A Standardized Approach for Firmware Updates over EtherNet/IP

Matt Frazer
Manager, Vendor Adoption Tools
ODVA

Joakim Wiberg
Director of Technology
ODVA

Presented at the ODVA
2026 Industry Conference & 24th Annual Meeting
October 21, 2026
Vilamoura, Portugal

Abstract

In Industrial Automation, keeping device firmware up to date is critical for maintaining security, reliability, and compliance with evolving regulations. With the introduction of the Cyber Resilience Act (CRA), firmware updates for connected devices will soon become mandatory, creating an urgent need for standardized methods to manage these updates efficiently across diverse equipment. Currently, firmware updates often require vendor-specific tools, which can complicate maintenance and increase operational risk in multi-vendor environments.

This paper presents a proposal for a common, harmonized, and open approach for performing firmware updates over EtherNet/IP. By specifying CIP Object(s) and services as a standardized, vendor-independent interface for firmware updates and exploring a common packaging format for firmware, the approach aims to simplify deployment, reduce errors, and streamline device management. Key use cases and implementation strategies are outlined, highlighting how the solution can accommodate the needs of both simple and complex devices.

Adopting such a standardized approach provides significant benefits to end users, enabling a single tool to manage firmware updates across an entire facility regardless of the number of vendors involved. This reduces the operational burden of managing multiple update processes, enhances consistency and security, and aligns with regulatory compliance requirements.

By providing a vendor-neutral framework for firmware updates, ODVA members and the broader EtherNet/IP community can adopt a more efficient, reliable, and scalable approach to maintaining device firmware, supporting both operational efficiency and regulatory compliance.

 ©2026 ODVA, Inc.

Keywords

EtherNet/IP, CIP, firmware update, firmware management, industrial automation, OPC, Open Packaging Convention, vendor-neutral, multi-vendor, Cyber Resilience Act, CRA, NIS2, IEC 62443, cybersecurity, industrial IoT, ODVA, firmware packaging, firmware container, PKI, digital signature

Introduction

Firmware updates are a cornerstone of maintaining secure, reliable, and compliant industrial automation systems. As connected devices proliferate across manufacturing facilities, the ability to efficiently manage firmware across an entire installed base becomes increasingly critical.

Today, firmware updates are largely managed through vendor-specific tools, creating an operational burden in multi-vendor environments. The European Union's Cyber Resilience Act (CRA) [1] further raises the stakes by mandating that connected devices receive timely security updates throughout their lifecycle.

This paper proposes a harmonized, vendor-neutral framework for performing firmware updates over EtherNet/IP, defining CIP Object(s) and services as a standardized interface, and exploring a common firmware packaging format to support consistent deployment across diverse devices and vendors.

Importance of firmware updates in industrial automation

Industrial automation devices — including programmable controllers, drives, sensors, and network infrastructure — rely on firmware to define their core behavior and capabilities. Keeping this firmware current is essential for several reasons.

From a security perspective, firmware updates deliver patches for known vulnerabilities, reducing the attack surface of connected devices and helping protect critical infrastructure from cyber threats. As industrial systems become increasingly networked and exposed to IT environments, the importance of timely security updates continues to grow.

Beyond security, firmware updates deliver bug fixes, performance improvements, and new features that help ensure devices operate as intended over their operational lifetime. Outdated firmware can lead to unexpected behavior, compatibility issues, and increased maintenance costs.

Finally, as regulations such as the EU Cyber Resilience Act come into force, maintaining up-to-date firmware will shift from best practice to a legal requirement for many device categories, making structured firmware management an operational necessity.

Current challenges with vendor-specific tools in multi-vendor environments

Despite the importance of keeping firmware current, the process of actually performing updates in industrial environments remains fragmented. Most device vendors provide their own proprietary tools, file formats, and update procedures, meaning that a facility with equipment from multiple vendors must maintain a corresponding set of disparate update workflows.

This fragmentation creates several practical challenges. Maintenance personnel must be trained on multiple tools and processes, increasing the risk of errors and inconsistencies. Scheduling and tracking updates across a mixed fleet of devices becomes difficult, making it hard to maintain a clear picture of the firmware status across the facility. In some cases, updates may be deferred simply due to the complexity of the process, leaving devices running outdated and potentially vulnerable firmware.

 ©2026 ODVA, Inc.

The fragmentation runs deeper than tooling alone. Vendors rely on different protocols and transport mechanisms for firmware updates — EtherNet/IP being one of several, with many vendors using entirely proprietary protocols or interfaces. It is not uncommon for a device to require a direct physical connection, such as USB, to perform an update. This rules out remote management entirely, requiring maintenance personnel to physically walk out to each device on the factory floor, connect a laptop, and run the update manually. In large facilities with hundreds or thousands of devices, this is a significant drain on time and resources.

Furthermore, vendor-specific approaches create a barrier to automation. Without a common interface, integrating firmware update management into broader asset management or lifecycle management systems requires custom integrations for each vendor — adding cost and complexity that is difficult to justify, particularly for smaller operations.

Finding the right firmware is a challenge in itself. Each vendor distributes firmware through their own channels — support portals, product pages, or direct contact with support teams — and there is no common mechanism to check whether a newer version is available for a given device. In a facility with equipment from dozens of vendors, staying on top of this manually is simply not realistic.

