# Aurora Focus

# JavaScript Programmer's Guide

MN-005230-03EN Rev A

## Terms of Use

### Proprietary Statement

### Product Improvements

Continuous improvement of products is a policy of Zebra Technologies. All specifications and designs are subject to change without notice.

### Liability Disclaimer

Zebra Technologies takes steps to ensure that its published Engineering specifications and manuals are correct; however, errors do occur. Zebra Technologies reserves the right to correct any such errors and disclaims liability resulting therefrom.

### Limitation of Liability

In no event shall Zebra Technologies or anyone else involved in the creation, production, or delivery of the accompanying product (including hardware and software) be liable for any damages whatsoever (including, without limitation, consequential damages including loss of business profits, business interruption, or loss of business information) arising out of the use of, the results of use of, or inability to use such product, even if Zebra Technologies has been advised of the possibility of such damages. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

# Contents

# About this Guide

This document provides information on using JavaScript within Zebra Aurora Focus, including debugging techniques, sample scripts, and interface descriptions for barcode and anomaly detection results.

## Notational Conventions

The following notational conventions make the content of this document easy to navigate.

- **Bold** text is used to highlight the following:
  - Dialog box, window, and screen names
  - Dropdown list and list box names
  - Checkbox and radio button names
  - Icons on a screen
  - Key names on a keypad
  - Button names on a screen
- Bullets (•) indicate:
  - Action items
  - List of alternatives
  - Lists of required steps that are not necessarily sequential
- Sequential lists (for example, those that describe step-by-step procedures) appear as numbered lists.

## Icon Conventions

The documentation set is designed to give the reader more visual clues. The following visual indicators are used throughout the documentation set.

**NOTE:** The text here indicates information that is supplemental for the user to know and that is not required to complete a task.

**IMPORTANT:** The text here indicates information that is important for the user to know.

**CAUTION:** If the precaution is not heeded, the user could receive a minor or moderate injury.

**WARNING:** If danger is not avoided, the user CAN be seriously injured or killed.

**DANGER:** If danger is not avoided, the user WILL be seriously injured or killed.

## Service Information

If you have a problem with your equipment, contact Zebra Global Customer Support for your region. Contact information is available at: zebra.com/support.

When contacting support, please have the following information available:

- Serial number of the unit
- Model number or product name
- Software/firmware type and version number

Zebra responds to calls by email, telephone, or fax within the time limits set forth in support agreements.

If your problem cannot be solved by Zebra Customer Support, you may need to return your equipment for servicing and will be given specific directions. Zebra is not responsible for any damages incurred during shipment if the approved shipping container is not used. Shipping the units improperly can possibly void the warranty.

If you purchased your Zebra business product from a Zebra business partner, contact that business partner for support.

# JavaScript Environment

JavaScript formatting provides the capability of post-processing inspection results, changing job status or FTP path, and sending manipulated results to different communication channels (HID, Serial, CDC, TCP).

Aurora Focus provides an execution environment based on the ECMAScript 2024 standard. For additional information, go to tc39.es/ecma262/2024/. Minor environmental restrictions are present.

Using the JavaScript features available in Aurora Focus is analogous to those available in web browsers such as Chrome or Edge and standalone JavaScript environments. In most cases, it is possible to rely on online tutorials and the help of AI assistants without any specific code customization.

The JavaScript runtime in Aurora Focus provides the following features:

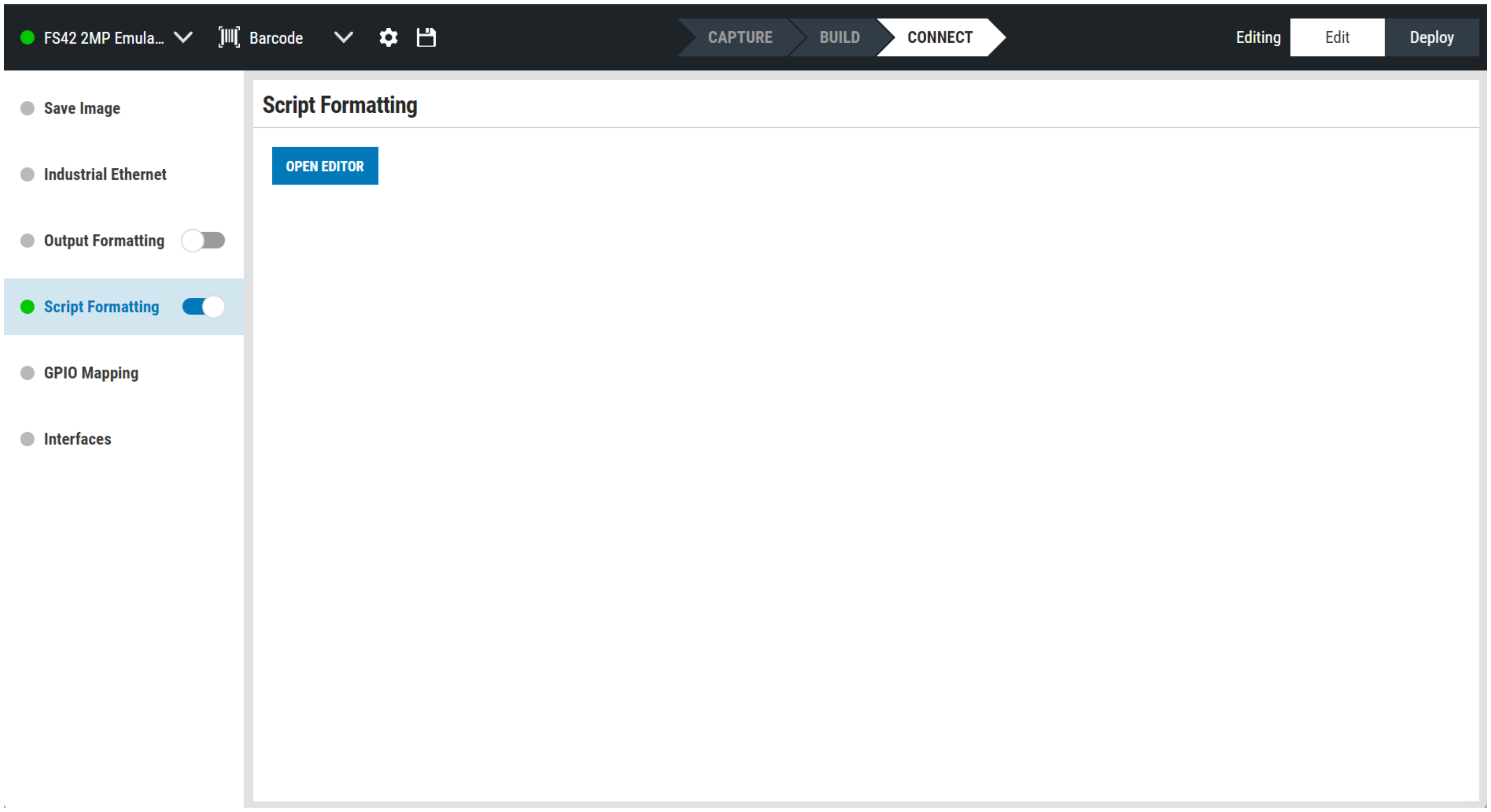
- Just-in-Time Compilation - the script is optimized during execution, generating efficient machine code after several iterations, as if written in C or C++.
- Dedicated Scripting Environment - separation of the scripting environment from the vision runtime environment ensures that problems with the scripting environment do not cause significant problems with job execution. This ensures that scripting problems should not break the acquisition environment.

