Critical manufacturing
an ASM PT company

Timing and State

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 - Timing and State Tutorial

Overview

Between one equipment event and the next, a workflow decides when to act, how long to wait, and what to remember. Clocks start runs on a period or on a schedule, `Sleep` and `Timeout` act inside a run, and the controller data store keeps the values that must outlive the run that produced them.

This tutorial is for integrators who completed Connect IoT - How Values Travel Tutorial. The File Raw driver plays a handler that drops one file per part. You add a file counter with a scheduled report, a watchdog that raises an alarm when files stop arriving, a time limit on slow work, a join that keeps the values of each file together, and a queue per part that remembers what the arrival of a part carried until the part leaves. At the end, the **Automation Manager** console shows each of them at work.

Info

For more information, see On timer, On Cron Job, Timeout, and Retrieve Data From Queue.

Choose the timing task by what the requirement says about time:

The requirement says	Task	What it does
"every 10 seconds"	`On Timer`	Starts a run every **Interval** milliseconds
"every day at 06:00"	`On Cron Job`	Starts a run at the moments of a cron expression, in UTC
"wait 3 seconds, then"	`Sleep`	Pauses the current run, which then continues
"if nothing arrives within 30 seconds"	`Timeout`	In Data Flow, a watchdog that emits `Error`. In Control Flow, a time limit on a

The requirement says	Task	What it does
branch		

Table: Timing tasks by requirement

Before You Start

- Use the File Raw integration and the **Automation Manager** console from Connect IoT - How Values Travel Tutorial, with an empty watched folder, for example `c:/temp/Handler`. Turn off **Auto Enable** on the triggers of the workflows of that tutorial, so that their lines do not mix with the new ones.
- Check that the **Automation Driver Definition** has the shipped events `OnNewFile` and `OnFileDeleted`. See Connect IoT - Raw File Driver.
- Raise the events by hand, or from an automated test with the Test Orchestrator File-Raw simulator ↗.
- In Control Flow expressions, this tutorial writes each block by a readable name, such as `onFile`. In your workflow, type `$` inside `{{ }}` and pick the block names that the designer suggests.

Sequence of Steps

Step 1: Count Files in the Data Store

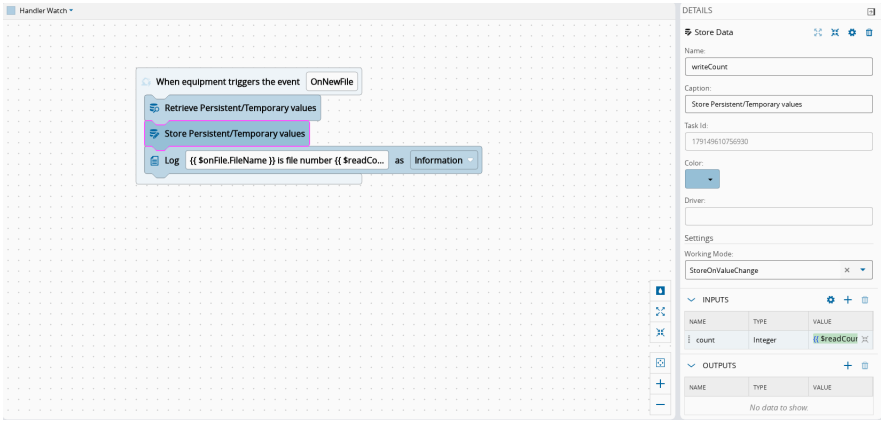
A value needed later in the same run travels with the run. A value that another run, another handler, or another workflow needs goes into the controller data store: `Store Data` writes it under an **Identifier**, and `Retrieve Data` reads it back by the same **Identifier**.

