Connect IoT - Raw File and CSV Tutorial

12.0.1

November 2026

DOCUMENT ACCESS

Public

DISCLAIMER

The contents of this document are under copyright of Critical Manufacturing S.A. it is released on condition that it shall not be copied in whole, in part or otherwise reproduced (whether by photographic, or any other method) and the contents therefore shall not be divulged to any person other than that of the addressee (save to other authorized offices of his organization having need to know such contents, for the purpose for which disclosure is made) without prior written consent of submitting company.

Connect IoT - Raw File and CSV Tutorial

Overview

Many machines exchange data through files: a tester writes one result file per run into a folder, a gauge writes one CSV file per lot, and some equipment waits for a file from the host before it starts. Connect IoT covers these cases with two drivers that share the same folder watcher:

- File Raw - reports the files that appear, change, or disappear in a folder, and runs file operations on request. It never opens a file on its own: the workflow reads the content with a command and parses it.
- CSV - watches a folder the same way, opens each CSV file, and sends its rows as events, one at a time. It reads the next row only when the workflow asks for it.

Choose the CSV driver only when each row must be handled and confirmed before the next one, or when the equipment appends rows to a file while it is read. For any other format, and for a CSV file read whole, use the File Raw driver and parse the text in the workflow.

This tutorial is for integrators who completed Connect IoT - Basic Configuration Tutorial and know the Data Flow designer. You build one **Automation Controller** with two drivers for a **Resource**, such as `Tester-01`:

- `Results` (File Raw) - writes a ready file when the connection comes up, then reads each new result file, logs its text, and moves the file to a `done` folder.
- `Gauge` (CSV) - logs the item of each row of a lot file and asks for the next row.

At the end, the **Automation Manager** console shows the content of each result file and each CSV row in order, and the watched folder holds only the files that nobody handled.

> **Info**
>
> For the protocol parameters, the shipped events, and the commands of both drivers, see Connect IoT - Raw File Driver and Connect IoT - CSV File Driver.

Before You Start

- Complete Connect IoT - Basic Configuration Tutorial: the **Resource** entity type is enabled for Connect IoT, your user is an integration user, and an **Automation Manager**, such as `FileTutorialManager`, is downloaded and runs in console mode.
- Create two folders on the machine, or in the container, where the **Automation Manager** runs, for example `C:/Tester/Results` and `C:/Gauge/Out`. The drivers resolve every path on that machine, and a watched folder must exist before the connection starts.
- Play the equipment yourself: create or copy files into the folders with any text editor. To run the same exercise as an automated test, the Test Orchestrator can run this controller against your MES, with its File-Raw simulator owning the folders in place of the equipment. See Connecting to a Live MES ↗. The Test Orchestrator is a .NET test framework, currently in alpha.

Sequence of Steps

Step 1: Create the Two Automation Protocols

Each driver is an npm package that an **Automation Protocol** pins with its version. Create one protocol per driver, as in How to: Create an Automation Protocol, and keep the default parameter values. You set the folder of each connection in the controller, in Step 4.

Name	Package
`File Raw`	`@criticalmanufacturing/connect-iot-driver-fileraw`
`CSV`	`@criticalmanufacturing/connect-iot-driver-csv`

Table 1: Automation Protocols of the tutorial

Both protocols are in the `Created` state. The parameters of `CSV` are those of `File Raw` plus six parser parameters, such as `hasHeader`, `delimiter`, and `numberOfLines`.

Step 2: Create the Raw File Driver Definition

The File Raw package ships a complete definition: nine properties, three events, and twelve commands. The driver takes the meaning of each entry from its extended data (`propertyType`, `eventTrigger`, or `commandType`) and ignores its device ID. A

definition for this protocol therefore declares nothing of its own, and the workflow selects the shipped entries by name.

1. Create an **Automation Driver Definition**, such as `Tester Results`, with the **Automation Protocol** `File Raw` and the **Entity Type** `Resource`, as in Create Automation Driver Definition.
2. Select **Next** on the properties, events, and commands steps without adding entries, and select **Create**.