## Using the JavaScript Editor

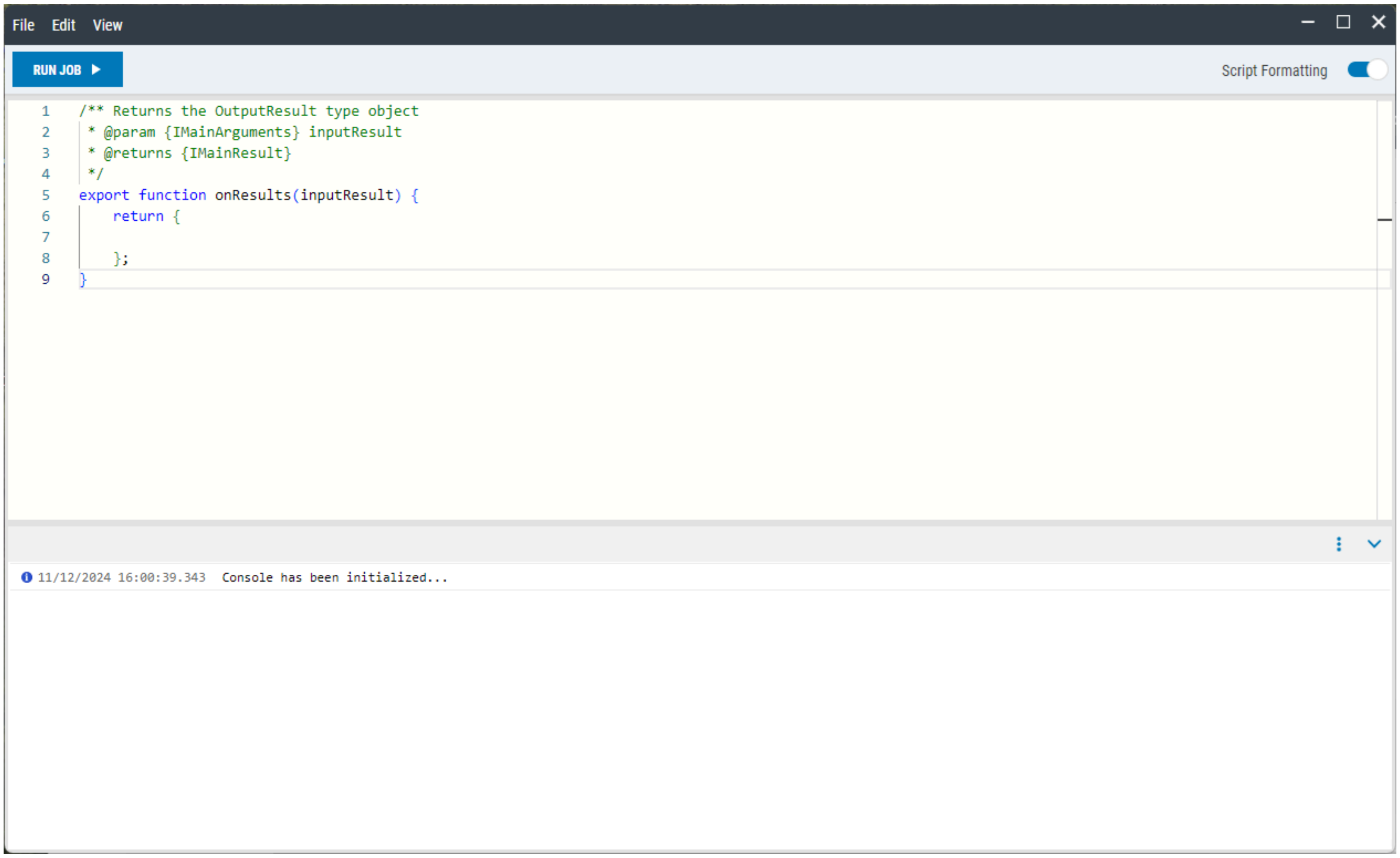
Access **Script Formatting** from the **Connect** tab to create custom scripts.

**NOTE:** JavaScript Formatting is supported on Aurora Focus version 9 and above.

1. Enable **Script Formatting** from the menu and click **Open Editor** to open a new editor window.



2. While in the editor, observe the UI elements: the navigation menu, the run job button, and the script formatting slider.



```
/** Returns the OutputResult type object
 * @param {IMainArguments} inputResult
 * @returns {IMainResult}
 */
export function onResults(inputResult) {
    return {

    };
}
```

```
11/12/2024 16:00:39.343   Console has been initialized...
```

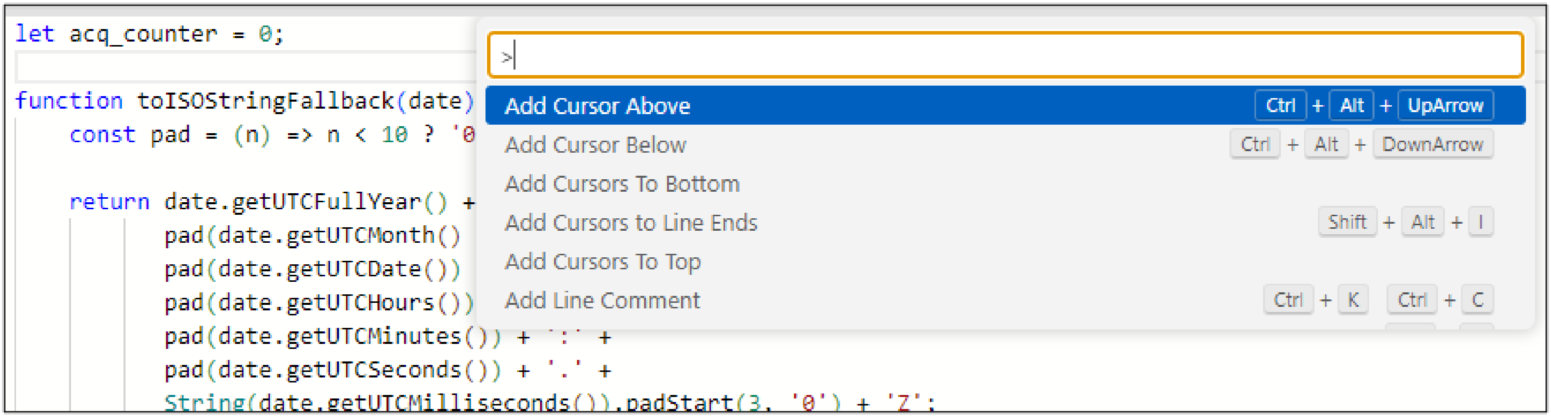
**Table 1**   JavaScript Editor UI Elements

| UI Element | Description |
|---|---|
| Navigation Menu | The top navigation menu provides access to **File**, **Edit**, and **View** functions for the application.<br>• Open an existing file with JavaScript code and save JavaScript to an external file from the **File** menu. JavaScript, by default, is saved in a job file when saving a whole job or deploying a job to the device.<br>• Use templates and snippets to quickly develop new JavaScript code that is specific to your use case from the **Edit** menu.<br>• Adjust the editor font size from the **View** menu. |

**Table 1** JavaScript Editor UI Elements (Continued)

| UI Element | Description |
|---|---|
| Run Job Button | Click **Run Job** on the control bar to validate test code in the editor by running a job on an image in the Filmstrip in Aurora Focus. You can also trigger a job by pressing F5 while in the editor. |
| Script Formatting | Disable JavaScript formatting by using the **Script Formatting** toggle in the top-right corner of the editor window. |

3. Press F1 while inside the editor for a list of all functionalities.

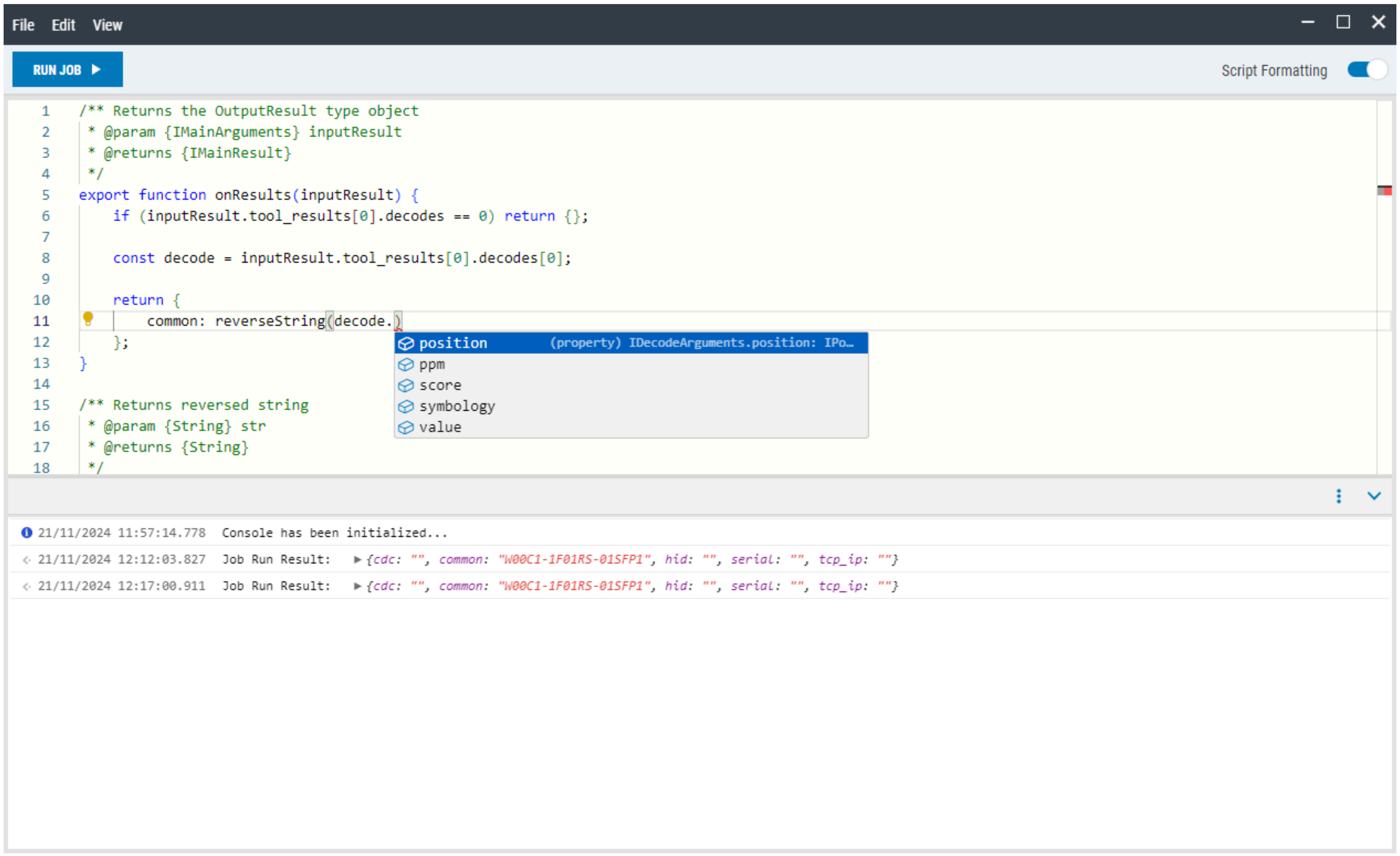


```
let acq_counter = 0;

function toISOStringFallback(date)
    const pad = (n) => n < 10 ? '0

    return date.getUTCFullYear() +
            pad(date.getUTCMonth()
            pad(date.getUTCDate())
            pad(date.getUTCHours())
            pad(date.getUTCMinutes()) + ':' +
            pad(date.getUTCSeconds()) + '.' +
            String(date.getUTCMilliseconds()).padStart(3, '0') + 'Z';
```