The result is that updates are frequently missed, not because maintenance teams are unwilling to act, but because they have no reliable way of knowing there is anything to act on. Devices continue running outdated firmware until a problem surfaces or a scheduled audit catches the gap.

A standardized discovery mechanism would address this directly, allowing devices or management tools to check a known source for the latest available firmware without manual intervention from the end user.

Regulatory drivers

The European Union's Cyber Resilience Act (CRA) introduces mandatory requirements for connected devices that will have direct implications for firmware management in industrial automation. The CRA requires that devices support firmware updates after shipment, feature secure update mechanisms, and that manufacturers provide security updates throughout the product lifecycle to address identified vulnerabilities.

In practice, most device vendors already provide a means to update firmware. The CRA does not fundamentally change this. What it does do is raise the bar on how updates are delivered and managed — secure, traceable, and available for the full support period is no longer a differentiator but a legal requirement.

The real challenge the CRA exposes is not whether updates are possible, but how efficiently they can be managed across a multi-vendor installation. While vendors are individually responsible for providing a secure update mechanism, the burden of actually applying those updates consistently across an entire facility falls on the end user. With each vendor fulfilling their CRA obligations in their own way, end users are left to navigate a fragmented landscape of tools and processes. A standardized approach to firmware updates over EtherNet/IP does not change what vendors must do — it simply makes life significantly easier for the people who have to manage it all.

Benefits and Impact

A standardized, vendor-neutral approach to firmware updates over EtherNet/IP delivers tangible benefits across the entire lifecycle of an industrial installation.

A single tool capable of managing firmware updates across all devices in a facility — regardless of vendor — fundamentally changes how maintenance teams operate. Rather than context-switching between multiple tools and processes, personnel can manage the entire update workflow from one place. This

frees up time, reduces training requirements, and makes it practical to keep firmware current across the full installed base.

Standardization reduces the surface area for mistakes. When every device speaks the same update interface, knowing when a new firmware version is available, where to find it, and how to obtain it becomes a consistent and predictable process rather than a vendor-by-vendor exercise. A common packaging format further reduces errors by embedding the metadata needed to verify that the right firmware is being applied to the right device. Because the update process itself is built on standardized CIP Objects and services, errors that occur during an update are also standardized, making them easier for tools and integrators to interpret consistently across vendors.

Consistent, auditable firmware update processes make it easier to demonstrate compliance with a range of regulatory and standards requirements. Regulations such as NIS2 [2] and standards such as IEC 62443-2-1 [3] and IEC 62443-2-2 [4] place requirements on organizations to maintain and document the security posture of their industrial systems. When updates can be tracked, verified, and logged through a single interface, maintaining an accurate record of the firmware status across the facility becomes straightforward rather than a manual exercise.

A standardized approach scales in a way that vendor-specific solutions cannot. As facilities grow, add new equipment, or onboard new vendors, the same tools and processes apply. Maintenance personnel need to learn one tool, one workflow, and one packaging format — regardless of how many vendors are represented on the factory floor. This reduces training costs, lowers the risk of errors from switching between unfamiliar tools, and makes it practical to maintain a larger installed base with the same headcount. Over time, the cumulative reduction in training overhead, error-related downtime, and maintenance complexity translates directly into lower operational costs.

Use Cases

The following use cases illustrate the range of scenarios that a standardized firmware update framework over EtherNet/IP needs to support. They span from straightforward single-device updates to more complex multi-vendor and multi-device scenarios, reflecting the diversity of real-world industrial installations. Together they inform the requirements for both the CIP Object(s) and services definition and the common firmware packaging format proposed in this paper.

Bundling Multiple Update Artifacts

A common, open container format is a prerequisite for any vendor-neutral firmware update solution. Rather than transferring a single firmware file, the container bundles everything a tool needs to perform and validate an update: a manifest describing the contents, one or more firmware images, release notes, license agreements, digital signatures, and any other supporting artifacts.

The assumption that a device carries only one firmware image does not always hold. As discussed in a later use case, many devices require multiple firmware images — for different processors, subsystems, or functional partitions. The container format must accommodate this from the start.

Solution

Building on an existing, proven open format rather than defining something new reduces the implementation burden for vendors and tool developers alike. A well-chosen format brings established tooling, broad platform support, and a known security track record. This paper proposes OPC-pkg (Open Packaging Convention, ECMA-376) as that format — flexible enough to package any file type, so that nothing about the update payload needs to be constrained by the container itself.

Simple device

The simplest and most common update scenario involves a device with limited capabilities — such as a sensor, actuator, or small network component — that runs a single firmware image, or at most two images covering the main application firmware and a bootloader.

A device with both a main application firmware and a bootloader presents a few different update scenarios. Both components may need to be updated together as part of a single operation, or each may need to be updated independently — for example, when a security patch only affects the bootloader, or when a feature update only touches the application firmware. In cases where both need updating, the order of application matters: typically, the bootloader is updated first, followed by the application firmware, to ensure compatibility and a safe update sequence.

These devices typically do not implement the File Object and lack a full filesystem. Storage is likely a flat flash memory organized as a simple non-volatile storage area.

