Critical manufacturing
an ASM PT company

How Values Travel

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.

12.0

Copy Page

Connect IoT - How Values Travel Tutorial

Overview

A workflow reads like a short program: each task is a call that receives inputs, emits outputs, and then signals `Success` or `Error`. Many defects that look like bad equipment data come from how values move between those calls: a task that never runs, a value left from an earlier event, or a value that the next event replaces before it is used.

This tutorial is for integrators who completed Connect IoT - Basic Configuration Tutorial and build workflows in the **Automation Controller** designer. A File Raw driver plays the equipment, so you raise each event yourself by dropping a file into a folder, and you read each result on the **Automation Manager** console.

At the end, you have a Data Flow workflow that shows each mechanism one at a time, and a Control Flow workflow that handles overlapping files without mixing their values.

> **Info**
>
> For more information, see Create a Workflow - Data Flow, Create a Workflow - Control Flow, and Log Message.

Before You Start

- Prepare a File Raw integration: an **Automation Protocol** on the `@criticalmanufacturing/connect-iot-driver-fileraw` package, an **Automation Driver Definition** with the driver's shipped events, and an **Automation Controller** with that driver. Connect IoT - Control Flow Configuration Tutorial creates these entities, and Connect IoT - Raw File Driver lists the events `OnNewFile` and `OnFileDeleted` and the properties they carry.
- Set `path` in the `Equipment Configuration` task of the controller setup workflow to an empty folder, for example `c:/temp/Handler`. Keep `alwaysStat` at its default,

`true`, so that `OnNewFile` carries `Size`.

- Run an **Automation Manager** in console mode with an instance of the controller.
- Raise the events by hand, or from an automated test with the Test Orchestrator File-Raw simulator ↗, which creates and deletes files in the watched folder.

> **Note**
>
> From version 12.0, the File Drivers tasks, such as `New File Event`, are deprecated. Listen to file events with `On Equipment Event`, as this tutorial does, and run file operations with `Execute Equipment Command`. See the File Raw tasks migration guide ↗.

