Critical manufacturing
an ASM PT company

Connect IoT - TCP/IP 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.

12.0

Copy Page

Connect IoT - TCP/IP Tutorial

Overview

The TCP/IP driver serves simple devices that exchange text or fixed-size frames over a socket, such as barcode readers, scales, label printers, and test rigs with a line-based protocol. It works at the transport level only:

- A **parser** cuts the incoming byte stream into frames.
- Every frame is tested against every property of the driver definition, and an event fires when one of its trigger properties takes the frame.
- A command sends a frame built from its device ID, its parameters, and the framing tokens you configure. It can wait for one answer, the next frame that arrives, and the driver does nothing more to match an answer to its question.

A device with sessions, message identifiers, or acknowledgments of its own needs its own driver, or the SECS/GEM driver for a GEM tool.

The connection has two modes, which describe the role of the driver, not the role of the device:

`mode`	The driver	Use it when	What to know
`Active`	Connects to `host:port`	The device listens on a port	The socket closing moves the connection to `Disconnected`, and the driver reconnects
`Passive`	Listens on `host:port` for up to `maxClients` devices	The device connects to a host on its own, as most readers configured with a host address do	A device leaving does not change the communication state, and every command goes to every connected device

In this tutorial, you integrate a packing station, such as `Station-01`, that connects to the controller and sends one line per scanned case, ended by CR LF. For each case, the controller asks the station for the weight with `W`, logs the answer, such as `S`

`12.50 kg`, and acknowledges the case with `OK`. The tutorial is for integrators who completed the basic configuration tutorial and build Data Flow workflows.

Info

For more information, see Connect IoT - TCPIP Driver.

Before You Start

- Complete Connect IoT - Basic Configuration Tutorial: you need an **Automation Manager** that you can start in console mode, the **Resource** entity type enabled for Connect IoT, an integration user, and a **Resource**, such as `Station-01`.
- Make the package `@criticalmanufacturing/connect-iot-driver-tcpip` available in the Connect IoT repository.
- Choose a free port on the machine that runs the **Automation Manager**, for example `7420`.
- Prepare a stand-in for the station: a TCP terminal that connects to that port and sends and receives lines ended by CR LF. This tutorial uses Ncat, the command-line networking tool of the Nmap project. On Windows, install Nmap ↗, which includes Ncat. On Linux, install Ncat with the package manager of your distribution. The command `ncat --crlf <manager-host> 7420` connects to the port on the host of the **Automation Manager**, and `--crlf` ends each line that you send with CR LF.

Alternative TCP Terminals

Any other TCP terminal that sends CR LF, such as PuTTY in `Raw` mode, works as well.

Automated Tests

To automate the checks for this tutorial, write a Test Orchestrator test that uses the TCP/IP Client simulator, which dials a `Passive` driver, or the TCP/IP Server simulator ↗, which listens for an `Active` driver.

Sequence of Steps

Step 1: Create the Automation Protocol

Go to **Business Data** > **Automation Protocol** and create a protocol named `TCP/IP` with the package `@criticalmanufacturing/connect-iot-driver-tcpip`, as described in How to: Create an Automation Protocol. Keep the default parameter values; you set the values below for the connection in Step 3.

Parameter	Value in this tutorial	Why it matters
`mode`	`Passive`	The station connects to the driver
`host`	`0.0.0.0`	In `Passive` mode, the address to listen on. In `Active` mode, the address of the device
`port`	`7420`	In `Passive` mode, `0` lets the operating system choose a port, which the driver log shows
`maxClients`	`1`	Keep `1` whenever the workflow sends commands, because a command goes to every connected device
`socketEncoding`	`ascii`	How the bytes on the socket become text before the parser reads them
`parserType`	`Readline`	How the stream is cut into frames
`delimiter`	`\r\n`	The line end of the station, removed from each frame

Choose the parser from the way the device ends a message. A wrong parser never completes a frame, so no event fires, or it splits one message into several:

Device	`parserType`	Parameters it reads
Ends each message with a line end or another fixed sequence	`Readline` or `Delimiter`	`delimiter`
Sends frames of a fixed size	`ByteLength`	`byteLengthNumber`
Ends a message with a pattern	`Regex`	`delimiter`, read as a regular expression, and `encoding`

Device	`parserType`	Parameters it reads
Stops sending at the end of a message	`InterByteTimeout`	`timeoutValue`, `maxBufferSize`

For `Readline`, `Delimiter`, and `Ready`, only the first `\r`, `\n`, and `\t` of `delimiter` are converted into control characters, so `\r\n\r\n` is not a valid delimiter, and a token such as `<ETX>` is matched as the five characters it contains. For a control character other than a line end, the `Regex` parser reads `delimiter` as a regular expression, where `\x03` names ETX, provided `socketEncoding` is a text encoding, such as `ascii`. Test such a configuration against the device before you rely on it.