>
Add Cursor Above Ctrl + Alt + UpArrow
Add Cursor Below Ctrl + Alt + DownArrow
Add Cursors To Bottom
Add Cursors to Line Ends Shift + Alt + I
Add Cursors To Top
Add Line Comment Ctrl + K Ctrl + C

4. Use IntelliSense for script recommendations to enable code completion. When a function has JSDoc documentation, the editor assumes its variable types and provides a recommendation for object properties. IntelliSense is also useful for standard JavaScript interfaces such as JSON or Date.

**NOTE:** It is recommended to keep JSDoc comments for the `onResult` function. This is necessary to receive recommendations for input and output objects.



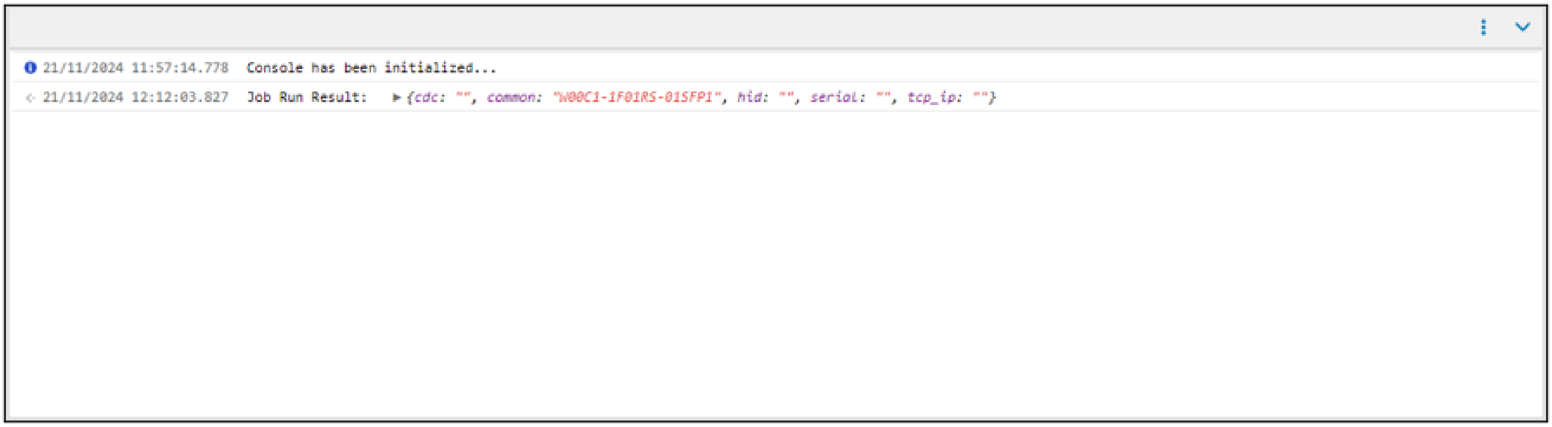
```
/** Returns the OutputResult type object
 * @param {IMainArguments} inputResult
 * @returns {IMainResult}
 */
export function onResults(inputResult) {
    if (inputResult.tool_results[0].decodes == 0) return {};

    const decode = inputResult.tool_results[0].decodes[0];

    return {
        common: reverseString(decode.)
    };
}

/** Returns reversed string
 * @param {String} str
 * @returns {String}
 */
```

```
21/11/2024 11:57:14.778  Console has been initialized...
21/11/2024 12:12:03.827  Job Run Result:  ▶{cdc: "", common: "W00C1-1F01RS-01SFP1", hid: "", serial: "", tcp_ip: ""}
21/11/2024 12:17:00.911  Job Run Result:  ▶{cdc: "", common: "W00C1-1F01RS-01SFP1", hid: "", serial: "", tcp_ip: ""}
```

5. Monaco Editor powers the code editor to integrate functionalities such as adjusting the format, undo/redo, renaming symbols, and find/replace. The following code snippet is a basic script that copies the first detected barcode from the first tool to the output.

```
/** Returns the OutputResult type object
* @param {InputResult} inputResult
* @returns {OutputResult}
*/
export function onResults(inputResult) {
  return {
      common: inputResult.tool_results[0].decodes[0].value
  };
}
```

**6.** Use the console below the code editor to view logs. The log icon and text color change depending on the log category.



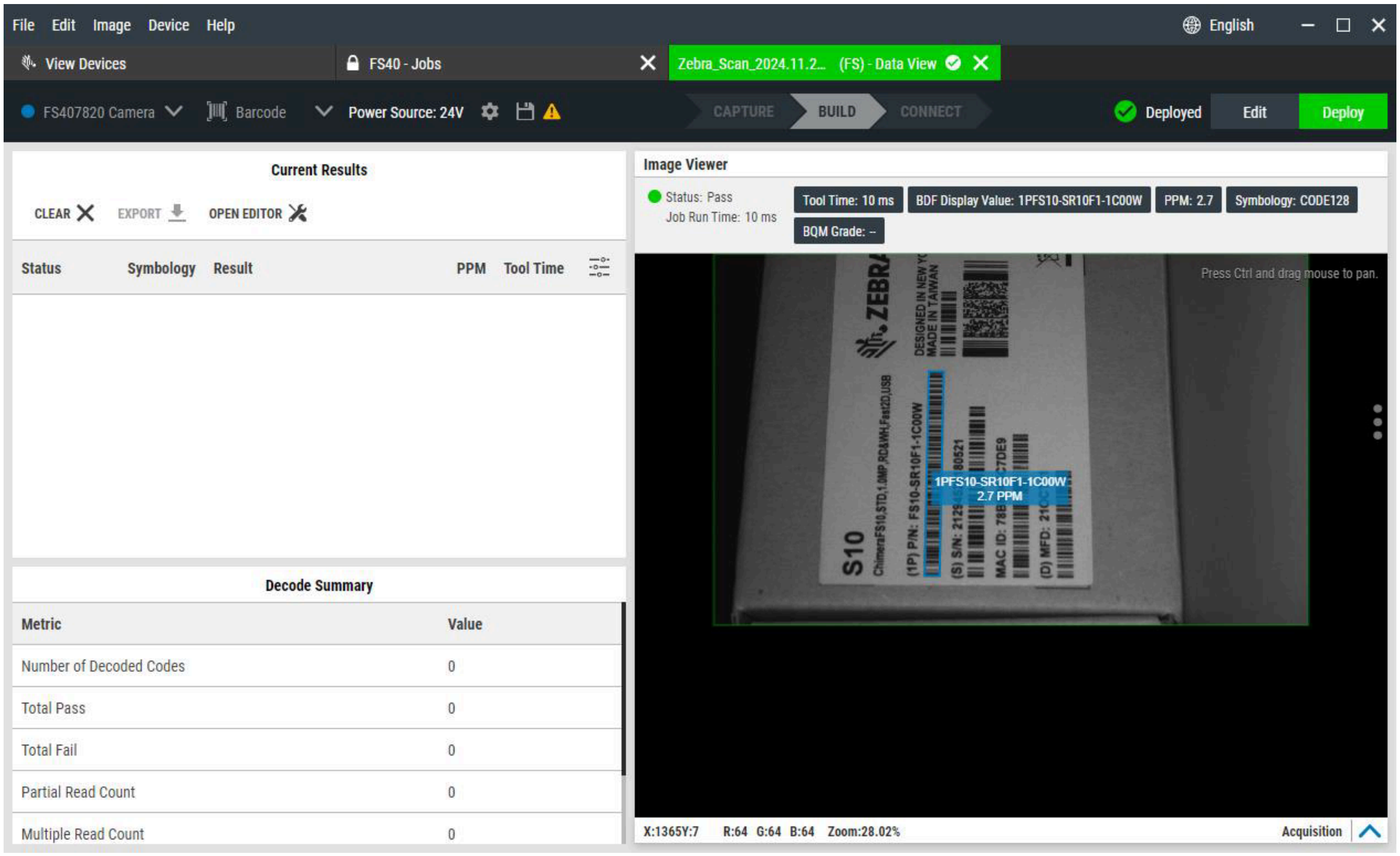
```
21/11/2024 11:57:14.778   Console has been initialized...
21/11/2024 12:12:03.827   Job Run Result:   ▶{cdc: "", common: "W00C1-1F01RS-015FP1", hid: "", serial: "", tcp_ip: ""}
```

**a)** Use ⋮ to save the console logs to a file or clear the console.

**b)** Use ⌄ to minimize the console.

## Using the Editor for a Deployed Job

Click **Open Editor** in the **Current Results** section while running a job in Aurora Focus to run the JavaScript editor.