1. Add a Control Flow workflow named, for example, `Handler Watch`, with an `On Equipment Event` trigger `onFile` whose **Equipment Event** is `OnNewFile`.
2. Place these tasks in the handler of `onFile`, in this order:

Task	Settings	Input rows
`Retrieve Data` (`readCount`)	**Outputs** tab: entry `count`, **Identifier** `handlerWatch.fileCount`, **Type** `Integer`, **Default Value** `0`	-
`Store Data` (`writeCount`)	**Inputs** tab: entry `count`, **Identifier** `handlerWatch.fileCount`, **Type**	`count`: `{{ $readCount.count`

Task	Settings	Input rows
	`Integer`, **Storage** `Persistent`	`+ 1 }}`
`Log Message`	**Mode** `RawText`; **Default Message** `{{ $onFile.FileName }} is file number {{ $readCount.count + 1 }}`	-

Table: Tasks that count the files



3. Save, and drop `a.csv` and then `b.csv` into the watched folder.

   The console shows `a.csv is file number 1` and `b.csv is file number 2`.

The **Identifier** is shared by every workflow of the controller, and the last write wins. Name each identifier after the workflow that owns it, as `handlerWatch.fileCount` does. A `Temporary` value is lost when the controller restarts. A `Persistent` value survives the restart and also outlives the workflow that wrote it, so a new workflow that reuses the identifier starts from the old value.

Reading and writing are two separate steps. Control Flow gives each file its own run with its own values, while a Data Flow counter fed by a burst of files pairs names with the wrong numbers. In Control Flow, give every `Store Data` entry an input row: an entry without one writes no value, and the data store deletes its identifier.

Step 2: Start Runs on the Clock

`On Timer` starts a run every **Interval** milliseconds, and `On Cron Job` starts one at each moment of a cron expression. Each tick or firing is a run of its own, and in Control Flow each runs the handler of the trigger.

1. In `Handler Watch`, add an `On Timer` trigger `heartbeat` with **Interval** `10000` and **Number of Occurrences** `3`. In its handler, place a `Log Message` with **Default Message** `Handler watch alive at {{ $heartbeat.timestamp }}`.
2. Add an `On Cron Job` trigger `report` with **Cron Expression** `*/30 * * * * *`, and place these tasks in its handler:

Task	Settings	Input rows
`Retrieve Data` (`readTotal`)	Entry `count`, as in Step 1	-
`Log Message`	**Mode** `RawText`; **Default Message** `Files since the last report: {{ $readTotal.count }}`	-
`Store Data`	Entry `count`, as in Step 1	`count`: `0`

Table: Tasks of the scheduled report

3. Save, drop two files, and wait one minute.

   The heartbeat shows three lines, ten seconds apart, and `Timer deactivated after '3' occurrences` appears just before the third line. On each whole and half minute, the report shows the number of files since the last report and sets it back to `0`.

Number of Occurrences `0`, the default, sets no limit, and the count starts again each time the clock is enabled. A tick is never early and often a few milliseconds late, so read the time from `timestamp` instead of counting ticks.

The cron expression starts with an optional seconds field, and it runs in UTC: `0 0 6 * * *` fires at 06:00 UTC, not at 06:00 on the clock of the plant. Write the expression in UTC, and change it when the plant changes to or from summer time. An expression that is not valid fails when the job starts, with `Cron expression '<expression>' is not valid`. With **Auto Enable** on, the error goes to a linked `Error` or to an `Unhandled Error` task, and without either, the workflow does not start. A start refused from `Enable/Disable`, in Data Flow, leaves the job off, and the next `true` checks the expression again.

In Control Flow, a clock runs from the start of the workflow. To start, stop, or change a clock from the workflow, build it in Data Flow and link its `Enable/Disable` and its inputs. A value linked to **Interval** reaches the timer as it arrives, so send a number of milliseconds, not a text.