Step 2: Define the Messages of the Station

The TCP/IP package ships no driver definition, so you declare every property, event, and command from the message format of the device. The `identifierType` of a property decides which frames it takes: `None` takes every frame, `Token` takes frames that start with `parserIdentifierValue`, and `RegularExpression` takes frames that match it.

1. Go to **Business Data** > **Automation Driver Definition** and create a definition named `Packing Station` on the `TCP/IP` protocol with the entity type `Resource`.
2. Add the property `CaseCode` of type `String`, readable and not writable, with the **Identifier Type** (`identifierType`) `Token` and the **Identifier Value** (`parserIdentifierValue`) `CASE-`.
3. Add the event `CaseScanned` with the **Event Working Mode** (`workingMode`) `AllValues`, the default, and add `CaseCode` to it with **Is Trigger** on.
4. Add the commands, with no parameters:

Name	Command (Command Device ID)	Extended data
`GetWeight`	`W`	**Command Structure Behavior** `UseCurrentCommandSettings`, **Message End Token** `<CR><LF>`, **Convert Non Printable ASCII Chars on Send Value** on, **Reply Expected** on, **Reply TimeOut (ms)** `5000`
`Acknowledge`	`OK`	**Command Structure Behavior** `UseCurrentCommandSettings`, **Message End**

Name	**Command (Command Device ID)**	**Token** `<CR><LF>`, **Convert Non Printable Extended data ASCII Chars on Send Value** on

5. Select **Create**.

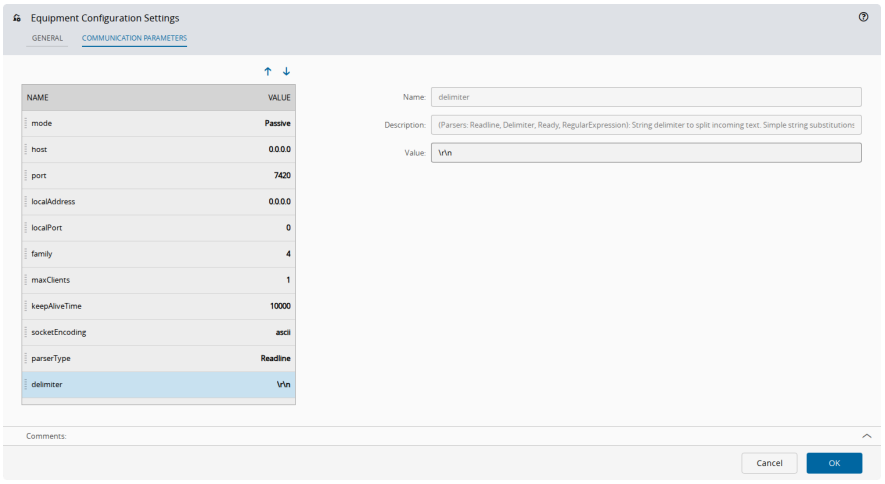
With the default **Command Device ID Usage**, `AtBeginning`, each command sends its device ID followed by the end token. The driver adds nothing at the end of a frame unless you configure it, and the conversion turns `<CR><LF>` into the two control characters. Without the conversion, the station receives the six characters `<CR><LF>`, and a `\r\n` typed in **Message End Token** is sent as four characters: only the parser `delimiter` understands escapes.

With **Reply Expected** (`expectsReply`), the command waits for the next frame that arrives, whatever its content, and returns it on `reply`. That frame is not offered to the properties, so it fires no event. While the command waits, other commands are queued. When no frame arrives within **Reply TimeOut (ms)**, the command fails.

> **Note**
>
> A `Token` property takes only the frames of its kind. Two `None` properties on one connection both take every frame, and both their events fire for each frame. Writing a property, for example with `Set Equipment Properties`, sends nothing to the device: only a command writes to the socket.

Step 3: Create the Controller and Set the Connection

1. Create an **Automation Controller** named `Packing Station` with the **Scope** `Connect IoT`, the entity type `Resource`, and the **Workflow Type** `Data Flow`.
2. In the driver definitions step, add `Station` with the `Packing Station` definition, and keep the core task package.
3. On the `Setup` page, open the settings of the `Equipment Configuration` task, set the parameters from the table in Step 1 on the **Communication Parameters** tab, and select **Save**.



