Critical manufacturing
an ASM PT company

Control Flow in Depth

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 - Control Flow in Depth Tutorial

Overview

In a Control Flow workflow, tasks are blocks nested under the event that starts them, and they run top to bottom. Each occurrence of the event gets a branch of its own, so a burst of events never overwrites the values of a run in progress. In exchange, nothing links one task to another: each task reads earlier results through expressions.

This tutorial is for integrators who completed Connect IoT - Basic Configuration Tutorial and Connect IoT - How Values Travel Tutorial. You build one workflow that handles each lot file dropped in a folder. Step by step, it reads the event values, routes the file by its size, goes through its lines, archives it with a retry, and asks a function, which any workflow of the **Automation Controller** can call, how to classify it. At the end, the **Automation Manager** console shows one ordered sequence of log lines per file.

Three rules apply to every step:

- Each occurrence builds a new branch. Its tasks are created when the event fires and released when the branch ends, so a run starts from the stored settings and sees nothing of the previous run.
- Settings are applied when the branch is built, before any task of the branch runs. Input rows are evaluated just before their task runs, so expressions go in input rows.
- A task reads the outputs of tasks that already ran, in its own branch or in a branch that encloses it, as `{{ $<block name>.<output> }}`.

Info

For the expression language, the row types, and the converter functions, see Control Flow Expressions. For the designer, see Create a Workflow - Control Flow.

Before You Start

Set up an **Automation Controller** with one File Raw driver, as in Connect IoT - Basic Configuration Tutorial, with these values:

Entity	Values
Automation Protocol	`File Raw`, with the package `@criticalmanufacturing/connect-iot-driver-fileraw`. See How to: Create an Automation Protocol.
Automation Driver Definition	`Lot Files`, with the protocol `File Raw`. The package ships the `OnNewFile` event and the `ReadFile` and `Move` commands that this tutorial uses.
Automation Controller	`Lot Files Controller`, with the **Workflow Type** `Control Flow`, one driver `Lots` with the definition `Lot Files`, and the core tasks package only. The File Drivers tasks are deprecated.
Communication parameters	On the **Equipment Configuration** task of the **Setup** page, `path` `C:/Lots/Incoming`. Keep the other parameters at their defaults.
Automation Controller Instance	Connected to a **Resource**, such as `Sorter-01`, and to an **Automation Manager** running in console mode. See How to Create a Controller Instance.

Table 1: Controller used in the tutorial

Create the folder `C:/Lots/Incoming` on the machine where the **Automation Manager** runs. Then, outside that folder, create two text files:

File	Content, one value per line	Size
`lot-a.txt`	`LOT-01`, `LOT-02`, `END`, `LOT-99`	Under 50 bytes
`lot-b.txt`	`LOT-01` to `LOT-08`, then `END`	Over 50 bytes

Table 2: Test files

Copy the test files into the watched folder at each checkpoint, and remove them from it afterward until Step 4 moves them for you. With `ignoreInitial` `false`, the default, every file left in the folder is reported again each time the connection starts. With `awaitWriteFinish` `true`, also the default, each event arrives about two seconds after the copy.

To run the exercise as an automated test instead, the Test Orchestrator File-Raw simulator ↗ can own the watched folder and create the files in place of the equipment.

> **Info**
>
> For the parameters, events, and commands of the driver, see Connect IoT - Raw File Driver and Connect IoT - Raw File and CSV Tutorial.

Sequence of Steps

Step 1: Read the Event Values Through Expressions

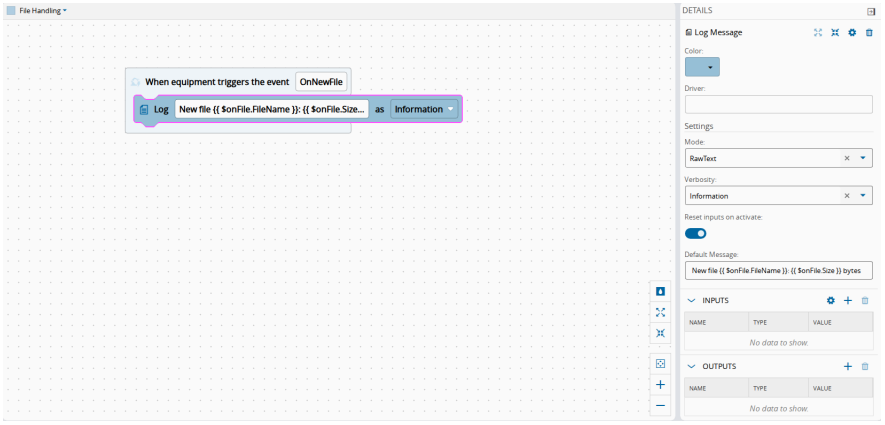
On the **Automation Controller**, add a workflow page, select **Control Flow**, and name the page `File Handling`. Add the following blocks, the second one inside the branch of the first. In the settings of each block, set **Name** to the value of the table: the expressions of this tutorial read each block by that name.