Sequence of Steps

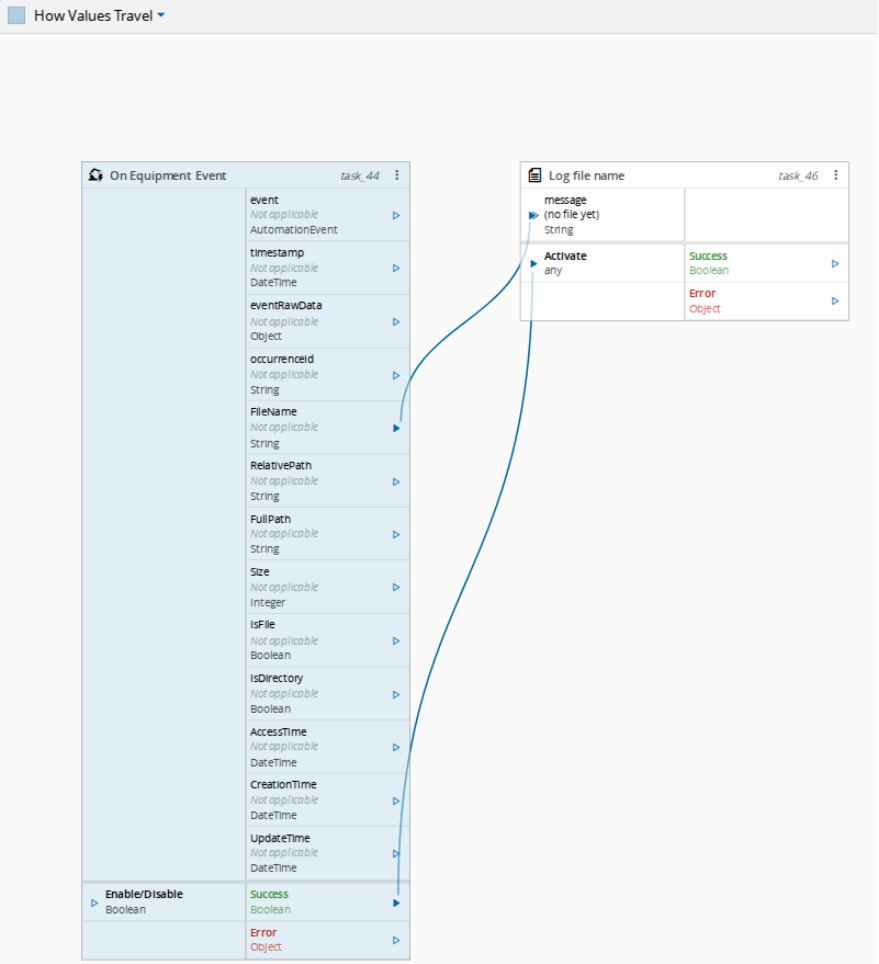
Step 1: Log Each New File

A task runs when a value reaches its `Activate` input. A value that reaches any other input waits there until the task is activated.

1. In the **Automation Controller**, add a Data Flow workflow named, for example, `How Values Travel`.
2. Add an `On Equipment Event` task and set **Equipment Event** to `OnNewFile`. Its outputs include `FileName` and `Size`.
3. Add a `Log Message` task named `Log file name`, with **Mode** `RawText` and **Default Message** `(no file yet)`.
4. Link the `FileName` output to the `message` input of `Log file name`, save, and drop a file, such as `a.csv`, into the watched folder.

   No line from `Log file name` appears: a value on `message` does not run the task.
5. Link the `Success` output of `On Equipment Event` to the `Activate` input of `Log file name`, save, and drop `b.csv`.

   The console shows `b.csv`.



Every task has `Activate`, `Success`, and `Error`. A task emits its own outputs first and `Success` last, so a task activated by `Success` finds the values already on its inputs. Link the values that a task reads, then link `Success` to `Activate` to say when to read them. Switch follows the same rule: it compares only when a value reaches `Activate`.

> **Note**
>
> From version 12.0, `Log Message` and `Switch` run only from `Activate`. The 12.0.0 migration adds, in Data Flow workflows, an `Activate` link from each output already linked to these tasks. See the normalized tasks migration guide ↗.

Step 2: Activate the Log From Another Event

In Data Flow, one instance of each task serves the whole workflow, and an input keeps its value until something clears or replaces it. `Log Message` clears its inputs after each line while **Reset inputs on activate** is on, its default.

1. Add a second `On Equipment Event` task with **Equipment Event** `OnFileDeleted`, link its `Success` to the `Activate` input of `Log file name`, and save.
2. Drop `c.csv`, and then delete it.

   The console shows `c.csv` when the file arrives and `(no file yet)` when you delete it. The first line cleared `message`, so the second activation falls back to **Default Message**.
3. Turn off **Reset inputs on activate**, save, and repeat with `d.csv`.

   The deletion now shows `d.csv` again: the task reads a value that another event left on its input, and nothing in the log says so.
4. Turn **Reset inputs on activate** back on, delete the `OnFileDeleted` task, and save.