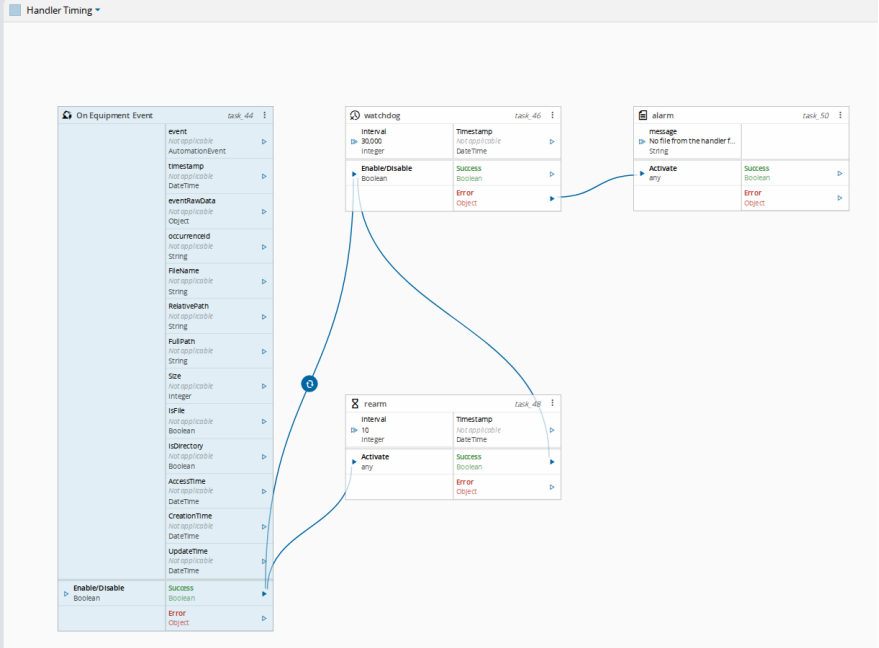
Step 3: Raise an Alarm When Files Stop Arriving

In Data Flow, `Timeout` is a watchdog: `true` on `Enable/Disable` arms it, `false` disarms it, and when **Interval** passes while it is armed, it emits `Error`. Arming a watchdog that is already armed is ignored, so each file must disarm it and then arm it again, in two separate deliveries.

1. Add a Data Flow workflow named, for example, `Handler Timing`, with an `On Equipment Event` task whose **Equipment Event** is `OnNewFile`.
2. Add these tasks and links:

Task	Settings	Links
`Timeout` (`watchdog`)	**Interval** `30000`	From `Success` of `On Equipment Event` to `Enable/Disable`, with the `Any to Constant` converter on the link, `type` `Boolean` and `value` `false`
`Sleep` (`rearm`)	**Interval** `10`	From `Success` of `On Equipment Event` to `Activate`. From `Success` of `rearm` to `Enable/Disable` of `watchdog`
`Log Message` (`alarm`)	**Mode** `RawText`, **Verbosity** `Warning`, **Default Message** `No file from the handler for 30 seconds`	From `Error` of `watchdog` to `Activate`

Table: Tasks and links of the watchdog



3. Save, drop a file every few seconds for half a minute, and then stop.

   While files arrive, each one shows `Timeout deactivated` and `Timeout activated for the next '30000 ms'`, and no alarm appears. The first file shows `No timeout is currently active. Ignoring cancel request` instead of the deactivation. Thirty seconds after the last file, the console shows `Timeout occurred` and then the alarm line.