Status: Pass
Job Run Time: 10 ms
Tool Time: 10 ms
BDF Display Value: 1PFS10-SR10F1-1C00W
PPM: 2.7
Symbology: CODE128
BQM Grade: --

| Status | Symbology | Result | PPM | Tool Time |
|---|---|---|---|---|

Decode Summary

| Metric | Value |
|---|---|
| Number of Decoded Codes | 0 |
| Total Pass | 0 |
| Total Fail | 0 |
| Partial Read Count | 0 |
| Multiple Read Count | 0 |

## Limitations

Understand the current limitations of the script editor and strategize so you do not encounter any issues.

- The ability to use external modules using the import keyword and the required functions is disabled.
- Asynchronous functions are not supported. Helper functions necessary to implement asynchronous functions, such as `setTimeout`, are not present.
- Network or file system IO functions are not supported.
- If memory usage exceeds a specific limit, the scripting environment restarts (including restarting global variables). This is no different from other runtime errors, except for refreshing the state and the time it takes to restart the environment. This can occur if you create a global variable in an array and add elements in every iteration, but do not remove them.

# Operation Mode

Use the Aurora Focus JavaScript environment to handle input/output data, interact with hardware, and utilize job status logic.

## Program Structure

Every script must contain the function `onResults`. This function takes one argument as a data structure and returns the structure.

The `onResults` function must be marked as export as in the following example:

```
export function onResults(inputResult) {
}
```

In addition to the body of the main function, a script can have multiple helper functions, classes, global variables, and code before the `onResults` function.

If you have experience with JavaScript, view your script as a JavaScript library and follow the same principles. Understand when the code is initially loaded and executed and the exported functions are used by external code.

## Input Data

The function `onResults` takes only one argument. This argument contains the output structure of all tools in the current job.

The `tool_results` and `tool_results_by_names` fields contain tools that display in the job but allow access to them differently. For example, `tool_results` is a list of tools where each index already has a defined tool type. In cases where the engine does not yet support the tools, the index takes the value `null`. The tools are sorted in the order they display in the job.

`tool_results_by_names` allows access to tools by their name, using dot notation, and support from IntelliSense, so `tool_results_by_names .my_tool` is the correct syntax. In `tool_results_by_names`, only currently supported tools display after their names. When FlowBuilder uses multiple tools, each tool is accessible by its friendly name.

**NOTE:** A friendly name might contain characters that are not valid as part of a JavaScript variable name. In this case, all invalid characters are replaced with underscore characters. In some instances, two tools might have the same JavaScript name (for example, friendly names were `tool!` and `tool@` and both got translated to `tool_`). In this case, the red triangle with an exclamation mark displays on the top right side of the editor (right before the script enabling switch). The warning sign explains which tools have a naming collision and which friendly names can be changed.

The descriptions of both fields are generated dynamically during job editing. As a result, changing the tool or its order generates IntelliSense errors at the script editing stage.

Only incoming data from the following tools is supported:

- Read Barcode
- Read DPM
- Read DPM & Barcode
- Datacode
- Anomaly Detection
- Deep Learning OCR
- Contrast
- Locate Edge
- Edge Detect
- Edge Count
- Gradient Full
- Gradient Horizontal
- Gradient Vertical

These tools share a common type and display in the argument in the same order as in Aurora Focus.

The following code snippet describes the input argument in TypeScript. In a typical development process, it is recommended to use the field subtitles feature in the editor.

```
interface IJobInformationArguments {
 name: string;
 start_time: string;
 acquisition_images_count: number;
 metrics: IMetricsArgument;
}
```

```
interface IMetricsArguments {
 trigger_counts: number;
 trigger_overruns: number;
 total_pass: number;
 total_fail: number;
 job_time_min: number;
 job_time_max: number;
 job_time_average: number;
}
```

The `job_success` field indicates if the job was completed successfully. The script can override this value using an output structure.

```
interface IMainArguments {
 tool_results: TOOL_RESULTS_TYPE[];
 tool_results_by_names: TOOL_RESULTS_BY_NAMES_TYPE;
 job_info: IJobInformationArguments;
 job_success: boolean;
}
```

The following code snippet describes definitions for additional types.

The `IBarcodeResultArguments` interface in TypeScript defines the structure for barcode scanning results.

It includes the following properties:

- `time: number;` This number indicates the time a tool or process takes, such as image acquisition.
- `tool_time: number;` similar to `time`, this is a number that might indicate the time taken by a tool or process.
- `success: boolean;` indicates whether a specific operation or process was successful (`true`) or not (`false`).
- `decode_time: number;` represents the time taken to decode a barcode.
- `decode_count: number;` represents the count of successful decodes.
- `decode_totalcount: number;` represents the total count of attempted decodes, successful or not.
- `decodes: IDecodeArguments[];` an array of objects that conform to the `IDecodeArguments` interface. Each element in this array represents a decoded result, and `IDecodeArguments` would be another interface that describes the structure of these decoded results.
- `unmatched_decodes: IDecodeArguments[];` as above, but codes that do not match the pattern specified in the tool have been detected.
- `statistics: IStatisticsArguments;` contains tools execution statistics accumulated between operations.

```
interface IBarcodeResultArguments {
 time: number;
 tool_time: number;
 success: boolean;
 decode_time: number;
 decode_count: number;
 decode_totalcount: number;
 decodes: IDecodeArguments[];
}
```

The `IPointArguments` interface defines an object structure representing a point in a two-dimensional space. It includes two properties: `x` and `y`, both of which are numbers.

- `x: number;` expected to be a number representing the x-coordinate of a point in a two-dimensional space.
- `y: number;` a number representing the y-coordinate of the point.

```
interface IPointArguments {
 x: number;
 y: number;
}
```

The `IQualityGradeArguments` interface defines an object structure for representing a quality grade.

- `grade_: number;` a number representing the numerical value of the grade.
- `letter_grade: string;` a string representing the letter equivalent of the grade, such as A, B, or C.

```
interface IQualityGradeArguments {
 grade_: number;
 letter_grade: string;
}
```

The `IDecodeBQMArguments` interface in TypeScript defines the structure for an object representing barcode quality metrics (BQM). Each property typically involves quality grading, using the `IQualityGradeArguments` interface, which includes a numerical grade and a letter grade.

```
interface IDecodeBQMArguments {
 overall_grade: IQualityGradeArguments;
 decodability: IQualityGradeArguments | null;
 decode: IQualityGradeArguments;
 defects: IQualityGradeArguments | null;
 minimum_edge_contrast: IQualityGradeArguments | null;
 modulation: IQualityGradeArguments;
 symbol_contrast: IQualityGradeArguments;
 quiet_zone: IQualityGradeArguments | null;
 left_quiet_zone: IQualityGradeArguments | null;
 right_quiet_zone: IQualityGradeArguments | null;
 wide_to_narrow_ratio: IQualityGradeArguments | null;
 inter_char_gap_2_narrow: IQualityGradeArguments | null;
 max_pixel_value_in_extended_area: number | null;
 maximum_reflectance: IQualityGradeArguments | null;
 minimum_reflectance: IQualityGradeArguments | null;
 unused_error_correction: IQualityGradeArguments | null;
 fixed_pattern_damage: IQualityGradeArguments | null;
 axial_nonuniformity: IQualityGradeArguments | null;
 grid_nonuniformity: IQualityGradeArguments | null;
 contrast_uniformity: IQualityGradeArguments | null;
 reflectance_margin: IQualityGradeArguments | null;
 horizontal_print_growth: IQualityGradeArguments | null;
 vertical_print_growth: IQualityGradeArguments | null;
}
```

`IStatisticsArguments` defines the tool execution statistics, considering all iterations (including the current iteration).