The workflows of this tutorial use these shipped entries:

Entry	Name	Behavior
Event	`OnNewFile`	Raised when a file that matches `fileMask` appears. Carries `FileName`, `RelativePath`, `FullPath`, `Size`, and the file times.
Command	`ReadFile`	Replies with the whole content of the file in `path`, decoded with `encoding`.
Command	`CreateDirectory`	Creates the folder in `path` and its missing parents. Does nothing when the folder exists.
Command	`Move`	Moves `source` to `destination`. Into an existing folder, the file keeps its name.
Command	`CreateFile`	Writes `content` to the file in `path`. The `flag` `a` appends instead of replacing.

Table 2: Shipped File Raw entries used in the tutorial

Each of these commands also takes `attempts` (default `1`) and `sleepBetweenAttempts` (default `1000` milliseconds): a failed command is retried before the task emits `Error`. Take the name and path of a file from the outputs of the event, not with `Get Equipment Properties`: the shipped properties describe the last file the watcher reported, and another file may have arrived since.

Step 3: Create the CSV Driver Definition with the Columns

The CSV package ships the file entries, the `MoreDataRequest` command, and the `OnDataAvailable` event, but no column: the package cannot know them. Declare each column the workflow needs as a property, and an event that carries the columns. The gauge of this exercise writes one CSV file per lot, one item per row:

```
ItemId,Thickness
W-01,178.2
W-02,181.5
```

1. Create an **Automation Driver Definition**, such as `Gauge Rows`, with the **Automation Protocol** `CSV` and the **Entity Type** `Resource`.
2. Add the following properties. Set the **Protocol Data Type** to the same value as the **Type**. For a column, the **Property Device ID** is the header text, character for character.

Name	Property Device ID	Type	Extended data
`ItemId`	`ItemId`	`String`	`propertyType` `FileContent`, `identifierType` `Name`
`Thicknes s`	`Thickness`	`Decimal`	`propertyType` `FileContent`, `identifierType` `Name`
`FullPath`	`FullPath`	`String`	`propertyType` `FullPath`
`EndOfFil e`	`EndOfFile`	`Boolean`	`propertyType` `EndOfFile`

Table 3: Properties of the CSV driver definition

3. Add an event, such as `OnItemRow`, with the extended data `eventTrigger` set to `NewData`, and link the four properties to it.
4. Select **Create**.

The driver raises every `NewData` event of the definition for each row, and converts the text of each column to the **Type** of its property. The shipped `OnDataAvailable` is also a `NewData` event, but it carries only `FullPath` and `EndOfFile`, so a workflow that listens to it receives each row without its values. `FullPath` and `EndOfFile` are declared again because the event properties step of the wizard lists only the properties of the definition itself.

Step 4: Create the Controller and Configure Both Connections

1. Create an **Automation Controller**, such as `File Tutorial Controller`, as in Create Automation Controller, with the **Scope** `Connect IoT`, the **Workflow Type** `Data Flow`, and the **Entity Type** `Resource`.

2. Add two driver definitions: `Results` with `Tester Results`, and `Gauge` with `Gauge Rows`. Keep only the core tasks package: the File Drivers tasks are deprecated, and the core `On Equipment Event` and `Execute Equipment Command` tasks replace them.
3. Select **Create**. The **Setup** page holds one chain for each driver: `Driver Event` (`OnInitialize`), `Equipment Configuration`, and `Driver Command` (`Connect`). See Create the Setup Workflow.
4. Open the settings of each `Equipment Configuration` task and set the following values on the **Communication Parameters** tab. Keep the other parameters at their defaults.

Driver	Parameter	Value	Effect
`Results`	`path`	`C:/Tester/Results`	The folder to watch.
`Results`	`fileMask`	`*.txt`	Reports result files only.
`Gauge`	`path`	`C:/Gauge/Out`	The folder to watch.

Table 4: Communication parameters to set

Some defaults matter for this exercise. With `ignoreInitial` `false`, every file already in the folder is reported when the connection starts, on every start. With `awaitWriteFinish` `true`, a file is reported only after its size holds still for 2000 milliseconds, so the equipment has finished writing it. With `depth` `0`, subfolders are not watched, so a `done` folder inside the watched folder is a safe place for handled files. On `Gauge`, keep `numberOfLines` at `1`: with more lines per block, the `FileContent` properties receive no value.

