession",  
  "body": {  
    "$type": "HmiSessionInfo",  
    "cellNames": ["NamedCell1", "NamedCell2"]  
  }  
}
```

This method returns session ID, so the response includes a body:

```
{  
  "$type": "resp",  
  "id": 5,  
  "body": "hs/~1234567890"  
}
```

A.6. Listen Request

This is a request to start an event notification. The path identifies an event resource e.g. "cam0/hmi/stateChanged", and the message body may be empty, or it may contain application specific information about how the event should be handled e.g. with what priority it should be raised. The CogSocket protocol merely provides a way to handle events that are identified by name, but the way in which event identifiers are defined and published for a given application is outside the scope of the protocol and this document. Similarly, the protocol provides a way of conveying arguments associated with events, but what arguments should be provided by any particular event is left as an application-level decision.

Here's an example of a listen request:

```
{
  "$type": "listen",
  "id": 7,
  "path": "cam0/hmi/stateChanged"
}
```

The response is simply an acknowledgement and contains no body.

```
{
  "$type": "resp",
  "id": 7
}
```

A.7. Unlisten Request

This is a request to stop an event notification. The path identifies an event resource e.g. "cam0/hmi/stateChanged", or the string "*" to discontinue all event notifications. If the client is not currently listening to the specified event, the request simply has no effect and does not generate an error.

Here's an example of an unlisten request:

```
{
  "$type": "unlisten",
  "id": 8,
  "path": "cam0/hmi/stateChanged"
}
```

The response is simply an acknowledgement and contains no body.

```
{
  "$type": "resp",
  "id": 8
}
```

A.8. Event Request

This request provides a notification of an event. The Request ID may zero (absent) for this request, indicating that the caller does not expect any response, or it may have a nonzero Request ID so that the caller is notified when the event has been consumed. The path identifies the event e.g. "cam0/hmi/jobLoadingChanged", and the message body may be empty, or it may contain a single event argument value or an array of argument values.

Here is an example event message containing an event with a single argument value:

```
{
  "$type": "event",
  "path": "cam0/hmi/jobLoadingChanged",
  "body": true
}
```

Because the request contains no Request ID in this example, there is no response.