- `total_pass : number;` total number of tool successes.
- `total_fail : number;` the total number of tool failures.
- `tool_time_min : number;` minimum tool-making time.
- `tool_time_max : number;` maximum tool-making time.
- `tool_time_average : number;`average execution time.

```
interface IStatisticsArguments {
total_pass: number;
total_fail: number;
tool_time_min: number;
tool_time_max: number;
tool_time_average: number;
}
```

The `IDecodeArguments` interface defines the structure of an object representing the result of a decoding process, such as decoding a barcode.

- `value: string;` represents the decoded value, such as the data extracted from a barcode.
- `raw_value: number[];` represents the decoded value, such as the data extracted from a barcode in ASCII codes.
- `symbology: string;` indicates the type or format of the decoded code, such as QR or Code128.
- `position: IPointArguments[];` an array of `IPointArguments` objects, each representing a point in two-dimensional space. It typically denotes the positions or corners of the decoded symbol within an image or a scanning field.
- `ppm: number | null;` represents pixel per module, which is used to measure barcode module density in the images being decoded. It can also be `null`, indicating that this information is not available or applicable.
- `score: IDecodeBQMArguments | null;` an object conforming to the `IDecodeBQMArguments` interface, representing the quality score of the decoding process. It can also be `null`, indicating that a score is not available or applicable.

```
interface IDecodeArguments {
 value: string;
 raw_value: number[];
 symbology: string;
 position: IPointArguments[];
 ppm: number | null;
 score: IDecodeBQMArguments | null;
}
```

The `IAnomalyDetectionResultArguments` interface in TypeScript defines the structure for anomaly detection results. It includes the following properties:

- `time: number;` a number that indicates the time taken by a tool or process, including image acquisition.
- `tool_time: number;` similar to time, this number indicates the time taken by a tool or process.
- `success: boolean;` indicates whether a specific operation or process was successful (true) or not (false).
- `statistics: IStatisticsArguments;` contains tools execution statistics accumulated between operations.
- `valid: boolean;` indicates whether an image is considered valid (true) or not (false).
- `score: number;` represents the score of an image.
- `confident: boolean;` indicates whether a result is considered confident (true) or not (false).

- `roi: IPointArguments[];` an array of IPointArguments objects, each representing a point in two-dimensional space. It typically denotes the positions or corners of the tool region of interest.

```
interface IAnomalyDetectionResultArguments {
 time: number;
 tool_time: number;
 success: boolean;
 statistics: IStatisticsArguments;
 valid: boolean;
 score: number;
 confident: boolean;
 roi: IPointArguments[];
}
```

The `IDeepLearningOcrResultArguments` interface in TypeScript defines the structure for deep learning-based OCR results. It includes the following properties:

- `time: number;` a number that might indicate the time a tool or process takes, such as image acquisition.
- `tool_time: number;` similar to time, this number might indicate the time taken by a tool or process.
- `success: boolean;` indicates whether a specific operation or process was successful (true) or not (false).
- `characters: ICharacterResultArguments[];` an array of ICharacterResultArguments objects, each representing a decoded character.
- `ocr_result: string;` represents the decoded characters.
- `statistics: IStatisticsArguments;` contains tools execution statistics accumulated between operations.

```
interface IDeepLearningOcrResultArguments {
 time: number;
 tool_time: number;
 success: boolean;
 characters: ICharacterResultArguments[];
 ocr_result: string;
 statistics: IStatisticsArguments;
}
```

The `ICharacterResultArguments` interface defines an object structure representing a decoded character. It includes the following properties:

- `value: string;` represents the confidence character.
- `roi: IPointArguments[];` an array of IPointArguments objects, each representing a point in a two-dimensional space. It typically denotes the positions or corners of the decoded character.
- `confidence: number;` represents the confidence of the character.
- `quality: number;`represents the quality of the character.
- `height: number;` represents the height of the character.
- `is_part_of_result_string: boolean;` indicates if a character is contained in the result string.

```
interface ICharacterResultArguments {
    value: string;
    roi: IPointArguments[];
    confidence: number;
    quality: number;
    height: number;
    is_part_of_result_string: boolean;
}
```

The `IContrastResultArguments` interface in TypeScript defines the structure for contrast results. It includes the following properties:

- `time: number;` this number indicates the time taken by a tool or process including image acquisition.
- `tool_time: number;` similar to time, this number indicates the time taken by a tool or process.
- `success: boolean;` indicates whether a specific operation or process was successful (true) or not (false).
- `value: number;` indicates contrast value.
- `minimum_threshold: number;` indicates the minimum threshold value set in the tool configuration.
- `maximum_threshold: number;` indicates the maximum threshold value set in the tool configuration.
- `minimum_calculation_value: number;` indicates the minimum pixel value.
- `maximum_calculation_value: number;` indicates the maximum pixel value.
- `statistics: IStatisticsArguments;` contains tools execution statistics accumulated between operations.

```
interface IContrastResultArguments {
 time: number;
 tool_time: number;
 success: boolean;
 value: number;
 minimum_threshold: number;
 maximum_threshold: number;
 minimum_calculation_value: number;
 maximum_calculation_value: number;
 statistics: IStatisticsArguments;
}
```

The `IEdgeResultArguments` interface in TypeScript defines the structure for edge detection results. It includes the following properties:

- `time: number;` this number indicates the time taken by a tool or process including image acquisition.
- `tool_time: number;` similar to time, this number indicates the time taken by a tool or process.
- `success: boolean;` indicates whether a specific operation or process was successful (true) or not (false).
- `edge_count: number;` indicates the number of detected edges.
- `line_segments: ILineArguments[];` an array of ILineArguments objects, each representing a line in two-dimensional space.

- `statistics: IStatisticsArguments;` contains tools execution statistics accumulated between operations.

```
interface IEdgeResultArguments {
 time: number;
 tool_time: number;
 success: boolean;
 edge_count: number;
 line_segments: ILineArguments[];
 statistics: IStatisticsArguments;
}
```

The `ILineArguments` interface defines an object structure representing a line in a two-dimensional space. It includes two properties: point1 and point2, both of which are IPointArguments.

- `point1: IPointArguments;` represents the line start point.
- `point2: IPointArguments;` represents the line endpoint.

```
interface ILineArguments {
 point1: IPointArguments;
 point2: IPointArguments;
}
```

## Constant Data

Each script has a global variable called `DEVICE` that is available from anywhere in the code. They contain constant data about the current device. When using an emulator, this variable contains applicable messages about a lack of support.

`DEVICE` is described by the following type:

```
interface IDevice {
 hostname: string;
 model: string;
 part_number: string;
 serial_number: string;
 uuid: string;
 firmware_version: string;
 type: DeviceType;
 sensor: ISensor;
 gpio: IGPIO;
}
```

Using the following types:

```
enum DeviceType {
  REAL = "REAL",
  EMULATED = "EMULATED",
}
```

```
enum GpioPortState {
  DISABLE = "DISABLE",
  TRIGGER = "TRIGGER",
  JOB_SWITCH = "JOB_SWITCH",
  ILLUMINATION_STROBE = "ILLUMINATION_STROBE",
  INPUT = "INPUT",
  OUTPUT_PUSH_PULL = "OUTPUT_PUSH_PULL",
  OUTPUT_HIGH_SIDE = "OUTPUT_HIGH_SIDE",
  OUTPUT_STATIC_HIGH = "OUTPUT_STATIC_HIGH",
  OUTPUT_STATIC_LOW = "OUTPUT_STATIC_LOW",
  OUTPUT = "OUTPUT",
}

interface IGPIOPort {
  number: number;
  state: IGpioPortState;
}

interface IGPIO {
  count: number;
  ports: IGPIOPort[];
}

interface ISensor {
  sensor_type: string;
  width: number;
  height: number;
}
```

## Output Data

Control the device’s output function by using `onResults` to return a specific and straightforward structure.

```
interface IFtpResult {
  common_file_name ? : (string | null);
  json_file_name ? : (string | null);
  img_file_name ? : ((string | string[]) | null);
}

interface IMainResult {
  common ? : (string | (number[] | Uint8ClampedArray | Uint8Array));
  tcp_ip ? : (string | (number[] | Uint8ClampedArray | Uint8Array));
  serial ? : (string | (number[] | Uint8ClampedArray | Uint8Array));
  hid ? : (string | (IHIDCombinationResult | IHIDPauseResult | HIDKey |
 string)[]);
  cdc ? : (string | (number[] | Uint8ClampedArray | Uint8Array));
  ftp ? : (IFtpResult | null);
  job_success ? : (boolean | null);
}
```

## Job Status

The `job_success` property redefines the inspection Pass/Fail status based on a user-defined logic.

If assigned with true or false values, the inspection status is pass or fail. It does not change if the value is not specified.