Info

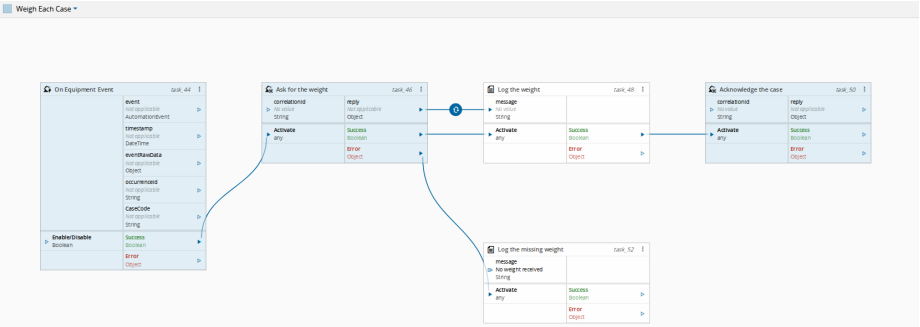
For more information, see Create the Setup Workflow.

Step 4: Build the Workflow

Add a workflow page named `Weigh Each Case`, place the tasks below with the driver `Station` for the equipment tasks, and link them as listed:

Task	Settings to set	Inputs and outputs to link
`On Equipment Event`	**On Equipment Event**: `CaseScanned`	-
`Execute Equipment Command`, named `Ask for the weight`	**Equipment Command**: `GetWeight`; keep **Timeout** at `60000`, longer than the **Reply TimeOut (ms)**	`Success` of the event to `Activate`
`Log Message`, named `Log the weight`	Default values	`reply` of `Ask for the weight` to `message`, through the converter `Any To String`; `Success` of `Ask for the weight` to `Activate`

Task	Settings to set	Inputs and outputs to link
`Execute Equipment Command`, named `Acknowledge the case`	**Equipment Command**: `Acknowledge`	`Success` of `Log the weight` to `Activate`
`Log Message`, named `Log the missing weight`	**Default Message** `No weight received`, **Verbosity** `Warning`	`Error` of `Ask for the weight` to `Activate`



Select **Save**. The **Timeout** of the task is the limit of the controller; keeping it longer than **Reply TimeOut (ms)** makes the driver error the one reported. `Success` of a command means the call returned: for `Acknowledge`, that the frame was written.

Step 5: Connect the Instance and Start the Manager

1. Connect the controller to the **Resource** `Station-01` and to your **Automation Manager**, and select the same **Resource** for the `Station` driver, as described in How to Create a Controller Instance.
2. Start the **Automation Manager** in console mode.

The driver listens on port `7420`. Until the first device connects, the driver instance does not reach `Communicating`, and frames that arrive before `Communicating` are dropped.

Validate the Result

1. Connect the stand-in station:

```
ncat --crlf <manager-host> 7420
```

The console logs `Client '<id>' connected to server.`, and in **Automation** > **Connect IoT** the driver instance moves through `Setup` to `Communicating`.

2. Send the line `CASE-7001`. The terminal shows `W`, the weight request of the controller.
3. Send the line `S 12.50 kg` within five seconds. The console logs `S 12.50 kg`, and the terminal shows `OK`.
4. Send the line `CASE-7002`, wait more than five seconds, and then send `S 12.50 kg`. The command fails with a message that it timed out while waiting for a reply from the device, `Log the missing weight` logs `No weight received`, and the late answer fires no event, because it does not start with `CASE-`.
5. Close the terminal. The console logs `Client '<id>' disconnected from server.`, and the driver instance stays `Communicating`.

Troubleshooting

Symptom	Cause	Fix
No event ever fires, or two messages arrive as one	The parser does not match how the device ends a message, or `delimiter` holds a token such as `<ETX>`	Set `parserType` and `delimiter` from the device format; use `Regex` for other control characters
The device receives the command but does not react	The frame has no line end, or `\r\n` was typed in **Message End Token**	Write `<CR><LF>` and turn the conversion on, on the command or for the connection
A `reply` holds a message that does not answer the command	The device sent another frame first, or several devices share a `Passive` connection	Use **Reply Expected** only for a device that answers at once and sends nothing on its own; otherwise declare the answer as its own `Token` property and event
A device stopped sending and nothing reports it	In `Passive` mode, a device disconnecting does not change the communication state	Do not rely on the state of a `Passive` connection to detect a loss; use `Active` when the device listens

Symptom	Cause	Fix
A command reaches every reader	In `Passive` mode, a command is written to every connected socket	Keep `maxClients` at `1`, or give each device its own connection on its own port

Next Steps

For a device that listens instead of connecting, set `mode` to `Active`, `host` to the address of the device, and `port` to its port; the rest of the tutorial is the same. To send every command with the same framing, set `commandMessageEnd` and `defaultConvertNonPrintableASCIICharsOnSend` as communication parameters, and keep **Command Structure Behavior** at `UseEquipmentSetupTaskSettings`. To report the weight to MES, continue with Connect IoT - Talking to MES Tutorial, and for devices that speak HTTP, see Connect IoT - REST Client and Server Tutorial.

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.