Task	Name	Settings to set	Input rows
`On Equipment Event`	`onFile`	**On Equipment Event** `OnNewFile`	-
`Log Message`	`announce`	**Verbosity** `Information`; **Default Message** `New file {{ $onFile.FileName }}: {{ $onFile.Size }} bytes`	-

Table 3: Blocks of Step 1

1. Save the **Automation Controller**. The **Automation Manager** receives the change.
2. Copy `lot-a.txt` into `C:/Lots/Incoming`.

The console shows `New file lot-a.txt: <size> bytes`. The size depends on the line endings of your editor.



3. Copy `lot-b.txt`. Its line appears with its own name and size: the second occurrence ran in a new branch.

The field after `Log` on the block is **Default Message**. `Log Message` evaluates it when `announce` runs, after `onFile` emitted its values. The expression names the block by its name, not by its block ID: a name that matches no block fails with `Undefined symbol $onFile`, and the branch ends on every file. The text around the expressions is joined without any operator. In other tasks, write expressions in input rows: only the few settings listed in Control Flow Expressions, such as **Default Message**, evaluate an expression.

Step 2: Route the File With a Condition

Add a `Condition` block after `announce`, in the `onFile` branch, and a `Log Message` block in each of its branches:

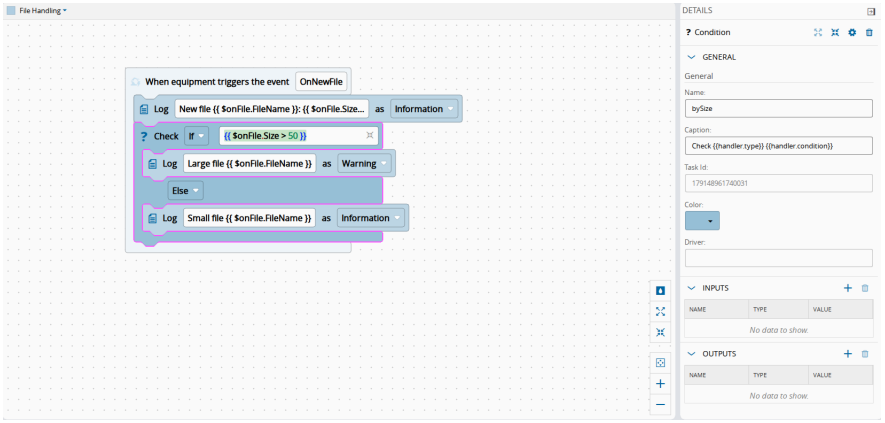
Task	Name	Settings to set	Input rows
`Condition`	`bySize`	`If` condition: `{{ $onFile.Size > 50 }}`	-
`Log Message`, in `If`	`large`	**Verbosity** `Warning`; **Default Message** `Large file {{ $onFile.FileName }}`	-
`Log Message`, in `Else`	`small`	**Verbosity** `Information`; **Default Message** `Small file {{`	-

Task	Name	`$onFile.FileName }}`	Input rows

Table 4: Blocks of Step 2

1. Save, then copy `lot-a.txt` and `lot-b.txt`.

   The console shows `Small file lot-a.txt` and the warning `Large file lot-b.txt`.



2. Change the condition to `{{ $onFile.Size }}`, save, and copy `lot-b.txt` again. The file now takes the `Else` branch.
3. Restore `{{ $onFile.Size > 50 }}` and save.

The `Condition` evaluates its branch conditions itself, when it runs, which is why a setting holds an expression here. A result that is not a Boolean is true only when it reads `true`, `t`, `yes`, `y`, or `1`, so a bare size of `60` is false: compare values instead of testing them bare. Combine tests with `and`, `or`, and `not (...)`, never with `&&`, `||`, or `!`, which fail to parse. The blocks after the `Condition` run whichever branch ran. To route a value in Data Flow, use Switch, which Control Flow does not offer.