Solution

To accommodate this, a new CIP Object — the NVS Object — is proposed. Rather than adapting the File Object, which assumes a richer storage model, the NVS Object is designed for simplicity from the ground up to reflect flat FLASH storage, keeping the implementation lightweight and avoiding unnecessary complexity on constrained hardware.

In this use case, the update tool identifies the device and matches the available firmware against it, confirming that the correct image is being applied to the correct device. The firmware is then downloaded directly to the device. Once the transfer is complete, the tool triggers a restart and polls the device until it comes back online, confirming that the new firmware has been successfully applied.

Complex device

Complex devices share many characteristics with simple devices but bring significantly more capability. They typically implement a full filesystem and the File Object, and run multiple firmware images — commonly a bootloader, a main firmware image, one or more application firmware images, and a filesystem image.

Despite having a full filesystem available, complex devices may still prefer to receive firmware updates through a flat FLASH storage area, just as simple devices do. The update framework needs to accommodate both approaches, allowing vendors to specify whether a given firmware image is delivered to the File Object, the NVS Object, or a combination of both.

The update process for complex devices also requires finer control over sequencing. With multiple firmware images to manage, vendors need to be able to specify the order in which images are downloaded and applied. In some cases, a restart may be required between individual images — for example, to activate a new bootloader before applying a dependent application firmware — while in other cases several images can be applied in sequence before a single restart.

From the end user's perspective, this complexity should be entirely transparent. Whether updating a simple sensor or a complex controller with multiple firmware images, the experience should be identical — select the device, apply the update, and confirm the result.

Solution

The framework supports this flexibility through the manifest: sequencing and restart behavior, and the choice of storage target for each image, are defined by the vendor as part of the firmware package rather than left to the update tool to determine. This is what keeps device complexity hidden behind a consistent

and predictable update experience — the sequencing, restart behavior, and storage target are concerns for the vendor to define within the firmware package, not for the maintenance team to navigate.

Multi-vendor facility management

In a facility with equipment from multiple vendors, the process of updating a device from one vendor should be indistinguishable from updating a device from any other. The end user selects a device, applies the available firmware, and confirms the result — regardless of who manufactured the device or how it is implemented internally.

Solution

This is the central promise of a standardized firmware update framework. By defining a common interface and a common packaging format, the differences between vendors become an implementation detail that stays behind the scenes. Maintenance teams no longer need vendor-specific knowledge to perform updates, and a single tool can manage the entire installed base across the facility.

Update multiple devices in parallel

In some scenarios, updating devices one at a time is not practical or desirable. When a group of devices needs to run the same firmware version — for example, after a security patch or a coordinated system upgrade — the ability to update all of them simultaneously reduces downtime and ensures consistency across the installation.

In this use case, the end user selects a set of devices and initiates the update. Different device types will naturally take different amounts of time to receive their respective firmware images, so the tool waits until the last transfer is complete before triggering a simultaneous reboot, ensuring that the new firmware is activated at the same time across the entire group.

Solution

The tool downloads the appropriate firmware to all selected devices in parallel, using the same standardized interface regardless of vendor. Because the interface is common, this process works across devices from different manufacturers without any additional effort from the end user — selecting a mix of vendors is no different from selecting devices from a single vendor. The tool handles each device through the same interface, and the end user sees a single, unified operation.

Sequential/multi-step firmware updates

Not every firmware update can be applied directly from a device's current version to the latest release. A device on version 1 may need to pass through an intermediate version, such as version 3, before it can accept version 5 — for example, because the newer image depends on a data migration, bootloader change, or configuration format that only the intermediate version introduces. A tool that simply fetches and applies the latest available firmware without checking this could leave the device in an invalid or unbootable state.

Solution

The manifest's version constraint fields address this directly: a firmware package can declare the minimum version that must already be present on the device. Where the requested update does not meet that constraint, the tool can determine that an intermediate version is needed first and sequence the updates accordingly, rather than applying an unsupported jump.

Custom and test/debug firmware distribution

Not all firmware is intended for general release. A customer may require a version built specifically for their installation — for example, to validate a fix ahead of an official release, or to run a test or debug build maintained for their site rather than the vendor's mainline release stream. It must be possible to identify and retrieve such custom, test, or debug versions alongside the general release stream, rather than through a separate, vendor-specific mechanism.

Solution

The manifest already identifies target devices using Vendor ID, Device Type, and Product Code. This can be extended with an additional identifier that distinguishes a custom, test, or debug version from the vendor's general release stream, allowing such firmware to be published, discovered, and applied the same way as any other package.

Online repository

Knowing that a firmware update is available is only useful if the update can actually be found and obtained. In a multi-vendor environment, tracking down the latest firmware for each device from each vendor's own distribution channel is time-consuming and error-prone, as discussed above. A central, vendor-neutral online repository addresses this directly.

Solution

Every CIP device can be uniquely identified through four attributes of the Identity Object:

- Vendor ID
- Device Type
- Product Code
- Revision

An update tool can query these attributes from any device on the network and use them to look up matching firmware in a central repository. For the purposes of this paper, it is assumed that such a repository is hosted by ODVA — a reasonable assumption given that all conformant CIP devices are already subject to ODVA testing and certification.