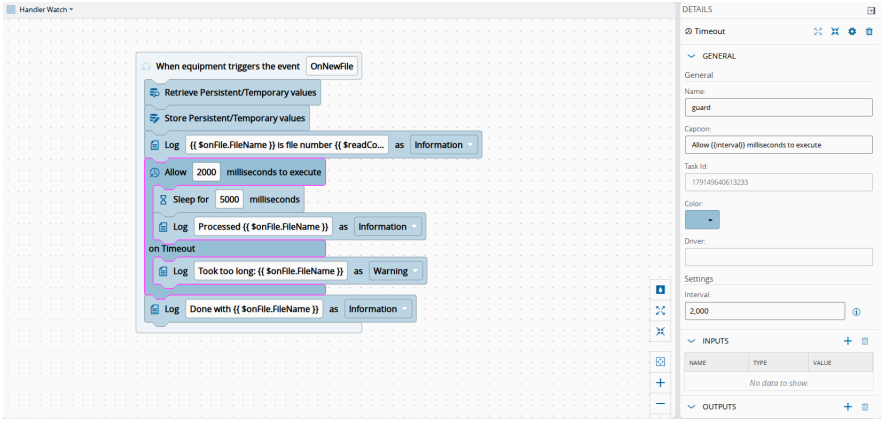
A watchdog armed by every file, with no `false` first, keeps its first deadline, shows `Timeout is already activated. Ignoring new request.`, and raises the alarm 30 seconds after the first file while files are still arriving. Two links from the same event to `Enable/Disable`, one `false` and one `true`, do not help either: both values reach the input in the same moment, the second replaces the first, and only `true` arrives. The 10-millisecond `Sleep` makes `true` a later delivery.

Step 4: Put a Time Limit on Slow Work

In Control Flow, `Timeout` runs the branch inside it under a time limit. When the limit passes, the engine stops that branch and runs the **on Timeout** branch. Either way, the flow then continues after the `Timeout`.

1. In the handler of `onFile` in `Handler Watch`, after the tasks of Step 1, add a `Timeout` (`guard`) with **Interval** `2000`.

2. Inside `guard`, place a `Sleep` with **Interval** `5000`, standing in for slow work, and then a `Log Message` with **Default Message** `Processed {{ $onFile.FileName }}`.
3. In the **on Timeout** branch of `guard`, place a `Log Message` with **Verbosity** `Warning` and **Default Message** `Took too long: {{ $onFile.FileName }}`.
4. After `guard`, place a `Log Message` with **Default Message** `Done with {{ $onFile.FileName }}`.



5. Save, and drop `c.csv`.

   The console shows `Starting execution with a timeout of '2000 ms'`, an error-level line with `execution timeout after '2000' ms. Stopping context tree`, the warning `Execution timeout after '2000' ms. Executing the onTimeout handler`, `Took too long: c.csv`, `Successfully executed the tasks within the timeout protected area`, and `Done with c.csv`. `Processed c.csv` never appears.

6. Set the **Interval** of the `Sleep` to `500`, save, and drop `d.csv`.

   The console shows `Processed d.csv` and `Done with d.csv`.

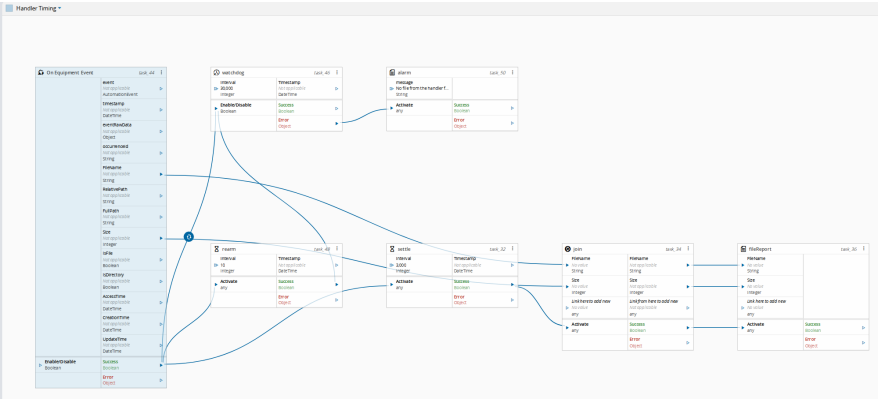
The success line is the same in both cases and ends with the limit, `after '2000 ms'`, not with the time taken. Only the warning before it tells a timeout apart, and the engine logs the stopped branch at error level although the flow continues. A Control Flow `Timeout` never emits `Error`, so decide in the **on Timeout** branch what a late result means: report it there, or end the branch with `Throw` when the rest of the flow must not run. To limit a single `Execute Equipment Command`, use its own **Timeout** setting instead.