The driver receives the communication parameters when it connects. After you change a parameter of a connection that is already `Communicating`, restart the controller instance.

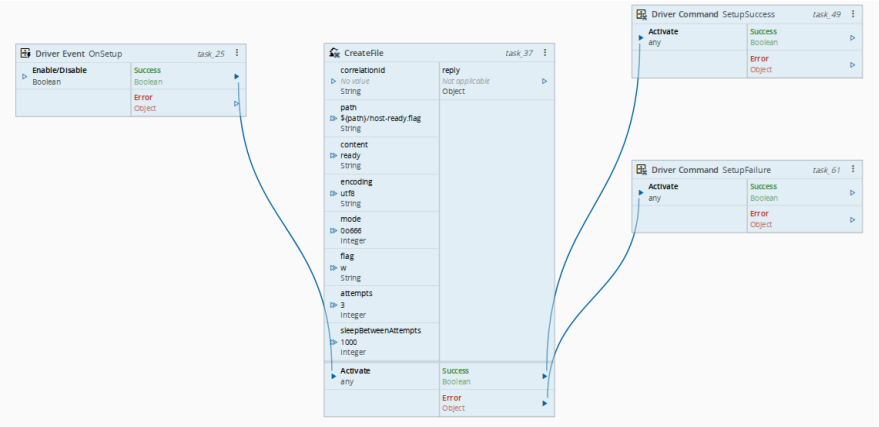
Step 5: Answer the Setup with a Ready File

Some file-based equipment waits for a file from the host before it starts. Write that file during the setup phase of the `Results` connection, and complete the setup only after the file exists.

1. On the **Setup** page, open the settings of the `Driver Event` task of `Results` and disable **Automatic Setup**.
2. Add the following tasks for the driver `Results` and link them. In the link columns of this tutorial, an `Execute Equipment Command` task is named after its command.

Task	Settings to set	Inputs/Outputs to link
`Driver Event`	**Event** `OnSetup`	-
`Execute Equipment Command`	**Equipment Command** `CreateFile`; values `path` `${path}/host-ready.flag`, `content` `ready`, `mode` `438`, and `attempts` `3`	`Activate` from `Driver Event` `Success`
`Driver Command`	**Command** `SetupSuccess`	`Activate` from `CreateFile` `Success`
`Driver Command`	**Command** `SetupFailure`	`Activate` from `CreateFile` `Error`

Table 5: Tasks that answer the setup phase



The driver replaces `${path}` with the watched folder, so the ready file follows the connection when `path` changes. The `.flag` extension keeps the file out of `fileMask`, so the watcher does not report it as a result. `438` is the decimal form of the permission `0o666`. The connection stays in the `Setup` state until a `Driver Command`

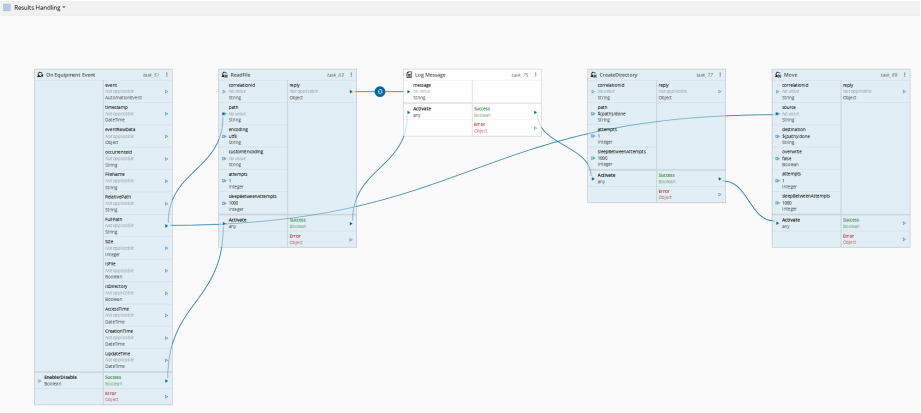
reports the result: an `OnSetup` handler that never sends `SetupSuccess` fails the setup after the **Setup Timeout** of the `Equipment Configuration` task, on every attempt. For the two ways to complete the setup phase, see Create the Setup Workflow.