With this repository in place, a tool can retrieve the latest available firmware for a given device, or list all available versions if an older version is needed — for example, to roll back to a previously stable release. The tool can also provide an interface to browse firmware across vendors, giving end users visibility into what is available across their entire installed base.

For installations where internet access is not always available — air-gapped systems or sites with limited connectivity — the tool can download and cache firmware locally in advance, then apply updates on that isolated network without requiring manual intervention to locate and transfer the files.

Recovery/Bootloader Mode

Whether and how a device implements a recovery or bootloader mode remains a vendor implementation choice. Where such a mode exists, it typically runs a stripped-down EtherNet/IP stack — just enough to accept a new firmware image and return to normal operation.

Solution

The minimal implementation required in this mode includes:

- Fundamental EtherNet/IP encapsulation services, such as ListIdentity, RegisterSession, and SendRRData
- Simple unconnected and Class 3 messaging, supporting a single connection at a time
- A minimal set of CIP objects, such as the Identity Object, Message Router Object, and the NVS Object proposed in this paper

The NVS Object is particularly well suited for this context. Its deliberately limited scope means it can be implemented with minimal resources, making it a practical fit for a constrained bootloader environment where a full CIP stack is not available.

Signature, integrity and authenticity

The firmware container must be digitally signed so that any tool consuming it can verify its integrity and authenticity. As firmware directly controls device behavior, a tampered or corrupted package could compromise the safety and security of an entire installation. A signature scheme ensures that the firmware originates from a legitimate source and has not been modified in transit or at rest. This verification needs to work in a public, open way — any conformant tool should be able to validate a container without vendor-specific knowledge or proprietary mechanisms. Container-level signing protects the manifest and packaging metadata that route each firmware image to the correct device, a layer that firmware-level integrity checking alone does not cover. Because this verification happens before any transfer to the device, a tool can reject a tampered or corrupted container outright, without spending time or bandwidth moving firmware to a device that would ultimately refuse it. For the firmware files within the container, integrity verification is also required, but the mechanism must remain generic, since this is closely tied to device capabilities and vendor implementation choices.

Solution

Container-level signing can be backed by a PKI hierarchy managed by ODVA or a globally trusted certificate chain; firmware-level integrity verification is declared per image in the manifest rather than mandated by a single mechanism. Both are detailed further above under the container's format and security specification.

Backup/restore

Before applying a firmware update, a tool must be able to upload the current configuration from a device. Some firmware updates reset or overwrite device configuration, and where this is the case, the tool must inform the user before proceeding. The user can then choose to back up the configuration first and restore it once the update is complete.

Solution

This backup and restore capability is most useful when defined generically rather than something tied to update-specific details. A generic interface allows a tool to collect and store configurations from any device and recover a known-good state if any update needs to be rolled back, without requiring device-specific handling. The firmware update workflow uses this capability as part of the update process itself.

Keeping the interface generic also means it can accommodate different device types and configuration structures without requiring vendor-specific handling. The tool does not need to understand the contents of the configuration — it only needs to be able to retrieve it, store it, and push it back to the device.

Transition path for existing devices/vendors

Lowering the barrier to adoption is critical for any common approach to gain traction across the EtherNet/IP ecosystem. Where possible, vendors should be able to build on their existing firmware update implementations rather than replace them entirely.

Solution

Two aspects of existing proprietary implementations cannot be carried forward. First, existing packaging formats are not compatible with the common approach, even when vendors already use the same open container format. The content and metadata that control update behavior will differ, and a generic tool has no way to interpret vendor-specific conventions. Second, existing transfer protocols cannot be reused, even those built on CIP. The common approach relies on standardized CIP objects and services whose behavior is governed by the metadata in the common container format. There is no mechanism to describe or accommodate an arbitrary proprietary protocol alongside this.

What can be preserved is how firmware is packaged internally and how the update is applied within the device. The common approach does not mandate anything about the internal update procedure. Vendors can embed their existing firmware binaries or packages inside the common container, and once the firmware package has been transferred to the device using the standardized interface, the device can apply the update and perform any authenticity checks using its existing internal mechanisms. This means the externally visible interface is standardized while the implementation inside the device remains under vendor control.

Interaction model and workflow

To illustrate how the proposed solution works in practice, this section presents sequence diagrams for two representative device types. The first covers a simple device using the minimum required functionality to perform a firmware update. The second covers a more complex device that handles multiple firmware images and uses secure connections. Together they show how the same standardized interface accommodates a wide range of device capabilities.

Simple device

The simple device case requires only a single firmware image. The firmware image includes a digital signature, allowing the device to verify integrity and authenticity after transfer. The tool uses only locally stored firmware files.

The sequence of operation is outlined in Figure 1. In steps 1 and 2, the tool queries the Identity Object to retrieve the device identity. In step 3, the tool uses this information to identify compatible firmware images available locally. In step 4, the user selects one of the compatible versions. Steps 5 through 79 cover the transfer of the firmware image, referred to as the firmware candidate until it has been verified, to the device. In step 80, the tool requests verification of the firmware candidate. In step 81, the device verifies its integrity and authenticity and, in step 82, reports the result back to the tool. Once verification succeeds, the tool issues a reset in steps 83 and 84. The device restarts in step 85 and applies the firmware candidate in step 86.