Step 5: Keep the Values of Each File Together

In Data Flow, `Synchronize` holds the values it receives for each run and emits them when a value reaches its `Activate`. A value that travels past a `Sleep` through

`Synchronize` stays with its own file, where a value linked straight past the `Sleep` is replaced by the next file, as in Connect IoT - How Values Travel Tutorial.

1. In `Handler Timing`, add a `Synchronize` (`join`) with **Emit when all defined** on, and a `Sleep` (`settle`) with **Interval** `3000`.
2. Link the `FileName` and `Size` outputs of `On Equipment Event` to the `Link here to add new` input of `join`, which adds the matching outputs. Link `Success` of `On Equipment Event` to `Activate` of `settle`, and `Success` of `settle` to `Activate` of `join`.
3. Add a `Log Message` (`fileReport`) with **Mode** `MultipleInputs`. Link the `FileName` and `Size` outputs of `join` to its `Link here to add new` input, link `Success` of `join` to its `Activate`, and save.



4. Drop `e1.csv`, and drop a larger `e2.csv` within three seconds.

   Three seconds after each file, `fileReport` shows the `FileName` and `Size` of that file.

`Sleep` continues the run that activated it, so its `Success` reaches `join` in the run of its own file. Activate `Synchronize` from the task that produces its last value, never from the trigger: an activation that arrives before the values is not kept, and the held values are dropped after **Expiration Time**, `60000` milliseconds by default, without a new value. Any value on `Activate`, `false` included, releases the held values. `Synchronize` exists only in Data Flow, because each Control Flow run already keeps its own values.

Step 6: Remember Each Part Until It Leaves

A queue in the data store holds values that you take out one at a time, and taking a value removes it. A queue named after each part carries what the first event of the part learned to its last event, and leaves nothing behind after the part is gone. In this

step, a new file is a part that arrives, and the deletion of the file is the part that leaves.

1. Add a Control Flow workflow named, for example, `Handler Parts`, with an `On Equipment Event` trigger `onArrive` on `OnNewFile` and an `On Equipment Event` trigger `onLeave` on `OnFileDeleted`.
2. Place these tasks in the handler of `onArrive`, in this order:

Task	Settings	Input rows
`Code` (`deadline`)	Output `until` of type `DateTime`. On the **Code** tab, `main` returns `{ until: new Date(Date.now() + 3600000) }`	-
`Store Data In Queue With Expiration`	**Storage** `Persistent`	`queueName`: `handler.part.{{ $onArrive.FileName }}`. `value`: `{{ {"size": $onArrive.Size} }}`. `queueExpiration`: `{{ $deadline.untilOut }}`