**NOTE:** It is possible to access the inspection status with the `onResults` function argument. The property is named job_success as well.

```
/**
* Changes overall inspection status into Fail if the Read_Barcode_1
* The tool has not found a single CODE 128 barcode.
*
* Please do not remove 'export' keyword or this comment.
* onResults function triggers on the results of each job run.
* For more detailed information please refer to the documentation.
* @param {IMainArguments} input_result - object with results of the job.
* @returns {IMainResult} - object-defining outputs per interface.
*/
export function onResults(input_result) {
const results = input_result.tool_results_by_names.Read_Barcode_1;
if (results.decode_count != 1) {
   return {
       job_success: false
   }
}

if (results.decodes[0].symbology != "CODE128") {
   return {
       job_success: false
   }
}

return {
   job_success: input_result.job_success
}
}
```

## TCPIP, RS232, and USB CDC Serial

Use the JavaScript feature to write data to TCP/IP, RS232, and USB CDC-Serial external interfaces.

```
common ? : (string | (number[] | Uint8ClampedArray | Uint8Array));
tcp_ip ? : (string | (number[] | Uint8ClampedArray | Uint8Array));
serial ? : (string | (number[] | Uint8ClampedArray | Uint8Array));
cdc ? : (string | (number[] | Uint8ClampedArray | Uint8Array));
```

The properties allow writing data to external interfaces: `tcp_ip` for TCP/IP, `serial` for RS232, and `cdc` for USB CDC-Serial. For binary data, assign either a string or an array of numbers for binary data. Numbers must be within the 0-255 range.

If you assign a value to a specific property, it is sent with the interface. If a value is assigned to the common property, it is also sent with the interface. Otherwise, no value is sent.

**NOTE:** Settings in the **Communication** the script respects tabs.

```
/**
* Prints the string value to the TCP/IP interface, and the value
* enriched with prefixes and suffixes to the serial interface.
*
* Please do not remove the 'export' keyword or this comment.
* onResults function triggers on the results of each job run.
* For more detailed information, please refer to the documentation.
* @param {IMainArguments} input_result - object with results of the job.
* @returns {IMainResult} - object-defining outputs per interface.
*/

export function onResults(input_result) {
const results = input_result.tool_results_by_names.Read_Barcode_1;
const decode_text = results.decodes[0].value;
const decode_with_prefix_and_suffix = [0x02, ...results.decodes[0].raw_value,
 0x03];
return {
  tcp_ip: decode_text,
  serial: decode_with_prefix_and_suffix
}
}
```

## HID Keyboard

Enable the HID keyboard to use the device in HID keyboard mode. Assign a string, single button, key, and modifier combination. You can also add a pause between keystrokes. All of those can be leveraged into a single assignment.

```
hid?: (string|(IHIDCombinationResult| IHIDPauseResult|HIDKey|string)[]);
```

An additional helper object, `HID` is available to simplify HID usage. `HID` properties are described in the following table.

**Table 2** Helper Object HID Properties

| Property | Description |
|---|---|
| HID.key | Gives access to a list of available keys. |
| HID.combination | Allows the creation of a key and a modifier combination. |
| HID.pause | Allows a pause between consecutive keystrokes to be created. |

**Table 2** Helper Object HID Properties (Continued)

| Property | Description |
|---|---|
| HID.modifier | Gives access to a list of available key modifiers. |

**NOTE:**

- HID.pause argument must be between 0 and 25000 ms.
- HID.combination allows you to press, at the same time, a key and up to 8 different modifiers.
- HID.combination produces a series of consecutive keystrokes with the same modifiers when used with more than one key.
- Keystroke delay and preferred keyboard layout can be changed through **Communication** tab in the device settings.
- If that property is not assigned, nothing is printed.

```
/**
* Opens notepad and types into it a decoded barcode value.
*
* Please do not remove the 'export' keyword or this comment.
* The onResults function triggers the results of each job run.
* For more detailed information, please refer to the documentation.
* @param {IMainArguments} input_result - object with results of the job.
* @returns {IMainResult} - object-defining outputs per interface.
*/
export function onResults(input_result) {
const run_command = [HID.combination(HID.key.R, HID.modifier.L_WINDOWS),
 HID.pause(200)]
const type_notepad = ["notepad", HID.key.ENTER, HID.pause(1000)];
const decode = input_result.tool_results_by_names.Read_Barcode_1.decodes[0];

return {
  hid: [...run_command, ...type_notepad, decode.value, HID.key.ENTER]
};
}
```

## FTP Control

The ftp field controls FTP output if enabled in the job. If the FTP field is not set, explicitly set to undefined, or set to null, then default behavior and names are used, as set in the FTP tab for the job.

If the FTP field is set to an object, the resulting file names and whether they are sent depend on the field's values.

If `img_file_name` is set to a single string, the resulting image names are set to that name with the image bank name and number appended to the end. If it is set to an array of strings, the resulting image names are set to the names of the strings (the first image takes the first string name, and so on).

If `common_file_name` is set, but `img_file_name` is not set, the name used with the image bank index is appended to the end. If neither is set, images are not sent over FTP.

If `json_file_name` is set, and the resulting JSON file name is set to that name. If `common_file_name` is set but`json_file_name` is not set, the common file name is used. If neither is set, JSON is not sent over FTP.

## GPIO Ports

Use the gpio field to send a pulse to available ports. Parameters of the pulse, such as length and delay, can be specified in the GPIO Mapping menu in General Settings. Pulse polarity can be set through the script, using one of the three values of the helper GpioPolarityResult enum object.

```
gpio: IGpioPort[];
```

**Table 3** Helper Object GpioPolarityResult Properties

| Value | Description |
|---|---|
| `GpioPolarityResult.HIGH` | Pulse has a high value. |
| `GpioPolarityResult.LOW` | Pulse has a low value. |
| `GpioPolarityResult.JOB_SETTINGS` | Pulse parameters are taken from the specification in the GPIO Mapping settings in the job. |

The field should be an array of objects. Each object contains two properties: port_number and polarity. Port_number is zero-starting number of port. Polarity can be one of three values described in Table 3.

**NOTE:**

- The initial value can be set through the GPIO Mapping tab in the job.
- The port that is written must be set as an output. It can be done through GPIO Mapping in device settings.
- The GPIO field of the DEVICE object can be used.

```
/**
 * Sends a pulse to the n-th port if the n-th tool execution was successful.
 * @param {IMainArguments} input_result - object with results of the job.
 * @returns {IMainResult} - object defining outputs per interface.
 */
export function onResults(input_result) {
    // For each tool for the FlowBuilder
    // take the success value, if positive send HIGH pulse
    let gpio_results = [];
    for (let i = 0; i < input_result.tool_results.length; ++i) {
        if (DEVICE.gpio.ports[i].state === GpioPortState.OUTPUT) {
            if (input_result.tool_results[i].success) {
                gpio_results.push({
                    port_number: i,
                    polarity: GpioPolarityResult.JOB_SETTINGS
                })
            }
        }
    }
    return {
        gpio: gpio_results
    };
}
```

## Beeper

Use the beeper field to control the beeper of the device and the Zebra Integrated Multi-Function Light.

```
beeper?: IBeepBeeper | BeeperStatusResult;
```

**Table 4** Helper Object Beeper Properties

| Value | Description |
|---|---|
| `Beeper.beep()` | This function triggers a beeper with a specified duration, tone, and volume. Helper objects, BeeperVolumeResult, BeeperToneResult, and BeeperDurationResult, can be used to specify these. |
| `Beeper.off()` | Does not trigger the beeper; it's independent of job_success. |
| `Beeper.device_settings()` | Triggers the beeper with parameters specified in the General Settings tab in Aurora Focus. |

**NOTE:** The beeper must be enabled using the General Settings tab in Aurora Focus before being used on the device.

```
/**
 * Beep if the overall grade is not good enough.
 * @param {IMainArguments} input_result - object with results of the job.
 * @returns {IMainResult} - object defining outputs per interface.
 */
export function onResults(input_result) {
    const decodes = input_result.tool_results_by_names.Read_Barcode.decodes;
    const bad_decodes = decodes.filter(decode => decode.score === null ||
        decode.score.overall_grade.grade < 3)
    const bad_values = bad_decodes.map(decode => decode.value + ", grade: " +
        decode.score?.overall_grade.grade);
    if (bad_values.length > 0) {
        console.error("Codes with bad grades: ", bad_values.join(" "))
        return {
            beeper: Beeper.beep(BeeperVolumeResult.HIGH,
 BeeperToneResult.MEDIUM,
                BeeperDurationResult.MEDIUM)
        };
    } else {
        return {
            beeper: Beeper.off()
        };
    }
}
```

## Script Invocation

The script executes at the end of each job when results are available. Its execution time includes the job's execution time.

The `onResults` function executes new arguments at the end of each job iteration. If there are any errors in the script, a standard JavaScript exception is thrown and propagated to Aurora Focus with no outputs set.

**NOTE:** For more information, refer to Debugging.

During the first iteration, the script execution may take longer because script code not enclosed in functions is executed first, such as initializing global variables.

**NOTE:** All other forms of result formatting are disabled when JavaScript is enabled.

## Example Script Lifecycle

The lifecycle of the script includes four stages: deployment, two iterations, and edit mode.