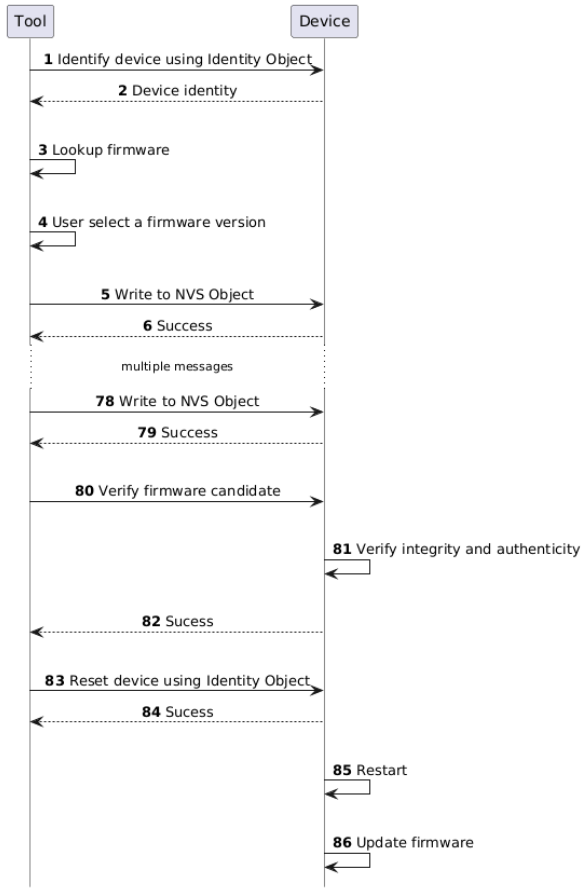


Figure 1 Simple device firmware update

Complex device

The complex device used in this example has two sub-images embedded into one firmware image. The digital signatures for the sub-images are not embedded into the firmware sub-images; they are provided as separate files included in the firmware package. In this example the tool does not have any locally stored firmware images and must retrieve them from an online repository. The complex device also uses the File Object for onboard storage of the firmware images.

The sequence of operation is outlined in Figure 2. In steps 1 and 2, the tool establishes a secure connection to the device. In steps 3 and 4, the tool queries the Identity Object to retrieve the device identity. In steps 5 and 6, the tool uses this information to identify compatible firmware images available at the online repository. In step 7, the user selects one of the compatible versions provided by the online repository. In steps 8 and 9, the tool downloads the selected firmware image from the online repository. In steps 10 and 11, the tool instantiates the File Object. Steps 12 through 79 cover the transfer of sub-image 1 to the device. In steps 80 and 81, the tool instantiates the File Object. In steps 82 and 83, the digital signature for sub-image 1 is transferred to the device. In step 84, the tool requests verification of the firmware candidate for sub-image 1. In step 85, the device verifies its integrity and authenticity and, in step 86, reports the result back to the tool. In steps 87 and 88, the tool instantiates the File Object. Steps 89 through 153 cover the transfer of sub-image 2 to the device. In steps 154 and 155, the tool instantiates the File Object. In steps 156 and 157, the digital signature for sub-image 2 is transferred to the device. In step 158, the tool requests verification of the firmware candidate for sub-image 2. In step 159, the device verifies its integrity and authenticity and, in step 160, reports the result back to the tool. Once both verifications succeed, the tool issues a reset in steps 161 and 162. The device restarts in step 163 and applies sub-image 1 in step 164 and sub-image 2 in step 165.