Step 3: Go Through the Lines With Foreach and Break

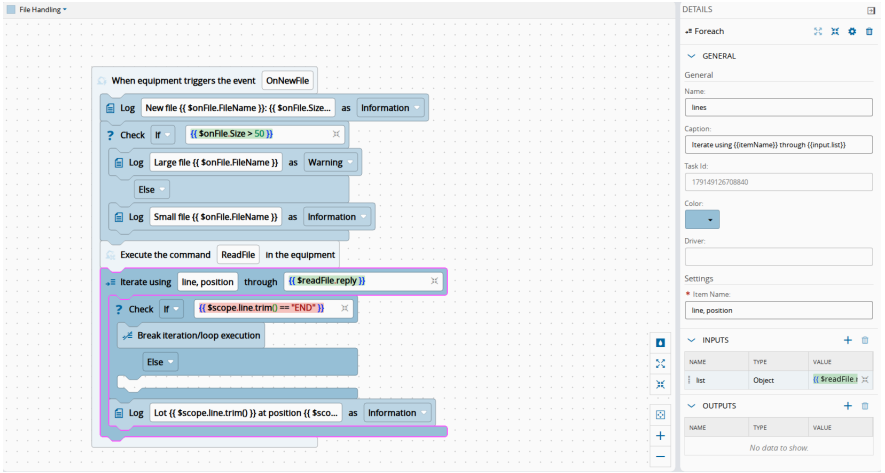
Add the following blocks after `bySize`, in the `onFile` branch. Nest `endMark` and `lot` in the branch of `lines`, and `stop` in the `If` branch of `endMark`.

Task	Name	Settings to set	Input rows
`Execute Equipment Command`	`readFile`	**Equipment Command** `ReadFile`	`path`: `{{ $onFile.Full Path }}`

Task	Name	Settings to set	Input rows
`Foreach`	`lines`	**Item Name** `line, position`	`list: {{ $readFile.reply }}`
`Condition`, in `lines`	`endMark`	`If` condition: `{{ $scope.line.trim() == "END" }}`	-
`Break`, in `If` of `endMark`	`stop`	-	-
`Log Message`, in `lines`	`lot`	**Verbosity** `Information`; **Default Message** `Lot {{ $scope.line.trim() }} at position {{ $scope.position }}`	-

Table 5: Blocks of Step 3

1. In the **Command Results** of `readFile`, change the **Value** of the `reply` row to `{{ $this.reply.split("\n") }}`. The command replies with the text of the file, and the `reply` row has the type `Object`, which you cannot change. An `Object` row parses text as JSON, which fails with `Failed to process outputs from task '<task ID>'`, while the array of lines passes as it is.



2. Save, then copy `lot-a.txt` and `lot-b.txt`.

> The console shows `Lot LOT-01 at position 0` and `Lot LOT-02 at position 1` for `lot-a.txt`, then the warning `Requested for the execution to break an iteration/loop`. `LOT-99` is never logged. For `lot-b.txt`, the eight lots appear before the break. The lines of the two files can interleave, because each file runs in its own branch.

`Item Name` with two names publishes each element under the first and its index, from `0`, under the second, both read through `$scope`. Write the line separator as the escape `"\n"`: a line break entered in a row reaches the evaluator as a space. `trim()` removes the carriage return that Windows editors leave at the end of each line. `Break` ends the innermost loop, the loop reports success, and the blocks after it run. After the loop, `$scope.line` no longer exists. A block can read only what already ran: placed after `lines`, `readFile` would fail the branch with `Undefined symbol $readFile`.

Step 4: Archive the File With a Retry

Moving a file can fail, for example when the destination already holds a file with that name. `Move` has an `attempts` parameter of its own, but a step without one, or a sequence of steps, needs the recommended retry pattern: a `For` loop whose branch holds a `Try..Catch`. The step and a `Break` go in `try`, so that the loop ends after the first success. A pause goes in `catch`, so that the next attempt waits. Add the following blocks after `lines`, in the `onFile` branch:

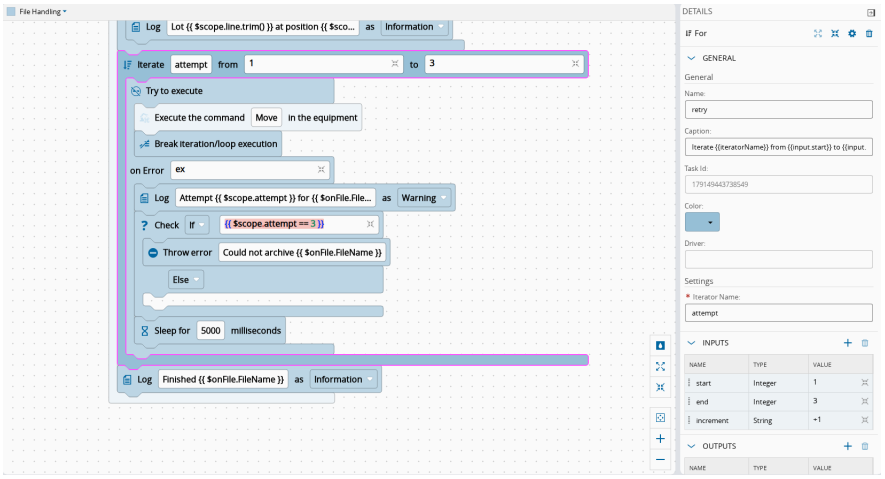
Task	Name	Settings to set	Input rows
`For`	`retry`	**Iterator Name** `attempt`	`start`: `1`; `end`: `3`; `increment`: `+1`
`Try..Catch`, in `retry`	`guard`	-	-
`Execute Equipment Command`, in `try`	`archive`	**Equipment Command** `Move`	`source`: `{{ $onFile.FullPath }}`; `destination`: `${path}/done/{{ $onFile.FileName }}`
`Break`, in `try` after `archive`	`done`	-	-