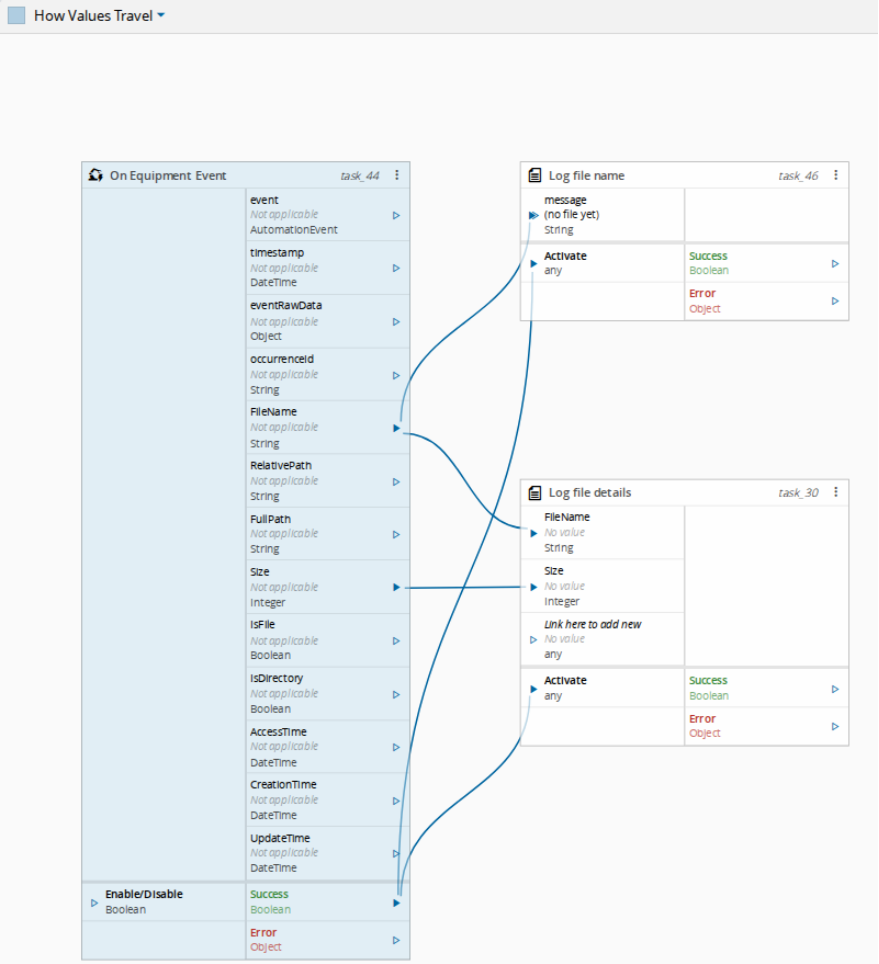
Tasks differ in what they keep between runs. `Execute Equipment Command`, for example, keeps its arguments unless **Clear Inputs (on activate)** is on, and that setting is off by default. Check the reference page of each task, and feed each chain from one event, so that every activation brings every value the task reads.

Step 3: Log Several Values Through Added Ports

Some tasks cannot declare in advance the values they carry, so you add their ports. Linking an output to the `Link here to add new` input of such a task adds a named input.

1. Add a `Log Message` task named `Log file details`, with **Mode** `MultipleInputs`.
2. Link the `FileName` and `Size` outputs of `On Equipment Event` to the `Link here to add new` input of `Log file details`, link `Success` to its `Activate`, and save.
3. Drop `e.csv`.

   `Log file details` shows `FileName = e.csv` and `Size = <bytes>`. Each label is the name of the added input.



The canvas labels an added port with the name you gave it, but each task stores the port under a name of its own. A Data Flow link uses the stored name without showing it, while a Control Flow input row or expression must write it:

Task	Added port on the canvas	Stored port name
`On Equipment Event`	output `FileName`	`FileName`
`Log Message`, `MultipleInputs` mode	input `FileName`	`FileNameIn`. In Control Flow, the task also reads a row named `FileName`

Task	Added port on the canvas	Stored port name
`Code`	input `Size`, output `status`	`SizeIn`, `statusOut`
`Synchronize`	`FileName`	`FileNameIn` and `FileNameOut`
`Store Data`, `Retrieve Data`	entry `count`	`count`

Table: Stored names of added ports

A wrong port name raises no error. The value does not arrive, and the task logs `<null>` or runs on a default value.

Step 4: Tell null From undefined

An output that emits `undefined` changes nothing on the input it reaches, so it activates nothing. An output that emits `null` is delivered like any other value, and it activates the task it reaches.

1. Add a `Code` task named `classify`, and on its **Outputs** tab add an output `status` of type `String`.
2. Link the `Size` output of `On Equipment Event` to the `Link here to add new` input of `classify`, which adds the input `Size`, and link `Success` to its `Activate`.
3. On the **Code** tab, make `main` return a status:

```
public async main(inputs: any, outputs: any): Promise<any> {
    return { status: inputs.Size > 0 ? "has data" : null };
}
```

4. Add a `Log Message` task named `Log status`, with **Mode** `RawText` and **Default Message** `(no status)`. Link the `status` output of `classify` to both the `message` and the `Activate` inputs of `Log status`, and save.
5. Drop a file with some text in it, and then an empty file, such as `empty.csv`.