Step 6: Read Each New Result File and Move It to Done

Create a Data Flow page, such as `Results Handling`, and add the following tasks for the driver `Results`. Each task runs from the `Success` of the task before it, so nothing moves a file that was not read and logged.

Task	Settings to set	Inputs/Outputs to link
`On Equipment Event`	**On Equipment Event** `OnNewFile`	-
`Execute Equipment Command`	**Equipment Command** `ReadFile`	`path` from the event `FullPath`; `Activate` from the event `Success`
`Log Message`	**Mode** `RawText`	`message` from `ReadFile` `reply`, through the converter `Any To String`; `Activate` from `ReadFile` `Success`
`Execute Equipment Command`	**Equipment Command** `CreateDirectory`; value `path` `${path}/done`	`Activate` from `Log Message` `Success`
`Execute Equipment Command`	**Equipment Command** `Move`; value `destination` `${path}/done`	`source` from the event `FullPath`; `Activate` from `CreateDirectory` `Success`

Table 6: Tasks of the Results Handling page



Three rules keep the files where you expect them:

- Link `FullPath`, never `FileName`. The driver resolves a relative path against its own working folder, not against `path`, so a bare file name reads or writes the wrong file.
- Set `destination` to the folder and create the folder just before the move. When `destination` does not exist, `Move` takes it as the new file name, so the first file becomes a file called `done` and every later move fails.
- End the handling of every file by taking it out of the watched folder. With `ignoreInitial` `false`, a file left in the folder is reported again on every start. With `ignoreInitial` `true`, the files that arrive while the controller is down are never reported.

`Log Message` writes only when its `Activate` input receives a value, so the `message` link alone logs nothing.

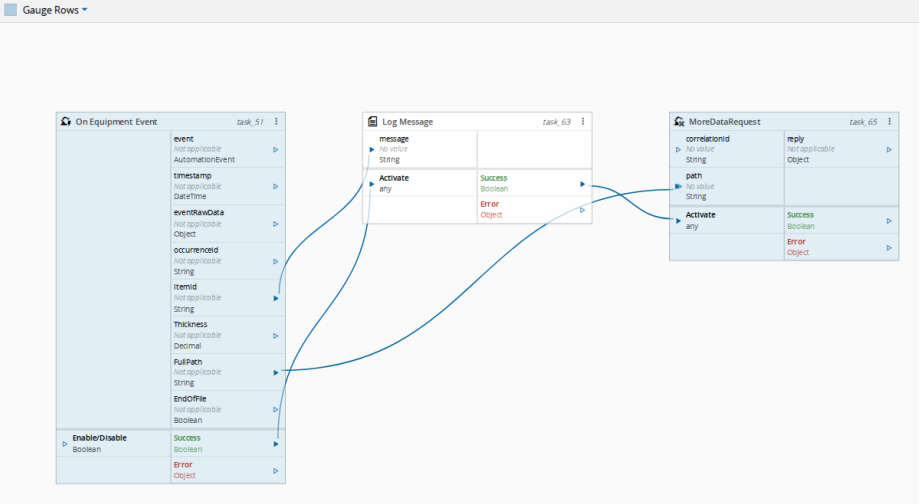
> **Info**
>
> Data Flow keeps one instance of each task per workflow. When a second file arrives before the first is moved, its `FullPath` replaces the first one in the waiting inputs. See Connect IoT - How Values Travel Tutorial. To build the same handler in Control Flow, place the tasks in the handler of the trigger in the same order, and type the `reply` row `Any`: an `Object` row parses the text as JSON and fails. See Control Flow Expressions.