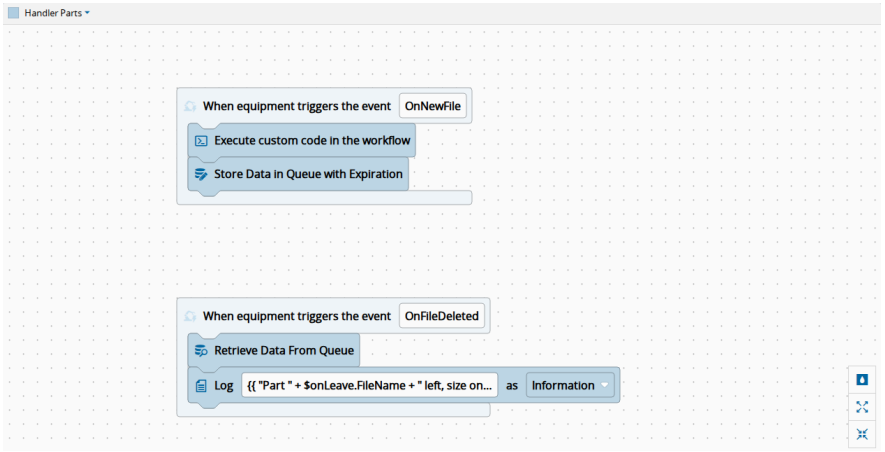
Table: Tasks that store a part

3. Place these tasks in the handler of `onLeave`, in this order:

Task	Settings	Input rows
`Retrieve Data From Queue` (`take`)	**Queue Type** `FIFO`	`queueName`: `handler.part.{{ $onLeave.FileName }}`. `defaultValue`: `{{ {"size": "(not recorded)"} }}`
`Log Message`	**Mode** `RawText`; **Default Message** `{{ "Part " + $onLeave.FileName + " left, size on arrival " + string($take.value.size) }}`	-

Table: Tasks that take a part

The `queueName` and `defaultValue` rows are not listed by default. Add each with **+** in **Inputs**: the settings of the same name, **Default Queue Name** and **Default Value**, do not evaluate an expression.



4. Save, drop `P-001.csv` with some text in it, and then delete it.

   The console shows `Part P-001.csv left, size on arrival <bytes>`, a size that only the arrival event carried.

Take the value at the start of the handler of the last event, on every path, before any condition. A queue that a condition skips stays in the data store for every part that the condition did not select, and with `Persistent` storage it grows without end. Store the values of a part as one object, and give the take a default object with the same fields: a text default read by field fails the branch.

A queue expiration is a fixed moment, set when a value is stored, and later stores and takes do not move it. After it, the next task that reads or writes the queue deletes the queue with everything it holds and emits an error. Set it well past the longest time a part stays in the equipment, so that the queue of a part whose last event never comes lapses by itself. For values that go stale one by one, give each value an `itemExpiration` instead. **Queue Type** decides, for each take, which value comes out next:

Queue Type	Next value
`FIFO`	The oldest value
`LIFO`	The newest value

Queue Type	Next value
`FEFO`	The value whose item expiration comes first. Values without one come after, oldest first

Table: Queue types

With any queue type, a value whose item expiration has passed is skipped. An empty queue gives the default value with `Success`, not an error, so test for the default when a timer drains a queue. Do not give a queue the name of a `Store Data` identifier: the queue tasks fail on a name that holds a plain value.

When one current value per identifier is enough, `Store Data With Expiration` stores it with a deadline, and `Retrieve Data` reads it until then. The deadline must be a date, not a text. The first read after the deadline deletes the value and emits on `Error` an error whose message is `Error while retrieving data: undefined`, and the next read gets the default value. A `Store Data` write over that identifier keeps the old deadline.

Validate the Result

Drop three files two seconds apart, wait one minute, and then delete one of the files. The console shows:

Workflow	Lines
`Handler Watch`	Each file counted in order, `Processed` and `Done with` for each file, and a report on each whole and half minute with the files since the previous one
`Handler Timing`	The name and size of each file three seconds after it, and one alarm 30 seconds after the last file
`Handler Parts`	One `Part ... left` line for the deleted file, with the size it had on arrival

Table: Expected console lines

What You Learned

- `On Timer` and `On Cron Job` start runs, the cron expression runs in UTC, and `Sleep` and `Timeout` act inside a run.
- A Data Flow watchdog is disarmed and armed again by each occurrence, in two separate deliveries, and a Control Flow `Timeout` decides in its **on Timeout** branch what late means.
- `Synchronize` keeps the values of each run together until the last producer activates it.
- The data store shares identifiers and queue names across the controller, `Persistent` values outlive restarts and workflows, and a queue per part is taken on every path with an expiration as a safety net.

Next Steps

The deadlines in this tutorial came from a small `Code` task. Continue with Connect IoT - Writing a Code Task Tutorial to decide when a `Code` task is the right tool and to write, log, and test one. 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.