   `Log status` shows `has data`, and then `(no status)`. The `null` reached `Activate` and ran the task, and the empty `message` fell back to **Default Message**.
6. Replace `null` with `undefined` in the code, save, and drop another empty file.

   `Log status` shows nothing: the `undefined` did not activate it.

7. Replace `undefined` with `"empty"`, save, and drop another empty file. `Log status` shows `empty`.

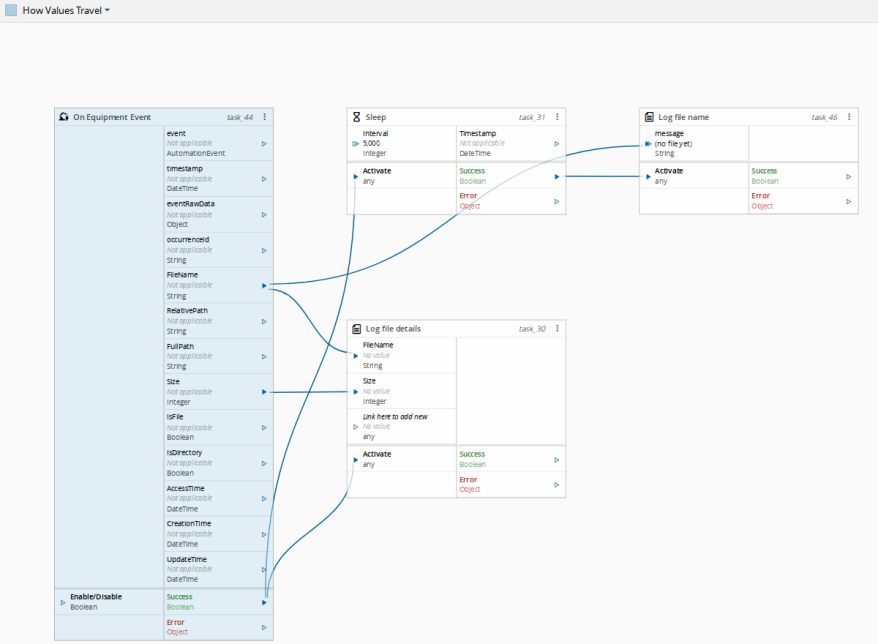
Give a value its fallback where you produce it, and do not emit `null`. The link from `status` to `Activate` only makes the difference visible: in a workflow, activate `Log status` from the `Success` of `classify`.

Step 5: Let Two Files Share One Task Instance

Overlapping events share every Data Flow task. When a second event reaches an input before the first run reads it, the second value replaces the first.

1. Add a `Sleep` task with **Interval** `5000`. Remove the link from `Success` of `On Equipment Event` to `Activate` of `Log file name`, link that `Success` to the `Activate` of `Sleep`, and link the `Success` of `Sleep` to the `Activate` of `Log file name`. Keep the link from `FileName` to `message`, and save.
2. Drop `f1.csv`, and drop `f2.csv` within five seconds.

   Five seconds after each file, `Log file name` shows `f2.csv`, and then `(no file yet)`. The second file replaced `f1.csv` on `message` while the first run waited, and the first line cleared it.



The `Sleep` only makes the window visible. Events that arrive closer together than their chain takes to run overwrite values in the same way. When a second value

reaches an input before the first was delivered, the controller warns `Task '<page>\ <task>' input '<input>' will receive value '<new>' instead of '<old>'`. No task or workflow setting makes Data Flow safe for fast events.