Task	Name	Settings to set	Input rows
`Log Message`, in `catch`	`failed`	**Verbosity** `Warning`; **Default Message** `Attempt {{ $scope.attempt }} for {{ $onFile.FileName }} failed: {{ $scope.ex }}`	-
`Condition`, in `catch`	`lastTry`	`If` condition: `{{ $scope.attempt == 3 }}`	-
`Throw`, in `If` of `lastTry`	`gaveUp`	**Error Text** `Could not archive {{ $onFile.FileName }}`	-
`Sleep`, in `catch` after `lastTry`	`pause`	**Interval** `5000`	-
`Log Message`, after `retry`	`finished`	**Verbosity** `Information`; **Default Message** `Finished {{ $onFile.FileName }}`	-

Table 6: Blocks of Step 4

1. Save, then copy `lot-a.txt`.

   The file moves to `C:/Lots/Incoming/done`, which `Move` creates when it is missing, and the console shows `Finished lot-a.txt`. The `done` folder is not watched, because `depth` is `0`.



2. Copy `lot-a.txt` again. `done` already holds a file with that name, so each move fails.

   The console shows three warnings, `Attempt 1`, `Attempt 2`, and `Attempt 3`, five seconds apart, each with the reason. Then it shows `Throwing error 'Could not archive lot-a.txt'` and `Failed to execute the workflow!` with that message. `Finished lot-a.txt` does not appear.

3. Copy `lot-a.txt` again, and delete `C:/Lots/Incoming/done/lot-a.txt` during the pause after the first warning. The second attempt moves the file, and `Finished lot-a.txt` appears.

The `catch` branch receives the error message, a text, under `$scope.ex`, the default **variableName** of the branch. `Break` in `try` is not a failure: the `catch` does not run, and the loop ends. Without `gaveUp`, the loop would end after the third attempt and report success like any loop, and `finished` would run as if the file was archived. A failure inside `catch` is not caught by the same `Try..Catch`, so `gaveUp` ends the loop and the branch. **Error Text** is a setting that `Throw` evaluates when it runs. The driver replaces `${path}` with the watched folder.

> **Note**
>
> In Control Flow, an `Unhandled Error` task does not receive the failures that a `Try..Catch` handles. A failure that escapes the `catch`, such as the one that `gaveUp` raises, reaches it as a failure of `guard`. See Unhandled Error.

Step 5: Move the Classification Into a Function

A function is a named branch that any workflow of the **Automation Controller** can run with `Call Function`, from Control Flow or Data Flow. Add a Control Flow page `Shared Functions` with these blocks:

Task	Name	Settings to set	Input rows
`Define Function`	`classifier`	**Function Name** `classifyFile`; **Arguments** `fileSize` (`Integer`); **Results** `category` (`String`)	-
`Return`, in `classifier`	`answer`	**Data** `{}`	Add `category` (`String`): `{{ $scope.fileSize > 50 ? "large" : "small" }}`

Table 7: Blocks of the Shared Functions page

On the `File Handling` page, add a `Call Function` block between `announce` and `bySize`, and change the condition of `bySize`:

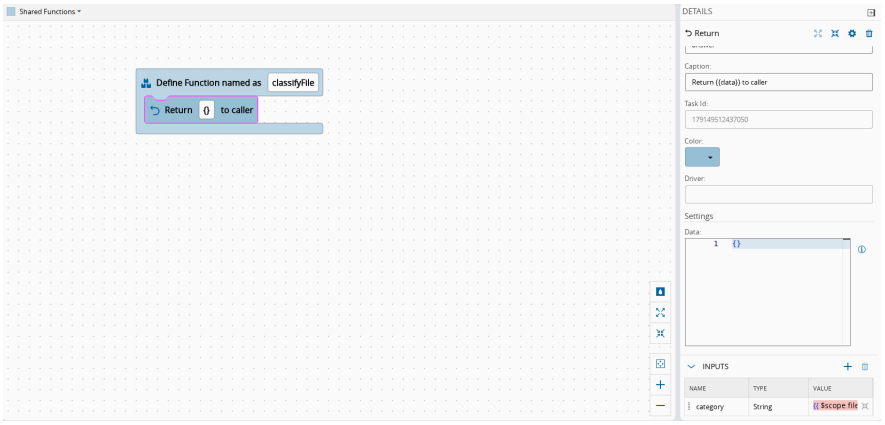
Task	Name	Settings to set	Input rows
`Call Function`	`classify`	**Function** `classifyFile`	`fileSize`: `{{ $onFile.Size }}`
`Condition`	`bySize`	`If` condition: `{{ $classify.category == "large" }}`	-

Table 8: Changes to the File Handling page

Selecting the function fills **Function Arguments** with `fileSize` (`Integer`) and **Function Results** with `category` (`String`), as the `Define Function` declares them.

1. Save, then copy `lot-a.txt` and `lot-b.txt`.

   Before the size lines, the console shows `Calling function 'classifyFile' from workflow 'Shared Functions', task '<block ID>'` for each file. The routing is the same as in Step 2.



`Call Function` looks the function up by its exact name, letter case included, across every workflow of the **Automation Controller**, each time it runs. When no function or more than one function has that name, it fails with `Found '<count>' functions with the name '<name>' within the entire controller`, so give each function a name that is unique in the controller. The argument is converted to the type of the function argument, and the function reads it as `$scope.fileSize`. `Return` sends back **Data** with each input row added under its own name. Keep **Data** a JSON object: when it is not valid JSON, the rows are not returned and the caller logs `Unable to parse the results into a valid json object`. `Return` ends the function only, and the caller continues with `bySize`.

A function suits reusable logic that needs a condition, a loop, or a `Try..Catch`. When the reusable piece must merge values that arrive separately or wait for several of them, build a Data Flow sub-workflow with Workflow Start and Workflow End, and call it with Workflow. From version 12.0.0, a sub-workflow does not start with the controller: it runs only while a `Workflow` task executes it, and the auto-enabled tasks inside it, such as timers, live only for that call. The controller logs `Workflow '<name>' will not be started because it is a subworkflow` when it loads one.

Validate the Result

1. Delete the `done` folder, then copy `lot-a.txt` and `lot-b.txt` into `C:/Lots/Incoming`.
2. Check the console. For each file, in this order, the log shows:

Block	Log line
`announce`	`New file lot-a.txt: <size> bytes`

Block	Log line
`classify`	`Calling function 'classifyFile' ...`
`small` or `large`	`Small file lot-a.txt`
`lot`	One line per lot until `END`, then the break warning
`done`	The break warning, after the move
`finished`	`Finished lot-a.txt`

Table 9: Log lines of one file

3. Check that `C:/Lots/Incoming` holds only the `done` folder, and that `done` holds both files.

The finished `onFile` branch reads top to bottom:

```
onFile (On Equipment Event, OnNewFile)
    announce (Log Message)
    classify (Call Function, classifyFile)
    bySize (Condition): If -> large, Else -> small
    readFile (Execute Equipment Command, ReadFile)
    lines (Foreach): endMark (Condition): If -> stop (Break); lot (Log
Message)
    retry (For, 1 to 3)
        guard (Try..Catch)
            try: archive (Move), done (Break)
            catch: failed (Log Message), lastTry (Condition): If -> gaveUp
(Throw); pause (Sleep)
    finished (Log Message)
```

What You Learned

- Each event occurrence runs in a branch of its own, and a block reads earlier blocks by name through `{{ $<block name>.<output> }}`.
- Expressions go in input rows, which are evaluated when the block runs, and only a few settings, such as a condition, **Error Text**, or **Default Message**, evaluate their own.
- A condition compares values, a row type decides how a value is converted, and `Break` ends only the innermost loop.

- A `For` with a `Try..Catch` retries a step, and running out of attempts is not a failure unless you throw one.
- A function is found by its unique name each time it is called, and `Return` ends only the function.

Next Steps

Nothing in this workflow outlives a branch. To keep values between occurrences, wait, and react on a schedule, continue to Connect IoT - Timing and State Tutorial. To turn the content of a file into data with code, see Connect IoT - Writing a Code Task Tutorial. For the order of all the Connect IoT tutorials, see Connect IoT - Learning Path.

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.