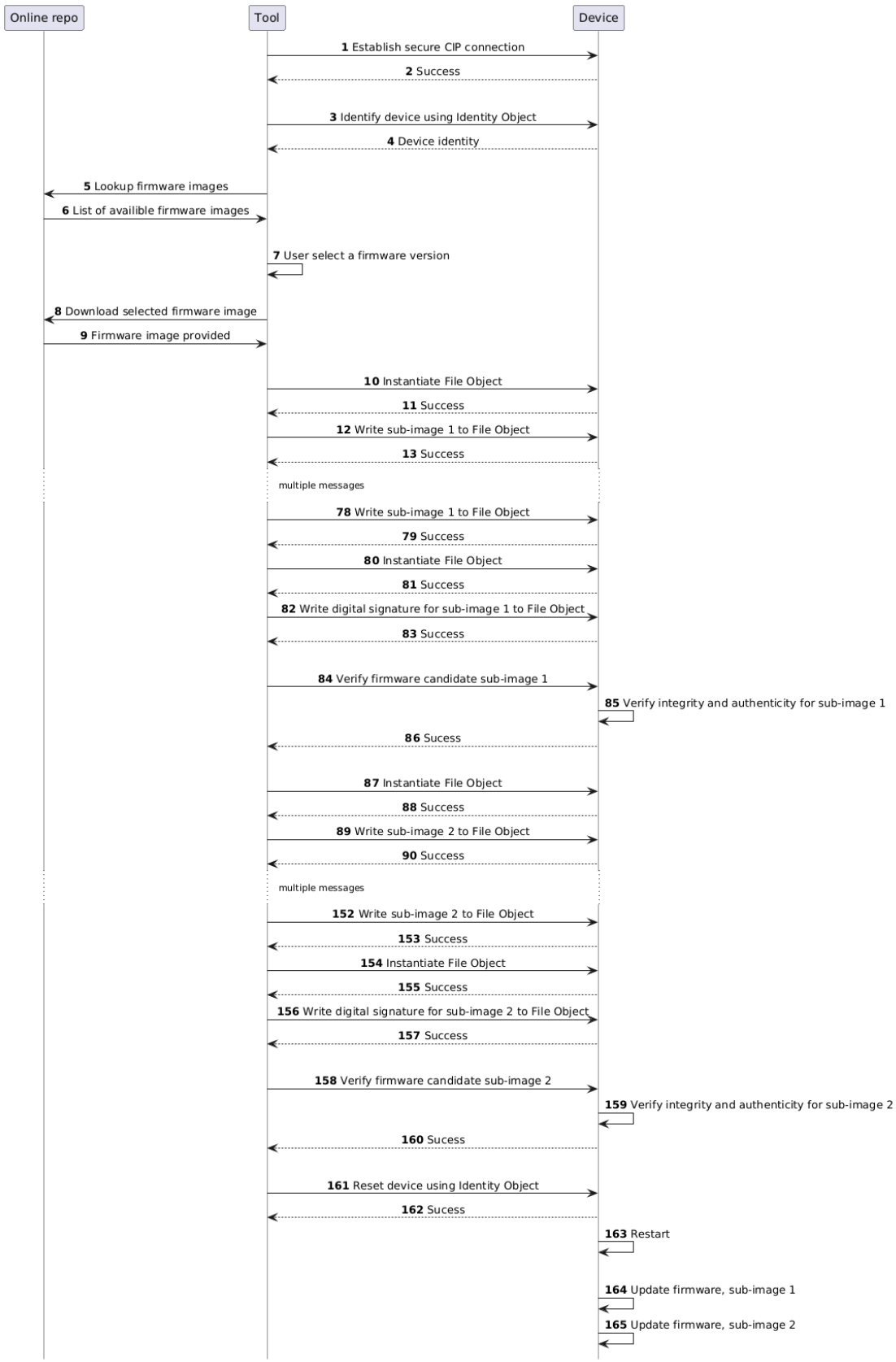


Figure 2 Complex device firmware update

CIP Object(s) and Services for Firmware Updates

A vendor-neutral, open, and harmonized approach is fundamental to any solution that aims to work across the broad and diverse EtherNet/IP ecosystem. For a firmware update mechanism to be genuinely useful, any conformant tool must be able to update any conformant device, regardless of vendor. This requires that the interface, behavior, and packaging are all defined in a way that no single vendor controls or can extend in incompatible ways.

A core design principle for ODVA is to build on existing, proven, and open standards rather than introduce new mechanisms where established solutions already exist. This reduces implementation risk, leverages well-understood security properties, and lowers the barrier for vendors and tool developers to adopt the approach.

Scalability is also a key consideration. The solution must work for simple, resource-constrained devices as well as complex multi-processor devices. A design that only serves one end of this spectrum will not achieve broad adoption across the EtherNet/IP ecosystem.

Security is treated as a first-class design concern, not an afterthought. Integrity, authenticity, and tamper detection are built into the approach from the start, ensuring that the update mechanism itself does not become an attack vector.

Finally, conformance and testability are essential. Consistent with ODVA's approach to standardization, the solution is designed to be verifiable through conformance testing, giving end users confidence that certified devices and tools will interoperate as expected.

The following sections introduce a proposed solution that builds on existing CIP messaging, objects, services, and attributes. Where existing CIP constructs are sufficient, they are used as-is. Where a gap exists, a new CIP Object is proposed e.g., NVS Object, introduced briefly in the use cases above — to provide the functionality needed to support a standardized firmware update interface.

CIP Object(s) definition

Identity Object

The Identity Object is used to uniquely identify a device. The following attributes are used to match a device against the firmware container metadata:

- Vendor ID
- Device Type
- Product Code
- Revision

One option for representing independently updatable subcomponents — for example, the firmware for the local HMI — is to use additional Identity Object instances. However, a more in-depth investigation would be needed if this paper's approach moves forward to formally defining and adding a firmware update feature to the CIP Networks Library. That investigation would need to ensure all cases involving individually updatable subcomponents are covered; it is possible that something beyond Identity Object instances is needed to capture relationships and compatibility between subcomponent versions.

The Identity Object is extended with a new Instance Attribute, Config Instance, that points to either a File Object instance or an NVS Object instance holding a copy of the latest device configuration. This allows a tool to locate and retrieve the configuration before performing an update, when needed.

The Status attribute of the Identity Object provides information about the device's current operational state. One bit is of particular relevance. The Owned bit indicates that the device is currently in use. The

firmware container metadata can instruct the tool to check this bit and block the update if the device is owned, preventing interruption of critical operations.

A new Backup Required bit, proposed within the Status attribute, indicates that the device holds a configuration that should be uploaded and stored before the update proceeds, enabling recovery after the update is complete.

The Status attribute can also indicate that a firmware update is currently in progress, allowing tools to monitor and manage the update process.

The Identity Object also provides the Reset service, which the tool uses to restart the device when needed during the update workflow.