In the deployment stage, the execution environment is prepared without actions. In the first iteration, global variables are initialized, and syntax errors are checked. The `onResults` function processes the `Test String`, updates counters, and logs results. The second iteration skips initialization and processes the `Wrong String`, adjusts counters, and logs accordingly. Finally, the script enters edit mode, unloading from memory, with variables set to reinitialize upon redeployment.

```
console.log("Script loading begin");
let no_match_counter = 0;
let match_counter = 0;
const match_string = "Test String";
let counter = 0;

/** Returns the OutputResult type object
* @param {InputResult} inputResult
* @returns {OutputResult}*/
export function onResults(inputResult) {
  console.log("onResult called count: ", counter);
  counter++; // select first decode for first tool
  if (inputResult.tools[0].decodes[0] === match_string) {
      match_counter++;
  } else {
      no_match_counter++;
      console.error("No match string find!");
      return {
          common: "Match count: " + match_counter + " No match count: " +
 no_match_counter
      };
  }
  console.log("Script loading end")
```

There are four main stages in the lifecycle of a script:

1. Deploy the job to initialize the script's execution environment.
2. Iteration 1:
   a. The code outside the `onResults` the function is executed. The global variables `no_match_counter, match_counter, match_string` and `counter` are created and initialized. The `console.log` functions are executed. Any syntax errors in the script are reported at this stage, and its execution is performed if errors are detected.
   b. The `onResults` the function is executed. Assume that in this iteration, the read code contains a string.g `Test String`.
   c. Values are set, and logs are sent during this stage.
      a. The following string is sent to all available communication channels: `Match count: 1 No match count: 0`
      b. Aurora Focus receives logs in the following order: `Script loading begins`, `Script loading end`, `onResult called count: 0`
      c. Global variables at the end of the script are as follows: `no_match_counter = 0`, `match_counter = 1, counter = 1`
3. Iteration 2:
   a. No initialization phase.
   b. The `onResults` the function is executed. Assume that in this iteration, the read code contains the string.g `Wrong String`.
   c. Values are set, and logs are sent during this stage.
      a. The following string is sent to all available communication channels: `Match count: 1 No match count: 1`
      b. Aurora Focus receives logs in the following order: `onResult called count: 0` and `No match string find!`. The second log belongs to a different category, as noted in the editor console.
      c. Global variables at the end of the script are as follows: `no_match_counter = 1`, `match_counter = 1, counter = 2`
4. Edit the job and unload the script from memory.

**NOTE:** During the next deployment, the variables from Step 2a are initialized.

**NOTE:** Each execution is always the first one when using an emulator or performing jobs on the camera while editing.

## Managing Execution Time

The JavaScript execution time may be unstable at times. Avoid instability by understanding its sources, just-in-time compiler operation, and garbage collector usage.

While smaller scripts typically do not result in instability, observed irregularities in script execution times result from two primary sources:

- Just-in-time compiler operation. When developing a script, subsequent iterations after the initial script are accelerated.

- The JavaScript memory model uses a garbage collector. Scripts that use a lot of memory must periodically use the garbage collector to clean up unused memory during execution, reducing the time it takes to execute.

## Text Encoding

The read content of the barcode, data code, or other tools offering output as text is treated as a single-byte encoding in accordance with ISO-8859-1.

In the script, assume that you receive an array of bytes as they display within the string. Output data to communication channels is sent in the same form as presented in the script, byte by byte, without any encoding.

Use data visualization to display the output structure within the console object with ASCII encoding.

All non-printable characters are printed as `<CONTENT>`.

For example, the code `0x8F` in hexadecimal notation (not a valid ASCII character) is displayed as `<0x8F>`.

# Debugging, Advanced Techniques, and Sample Scripts

This section includes debugging tactics and sample scripts to implement into your workflow.

## Debugging

After you have mounted the camera, you can use basic script debugging features. The device always sends script execution data when in deploy mode.

The editor has a console at the bottom that displays the following data:

- The console displays the results returned by the `onResults` function. The results must match the expected field names of the structure. Data that does not match the schema is skipped.
- Use of the console object display in the console.
- Syntactic or runtime issues are reported to the console, as well as all JavaScript exceptions.

The object console provides the following methods for the logs category: `console.log`, `console.warn`, `console.error`, and `console.debug`. They work similarly to the `console.log` function found in environments such as web browsers. The function accepts any number of arguments that can be concatenated into a single string. The objects passed are converted to text strings according to the standard rules of the JavaScript language.

Chained instructions, multiple reassignments, nested loops, and other complex expressions might be challenging to understand without providing insight into how the data is transformed. The following code snippet does not sufficiently explain its function.

```
export function onResults(input_result) {
 if (input_result.tool_results[0].decodes.length === 0) { // do not process
 when there are no decodes
     return {};
 }
 const orderedDecodes = input_result.tool_results[0].decodes.map(decode =>
 decode.value).sort().join(',');
 return {
     tcp_ip: orderedDecodes,
 };
}
```

Decodes are transformed when the console object is valid. Split the steps to separate assignments and log the actual values on the console. This method tracks the progress of the data transformation step by step and quickly tracks bugs. The following example uses the console object.

```
export function onResults(input_result) {
 if (input_result.tool_results[0].decodes.length === 0) { // do not process
 when there are no decodes
     return {};
 }
 const decodes = input_result.tool_results[0].decodes;
 const decodedValues = decodes.map(decode => decode.value);
 console.log('decoded values:', decodedValues);
 const sortedValues = decodedValues.sort();
 console.log('sorted values:', sortedValues);
 const joinedResult = sortedValues.join(',');
 console.log('joined result: ', joinedResult);
 return {
     tcp_ip: joinedResult,
 };
}
```

Each step is logged in the console, and each step of the decode transformation from the raw decodes to a single string is observable. Use different colors in the console. `log`, `warning` or `error` methods to improve readability.

**NOTE:** While it is helpful to debug this way, the first approach may be preferable in production code because it does not declare many variables and uses less memory. After you validate the code, writing a concise version is recommended. If it is essential to keep the debug version of the code, comment it out and keep it for future reference to understand its purpose.

```
export function onResults(input_result) {
 if (input_result.tool_results[0].decodes.length === 0) { // do not process
 when there are no decodes
     return {};
 }

 // map all decodes to a single string where all decodes are separated by a
 comma and ordered alphabetically
 const orderedDecodes = input_result.tool_results[0].decodes
     .map(decode => decode.value) // map array of decodes to array of docoded
 values
     .sort() // sort alphabetically
     .join(','); // join elements of the array with a comma to a single
 string
 return {
     tcp_ip: orderedDecodes // output thorugh tcp
 };
}
```

## Using Simulated Data

If you cannot use real-world results, use simulated data until you can validate with representative data.

Assign the data to a variable to simulate an actual result of a camera read. The following example simulates the results of a ManyCode read.

```
export function onResults(input_result) {
   if (input_result.tool_results[0].decodes.length === 0) { // do not process
 when there are no decodes
       return {};
   }

   // mock the data for development and testing purposes, replace it with the
 actual data when possible
   const mockedData = ["D21OCT21", "78B8D65C7DE9", "S21294520180521",
 "1PFS10-SR10F1-1C00W"];
   //const orderedDecodes = input_result.tool_results[0].decodes // uncomment
 this line when we switch to actual data
   //.map(decode => decode.value)                                 // uncomment
 this line when we switch to actual data
   const orderedDecodes = mockedData // remove or comment out this line when
 we switch to actual data
       .sort()
       .join(',');
   return {
       tcp_ip: orderedDecodes
   };
}
}
```

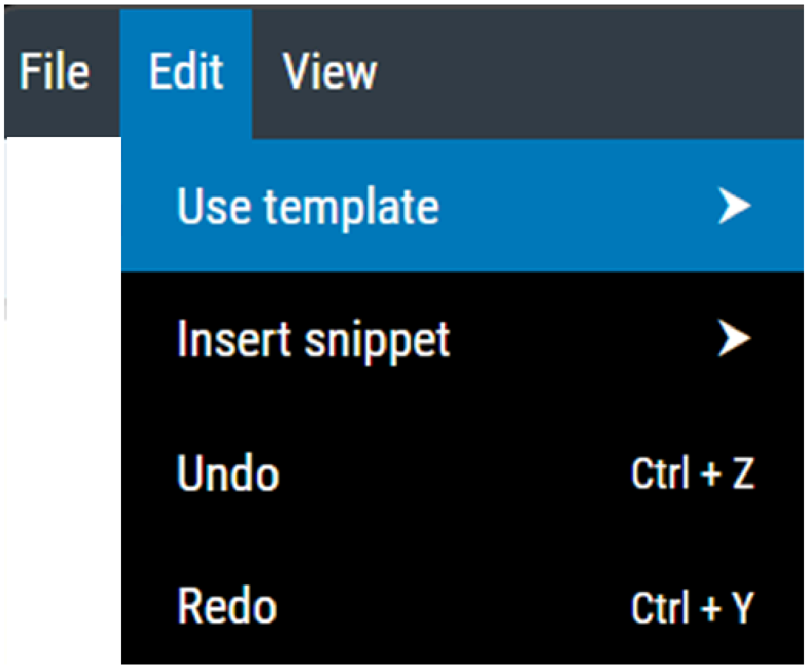
**WARNING:** Simulated data is hardcoded and user-defined. Use this method only for development and initial testing purposes. Conduct final tests using actual results to ensure the code works as intended.