Step 7: Read the CSV Rows One at a Time

Create a Data Flow page, such as `Gauge Rows`, and add the following tasks for the driver `Gauge`:

Task	Settings to set	Inputs/Outputs to link
`On Equipment Event`	**On Equipment Event** `OnItemRow`	-
`Log Message`	**Mode** `RawText`	`message` from the event `ItemId`; `Activate` from the event `Success`
`Execute Equipment Command`	**Equipment Command** `MoreDataRequest`	`path` from the event `FullPath`; `Activate` from `Log Message` `Success`

Table 7: Tasks of the Gauge Rows page



The driver and the workflow take turns. The driver raises `OnItemRow` with the first row and waits. `MoreDataRequest` confirms the row just handled, raises the event with the next row, and replies `true`. When no row is left, it replies `false` and raises nothing. Follow these rules:

- Link `path` from the event `FullPath`, exactly. The driver applies no token to this command, and any other path fails with `Unable to fetch more data in file`.
- Ask for the next row after every row, the last one included, where `EndOfFile` is `true`. The driver stores how far it read a file only when the next request arrives, so after a restart the last unconfirmed row is sent again. A row handler that writes to MES must tolerate a row it has already seen.

- Run once-per-file work, such as moving the file, after the final request, and read one file at a time: after a `MoreDataRequest`, `FullPath` can belong to another file.

Step 8: Connect the Controller and Start It

1. Connect the controller to the **Resource** `Tester-01` and the **Automation Manager**, as in How to Create a Controller Instance. On the drivers step, select the same **Resource** for `Results` and for `Gauge`.
2. Start the **Automation Manager** in console mode.

On the Connect IoT page, both driver instances reach `Communicating`, and `host-ready.flag` appears in `C:/Tester/Results`. When a watched folder does not exist, the connection goes from `Connecting` to `Disconnected` and retries after each **Reconnect Interval**, without reaching `ConnectingFailed`: only the driver log names the folder, with `Unable to find path`.

Validate the Result

1. Create a file `run-0042.txt` in `C:/Tester/Results` with the following content:

   ```
   lot=B-1001
   peak=212.5
   ```

   About two seconds later, the console shows the two lines logged by `Log Message`, and `run-0042.txt` is in `C:/Tester/Results/done`.
2. Restart the controller instance on the Connect IoT page. No result is reported again: the folder holds no file that matches `*.txt`. `host-ready.flag` is written again during the new setup.
3. Copy a file `lot-L1001.csv` with the content of Step 3 into `C:/Gauge/Out`. The console shows `W-01` and then `W-02`, each logged before the next row is requested.

Next Steps

Turn the text you read into data with a Code task, as in Connect IoT - Writing a Code Task Tutorial, and send the results to MES, as in Connect IoT - Talking to MES Tutorial. To compare protocols that push values instead of files, continue with Connect IoT - OPC UA Tutorial or Connect IoT - MQTT Tutorial. When a folder stays silent or a file lands in the wrong place, see Troubleshooting by Symptom.

Legal Information

Disclaimer

The information contained in this document represents the current view of Critical Manufacturing on the issues discussed as of the date of publication. Because Critical Manufacturing must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Critical Manufacturing, and Critical Manufacturing cannot guarantee the accuracy of any information presented after the date of publication. This document is for informational purposes only.

Critical Manufacturing makes no warranties, express, implied or statutory, as to the information herein contained.

Confidentiality Notice

All materials and information included herein are being provided by Critical Manufacturing to its Customer solely for Customer internal use for its business purposes. Critical Manufacturing retains all rights, titles, interests in and copyrights to the materials and information herein. The materials and information contained herein constitute confidential information of Critical Manufacturing and the Customer must not disclose or transfer by any means any of these materials or information, whether total or partial, to any third party without the prior explicit consent by Critical Manufacturing.

Copyright Information

All title and copyrights in and to the Software (including but not limited to any source code, binaries, designs, specifications, models, documents, layouts, images, photographs, animations, video, audio, music, text incorporated into the Software), the accompanying printed materials, and any copies of the Software, and any trademarks or service marks of Critical Manufacturing are owned by Critical Manufacturing unless explicitly stated otherwise. All title and intellectual property rights in and to the content that may be accessed through use of the Software is the property of the respective content owner and is protected by applicable copyright or other intellectual property laws and treaties.

Trademark Information

Critical Manufacturing is a registered trademark of Critical Manufacturing.

All other trademarks are property of their respective owners.

Copyright © 2023 Critical Manufacturing. All rights reserved.