Message Router

The Message Router Object is required when transferring firmware or configuration as a single opaque binary using the Send_Receive_Fragment service. If Send_Receive_Fragment is not used, the Message Router Object does not need to be implemented.

Connection Manager

When Send_Receive_Fragment is used, the Connection Manager Object is also required. This is because Send_Receive_Fragment can only be used over a Class 3 connection, and establishing a Class 3 connection requires the Connection Manager.

CIP Security Objects

Where security is required, devices shall use the Confidentiality Profile defined in CIP Security. This provides authenticity between the tool and the device, and optionally encrypts data in transit. Whether CIP Security is required, and whether confidentiality is enabled, is specified in the firmware container metadata, giving the tool the information it needs to establish the appropriate level of security before proceeding with the update.

File Object

The File Object is well suited for more complex devices and for devices using CIP Security, as the File Object is already required by CIP Security. It is capable and flexible, but this comes at the cost of added complexity and higher resource requirements. Where a device provides a copy of its latest configuration, a dedicated File Object instance within the vendor-specific instance range is used for this purpose, with the Config Instance attribute of the Identity Object pointing to it.

NVS Object

For simpler, more resource-constrained devices, the NVS Object is proposed as a lightweight alternative to the File Object. Its deliberately limited scope makes it practical to implement on devices where the File Object would be too demanding.

The NVS Object can have one or more instances. Each instance represents a contiguous non-volatile storage area accessible through simple read and write services. How this storage area is represented internally in the device is up to each vendor.

Each instance may represent a single firmware image, or alternatively a single instance can be used to handle multiple firmware images sequentially — the tool downloads and applies one image at a time, reusing the same instance for each. This is entirely implementation-dependent and vendors are free to choose whatever approach makes the most sense for their device.

A dedicated NVS Object instance can also be used to hold a copy of the latest device configuration. The Config Instance attribute of the Identity Object points to this instance.

Firmware Packaging Format

Requirements for a common packaging format

A common firmware packaging format must meet several requirements to work across the full range of EtherNet/IP devices and vendors.

The format must be vendor-neutral and based on open, proven technology. Proprietary formats lock tools to specific vendors and raise the barrier for implementors. Building on an established open standard reduces implementation risk and avoids reinventing what already exists.

A real-world firmware package rarely contains a single file. A device may require multiple firmware images for reasons beyond having multiple processors — a single processor device can still need separate images for the main application, bootloader, web pages, or other components. Packages also carry accompanying material — release notes, license agreements, digital signatures, and metadata that governs how the update tool behaves. The format must accommodate all of this without restricting what file types or structures vendors can include.

The format must be flexible enough that vendors can embed their existing firmware binaries unchanged. This is essential for the transition path discussed earlier: the outer container is standardized, but the contents remain under vendor control.

Finally, the format must be self-describing. The metadata it carries must give any conformant tool enough information to identify compatible devices, verify integrity, and execute the update correctly — without vendor-specific knowledge or out-of-band communication.

Proposed format specification

The firmware container is based on the Open Packaging Convention (OPC), defined in ECMA-376 [5]. OPC is an open, well-documented standard with broad tooling support across platforms and programming languages. Using OPC as the foundation means implementors can build on existing, proven libraries rather than writing custom parsing code.

A firmware package is an OPC package with a defined internal structure. The package contains a manifest that describes the contents and controls how a conformant tool processes the package. Alongside the manifest, the package holds the firmware images, digital signatures, and any accompanying files the vendor chooses to include — release notes, license agreements, or supporting documentation.

OPC's relationship model maps naturally to the structure of a firmware package. Relationships between parts allow the manifest to reference firmware images, signatures, and other files in a standardized way, without hardcoding paths or requiring vendor-specific conventions.

The container places no restrictions on the firmware images themselves. Vendors include whatever binary format their device expects. The manifest references each part and provides the metadata needed for the tool to handle it correctly.

Metadata and versioning considerations

The manifest is the control document for the firmware package. It carries the metadata a conformant tool needs to identify compatible devices, present the package contents to the user, and execute the update correctly.

The manifest format is JSON. The schema would be defined using JSON Schema, which allows any conformant tool to validate a manifest file for correctness before processing it. This catches malformed or incomplete packages early, before any transfer to the device begins. Using JSON keeps the manifest human-readable and easy to work with across platforms and programming languages.

At a minimum, the manifest must identify the target device or devices. This includes vendor ID, product type, and product code — the same identifiers the tool reads from the Identity Object. A tool uses this information to confirm that a package is compatible with the device it is about to update before any transfer begins.

The manifest must also carry version information. This covers the firmware version contained in the package as well as any minimum or maximum version constraints — for example, where a firmware update requires a specific bootloader version to already be present on the device. Version information lets the tool warn the user or block an update that would leave the device in an invalid state.

For packages containing multiple firmware images, the manifest describes each one individually. It identifies the role of each image — main application, bootloader, web pages, or other components — and specifies the order in which they must be applied where sequencing matters.

The manifest also controls how the tool transfers each firmware image to the device. For each image, it specifies whether the NVS Object or the File Object is used, and which instance of that object is the target. Where a package contains multiple images, different images may target different object instances — for example, one instance per processor or component. The manifest makes this explicit, so the tool does not need to discover or infer the correct target at runtime.