## Using Custom Templates

Integrate your own JavaScript templates into Aurora Focus.

Create your template in a file with a .js extension, locate the Aurora Focus folder (where the application is installed), and save it in the `resources\public\js_editor_templates` subfolder.

After restarting the application, the template is available in the **Use template** section under **Edit** in the top navigation bar.



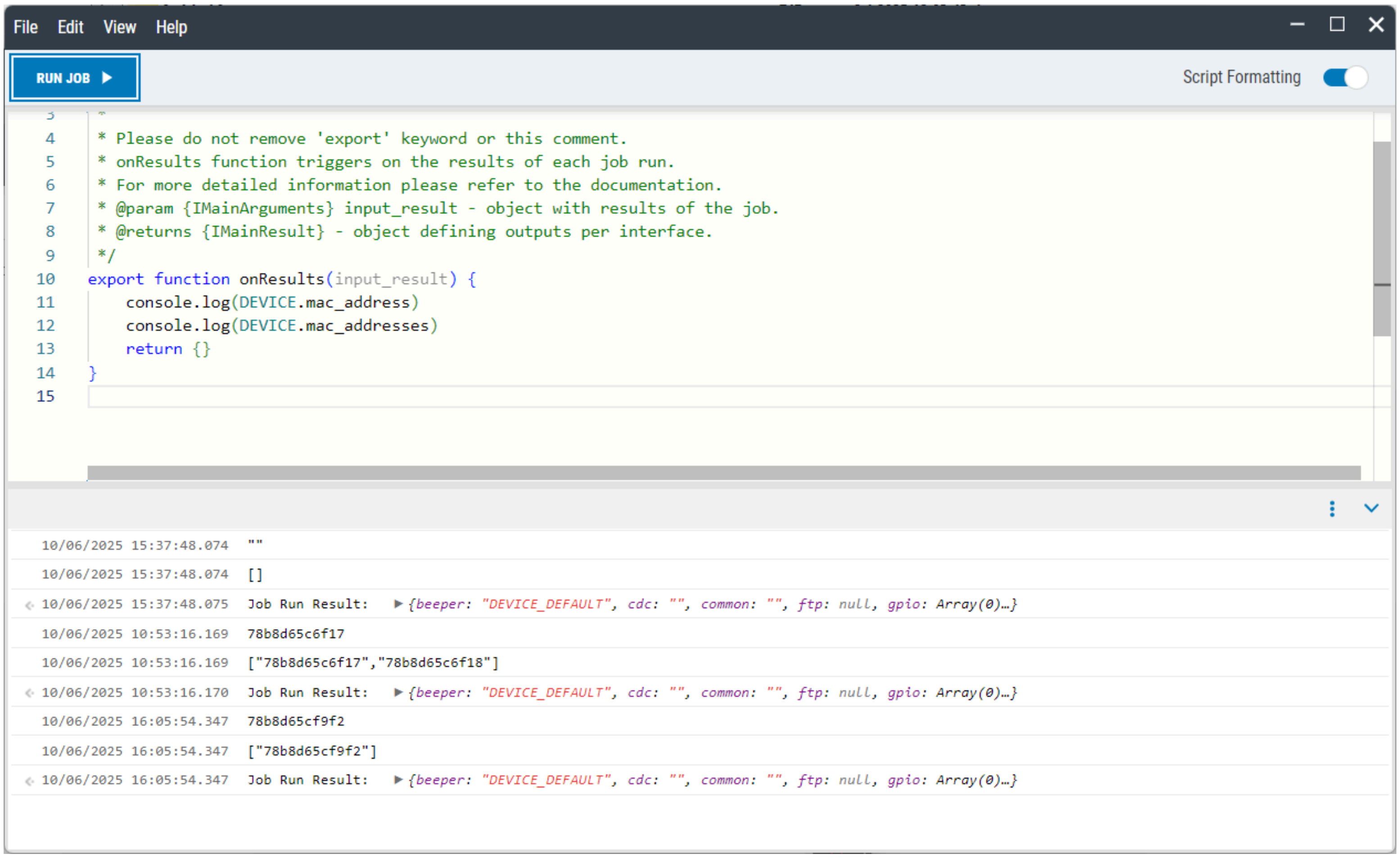
**NOTE:** The custom template you create is only available in that instance of Aurora Focus. If the file is deleted, the template is no longer available.

## Obtaining the Device MAC Address

Use the Aurora Focus JavaScript functionality to obtain the device's MAC address.

Use the following script to retrieve the device MAC address:

```
export function onResults (input_result) {
console.log(DEVICE.mac_address)
console.log(DEVICE.mac_addresses)
return {}
}
```



```
 * Please do not remove 'export' keyword or this comment.
 * onResults function triggers on the results of each job run.
 * For more detailed information please refer to the documentation.
 * @param {IMainArguments} input_result - object with results of the job.
 * @returns {IMainResult} - object defining outputs per interface.
 */
export function onResults(input_result) {
    console.log(DEVICE.mac_address)
    console.log(DEVICE.mac_addresses)
    return {}
}
```

```
10/06/2025 15:37:48.074  ""
10/06/2025 15:37:48.074  []
10/06/2025 15:37:48.075  Job Run Result:  ▶{beeper: "DEVICE_DEFAULT", cdc: "", common: "", ftp: null, gpio: Array(0)…}
10/06/2025 10:53:16.169  78b8d65c6f17
10/06/2025 10:53:16.169  ["78b8d65c6f17","78b8d65c6f18"]
10/06/2025 10:53:16.170  Job Run Result:  ▶{beeper: "DEVICE_DEFAULT", cdc: "", common: "", ftp: null, gpio: Array(0)…}
10/06/2025 16:05:54.347  78b8d65cf9f2
10/06/2025 16:05:54.347  ["78b8d65cf9f2"]
10/06/2025 16:05:54.347  Job Run Result:  ▶{beeper: "DEVICE_DEFAULT", cdc: "", common: "", ftp: null, gpio: Array(0)…}
```

## Sample Scripts

The following sample scripts demonstrate different scenarios and editor features.

Click **Edit** on the top menu bar in the **Use template** section to use additional sample scripts. Each template has comments that describe the general purpose and blocks of code.

```
/**
* Read all codes using ManyCode, order them alphabetically, separate them
 with a comma, and output them over TCP.
* Ensure that the Barcode Tool is in the first position in this job—
otherwise, the code might return errors and not work.
* Please do not remove the 'export' keyword or this comment.
* onResults function triggers on the results of each job run.
* For more detailed information, please refer to the documentation.
* @param {IMainArguments} input_result - object with results of the job.
* @returns {IMainResult} - object defining outputs per interface.
*/

export function onResults(input_result) {
  if (input_result.tool_results[0].decodes.length === 0) { // do not process
 when there are no decodes
```

```
      return {};
  }

  const orderedDecodes = input_result.tool_results[0].decodes
      .map(decode => decode.value) // map array of decodes to array of
 decoded values
      .sort() // sort alphabetically
      .join(','); // join elements of array with a comma to a single string
  return {
      tcp_ip: orderedDecodes // output through tcp
  };
}

/**
* Capture barcodes with Manycode and compare two strings. If they're the same
 output value, otherwise output "FAIL" through TCP and HID.
* Please do not remove the 'export' keyword or this comment.
* onResults function triggers on the results of each job run.
* For more detailed information, please refer to the documentation.
* @param {IMainArguments} input_result - object with results of the job.*
 @returns {IMainResult} - object defining outputs per interface. */

export function onResults(input_result) {

  // return fail because there are no decodes to compare

  if (input_result.tool_results[0].decodes.length < 2) {
      return {
          job_success: false,
      };

  }
  // at this point, we are sure that there are at least two decodes here due
 to the check above
  // assign first and second decode value to variables for readability


  const firstDecode = input_result.tool_results[0].decodes[0].value;
  const secondDecode = input_result.tool_results[0].decodes[1].value;

  // compare decodes and return the decode value through TCP and HID if they
 are equal
  // set job_success to true, because the job did get two reads and did run
 correctly

  if (firstDecode === secondDecode) {

      return {

          tcp_ip: firstDecode,
          hid: firstDecode,
          job_success: true,
      };
  }

  // return "FAIL" because the first and second decode differ, and
 job_success because the job did get two reads and did run correctly,
  // even though the decodes did not match

  return {

      tcp_ip: "FAIL",
      hid: "FAIL",
      job_success: true, // job success is true even though the decodes do
 not match - the job did run
  };

}
```