Step 6: Rebuild the Handler in Control Flow

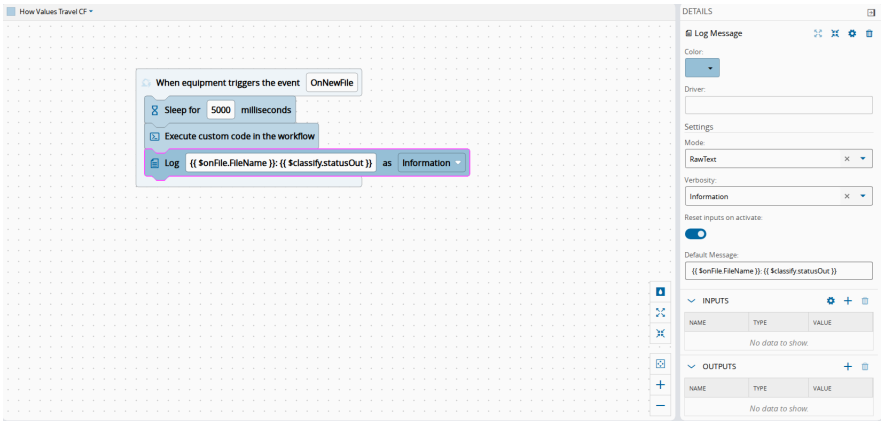
A Control Flow branch builds its tasks anew for each event, so each run reads only its own values. There are no links: an input row reads an earlier task through an expression, `{{ $<block name>.<stored output name> }}`. This step writes the `On Equipment Event` block as `onFile` and the `Code` block as `classify`; in your workflow, type `$` inside `{{ }}` and pick the block names that the designer suggests.

1. In the `How Values Travel` workflow, turn off **Auto Enable** on `On Equipment Event`, so that only the new workflow reacts to files, and save.
2. Add a Control Flow workflow named, for example, `How Values Travel CF`, and place an `On Equipment Event` task with **Equipment Event** `OnNewFile`.
3. Place these tasks in the handler of the event, in this order:

Task	Settings	Input rows
`Sleep`	**Interval** `5000`	-
`Code`	Input `Size` of type `Integer`, output `status` of type `String`, and the code from Step 4 with `"empty"`	`SizeIn`: `{{ $onFile.Size }}`
`Log Message`	**Mode** `RawText`, and **Default Message** `{{ $onFile.FileName }}: {{ $classify.statusOut }}`	-

Table: Tasks in the Control Flow handler

The field after `Log` on the block is **Default Message**. In Control Flow, `Log Message` evaluates the expressions in it.



4. Save, drop `g1.csv` with some text in it, and drop an empty `g2.csv` within five seconds.

   After each wait, the console shows `g1.csv: has data` and then `g2.csv: empty`, each line under the execution context of its own file.

An expression that reads `$classify.status` gets nothing, because `Code` emits its added output as `statusOut`. Handle a fast or overlapping event in Control Flow. Data Flow suits events spaced further apart than their chain takes to run, and workflows that switch their own triggers on and off.

Validate the Result

1. Drop three files within five seconds, one of them empty.

   `How Values Travel CF` shows one line per file, each with its own name and status. `How Values Travel` shows nothing, because its trigger is off.

2. Turn **Auto Enable** back on in `How Values Travel`, save, and repeat.

   `Log file details` and `Log status` show each file immediately. `Log file name` shows the name of a later file after the first wait and `(no file yet)` after the others. The Control Flow lines stay correct.

What You Learned

- A task runs when a value reaches `Activate`, and it emits its outputs before `Success`.
- A Data Flow task is one instance for the whole workflow: it keeps what its inputs received unless it clears them, as **Reset inputs on activate** does for `Log Message`.

- An added port has the name you gave it on the canvas and a stored name, such as `statusOut`, that Control Flow rows and expressions use.
- An `undefined` output activates nothing, and a `null` output is delivered.
- Overlapping events overwrite values in Data Flow, and Control Flow gives each event its own run.

Next Steps

Continue with Connect IoT - Control Flow in Depth Tutorial to branch, loop, and handle failures in Control Flow, and with Connect IoT - Timing and State Tutorial to add clocks, time limits, and values that outlive a run. The Connect IoT - Learning Path lists the full sequence.

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.