The manifest specifies the transfer sequence for the firmware images and their associated digital signatures. For each firmware image, the manifest defines whether its digital signature is transferred before or after the image itself. In most cases the signature follows the image, allowing the device to verify integrity immediately after the image has been received.

Where a package contains multiple firmware images, the manifest defines the order in which they are transferred. This is important where dependencies exist between components — for example, where the bootloader must be updated before the main application.

The manifest also controls when a device reset is issued. A reset may be required after each individual firmware image and its signature have been transferred, or it may be deferred until all images and signatures in the package have been transferred. The appropriate behavior depends on the device architecture and is specified per image in the manifest. Where a reset is deferred, the device holds all received images in storage until the final reset triggers the update across all components simultaneously.

Security considerations (signing, integrity verification)

Firmware directly controls device behavior. A tampered or corrupted package could compromise the safety and security of an entire installation. The firmware container must therefore be digitally signed so that any tool consuming it can verify its integrity and authenticity before transferring anything to a device.

OPC defines a standard mechanism for including digital signatures within a package. This mechanism covers the package contents as a whole, allowing any conformant tool to verify that the package has not

been modified since it was signed. Using the OPC signature mechanism means no custom signing infrastructure is needed at the container level — the approach builds on what the format already provides.

This verification must work in a public, open way. Any conformant tool must be able to validate a container without vendor-specific knowledge or proprietary mechanisms. One approach is to use digital signatures backed by a PKI hierarchy managed by ODVA, where ODVA acts as the root and each member organization operates as a subordinate. Alternatively, a globally trusted certificate chain could serve the same purpose.

For the firmware images within the container, integrity verification is also required. Since this is closely tied to device capabilities and vendor implementation choices, the specification does not mandate a specific mechanism. Instead, the manifest declares what integrity verification method each firmware image uses, giving the tool enough information to perform the check without needing vendor-specific knowledge.

Beyond integrity, the signing scheme protects against supply chain attacks. A valid signature confirms that the package originates from a legitimate vendor and has not been modified in transit or at rest. Tools must reject any package where signature validation fails, and must not transfer firmware to a device from an unverified package.

Conclusion

This paper has proposed a harmonized, vendor-neutral framework for performing firmware updates over EtherNet/IP. The approach rests on three pillars: standardized CIP objects and services as the update interface, a common firmware packaging format based on OPC, and a JSON manifest that gives conformant tools everything they need to execute an update correctly without vendor-specific knowledge.

The use cases demonstrate that the approach scales from the simplest single-image device update to complex multi-image scenarios involving multiple processors, online repositories, parallel updates across vendors, and backup and restore. The transition path shows that vendors can adopt the common interface without discarding existing internal update mechanisms — lowering the barrier to implementation and increasing the likelihood of broad adoption across the EtherNet/IP ecosystem.

Security is treated as a first-class concern throughout. OPC digital signatures protect container integrity, the PKI hierarchy ensures authenticity, and per-image integrity verification extends protection to the firmware itself.

Future work includes defining the JSON manifest schema in full, specifying conformance test requirements, and working with ODVA members to validate the approach against a broader range of device architectures. Establishing the PKI hierarchy and the processes around certificate management for member organizations will also require further definition.

If there is sufficient member interest, the next step is to create a project charter and submit it to the MAC and TRB for approval. Following approval, the charter would be handed to the relevant SIGs to drive the actual development work. The project charter would also need to address responsibilities that fall directly on ODVA — specifically the development and ongoing maintenance of the online repository and its associated processes, and whether ODVA should provide a generic firmware update tool for use by the community.

ODVA members and the broader EtherNet/IP community are encouraged to engage with this proposal, provide feedback from implementation experience, and contribute to the standardization effort.

References

[1] Cyber Resilience Act — European Commission, Regulation (EU) 2024/2847

[2] NIS2 — European Commission, Directive (EU) 2022/2555 on measures for a high common level of cybersecurity across the Union
[3] IEC 62443-2-1 — Security for industrial automation and control systems — Part 2-1: Security management system requirements
[4] IEC 62443-2-2 — Security for industrial automation and control systems — Part 2-2: Implementation guidance for an ISCMS
[5] ECMA-376 — Open Packaging Convention, used as the basis for the firmware container format

****

The ideas, opinions, and recommendations expressed herein are intended to describe concepts of the author(s) for the possible use of ODVA technologies and do not reflect the ideas, opinions, and recommendation of ODVA per se. Because ODVA technologies may be applied in many diverse situations and in conjunction with products and systems from multiple vendors, the reader and those responsible for specifying ODVA networks must determine for themselves the suitability and the suitability of ideas, opinions, and recommendations expressed herein for intended use. Copyright ©2026 ODVA, Inc. All rights reserved. For permission to reproduce excerpts of this material, with appropriate attribution to the author(s), please contact ODVA on: TEL +1 734-975-8840 FAX +1 734-922-0027 EMAIL odva@odva.org WEB www.odva.org. CIP, Common Industrial Protocol, CIP Energy, CIP Motion, CIP Safety, CIP Sync, CIP Security, CompoNet, ControlNet, DeviceNet, and EtherNet/IP are trademarks of ODVA, Inc. All other trademarks are property of their respective owners.