Modernizing EZ-EDS for the Future of Device Description Development

Vivek Hajarnavis
Platform Lead
Rockwell Automation

Jim Jarrett
Experience Principal
Rockwell Automation

Joakim Wiberg
Director of Technology
ODVA

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

Abstract

EZ-EDS has long served as a key tool for creating, editing, and validating EDS files within industrial communication ecosystems. As device description technologies evolve, the requirement for additional mandatory requirements present both opportunities and challenges for tools that support EDS development workflows. To remain effective, EZ-EDS must adapt to these changes while continuing to provide a streamlined and reliable user experience for developers.

This paper presents a series of enhancements and extensions to EZ-EDS aimed at improving both functionality and usability. A primary focus is the introduction of support for new requirements, ensuring that the tool remains aligned with current and emerging specification needs. These updates address structural, validation, and workflow considerations essential to next-generation device description practices.

In addition to technical compliance, the paper examines broader usability improvements intended to simplify the development process for EDS authors. These enhancements include interface refinements and workflow optimizations designed to reduce complexity and improve efficiency. Together, these changes position EZ-EDS as a more capable, adaptable, and user-friendly platform for EDS development.

Keywords

Electronic Data Sheet (EDS), EDS Interpreter, EZ-EDS, CIP, EtherNet/IP

 ©2026 ODVA, Inc.

Definition of terms

Connection Originator — A device or tool, such as a network scanner, that initiates a CIP connection to a target device based on the information declared in that device's EDS file.

ConstructedParamN — An Entry Keyword within an EDS file's Parameter Section used to declare a parameter that is built from other parameters, rather than representing a single discrete value; EDS Interpreters are required to support it even though device support remains optional.

EDS (Electronic Data Sheet) — A structured file, organized into Sections, Entries, and Fields, that conveys a CIP device's supported objects, attributes, services, and connection formats to configuration and commissioning tools.

EDS Interpreter — Software that parses an EDS file and acts on its contents — for example tools used to configure a network scanner or a connection originator — enabling it to recognize a device, expose its configurable parameters, and establish the connections the device supports.

EZ-EDS — ODVA's tool for creating, editing, and validating EDS files, and the subject of this paper's proposed enhancements.

ParamN — An Entry Keyword within an EDS file's Parameter Section used to declare a device parameter and its associated attributes.

Introduction

Electronic Data Sheet (EDS) files play a fundamental role in CIP-based device ecosystems, providing the machine-readable device descriptions that enable configuration tools to recognize, configure, and integrate devices on EtherNet/IP networks. As the complexity and capability of industrial devices continue to grow, the tools used to create and validate EDS files must evolve accordingly.

EZ-EDS has long served as the standard tool for creating, editing, and validating EDS files, offering developers a streamlined environment for authoring device descriptions that conform to CIP specifications. Its role in the EDS development workflow is well established across the industrial automation community.

The emergence of EDS improvements introduces new requirements for device description technologies, creating both opportunities and challenges for tools that support EDS development. These requirements span structural, validation, and workflow considerations that are essential to next-generation device description practices. To remain effective and relevant, EZ-EDS must adapt to these evolving specification demands while continuing to provide a reliable and efficient authoring experience.

This paper presents a series of enhancements and extensions to EZ-EDS designed to address these needs. The primary focus is the introduction of support for all EDS requirements, ensuring alignment with current and emerging specifications. Beyond technical compliance, the paper also examines usability improvements intended to simplify the development process for EDS authors, including interface refinements and workflow optimizations. Together, these changes position EZ-EDS as a more capable, adaptable, and user-friendly platform for EDS development.

Background

EDS in the EtherNet/IP Device Integration Workflow

In EtherNet/IP networks, EDS files serve as the primary mechanism through which devices communicate their capabilities and configuration requirements to engineering tools. When a device is commissioned on

a network, the associated EDS file provides the information necessary for configuration tools — such as network scanners and connection originators — to recognize the device, enumerate its configurable parameters, and establish appropriate connections.

An EDS file describes a device in terms of its supported CIP objects, attributes, and services, as well as the connection types and data formats it supports. This information is consumed by EDS Interpreters, which parse the file and present its contents to end users in a meaningful way. The accuracy and completeness of an EDS file is therefore directly tied to the quality of the user experience during device commissioning and integration. Figure 1 illustrates this relationship in practice, showing how a Connection Originator tool presents the Input and Output Assembly structures — and their individual members — as declared in the target device's EDS file.

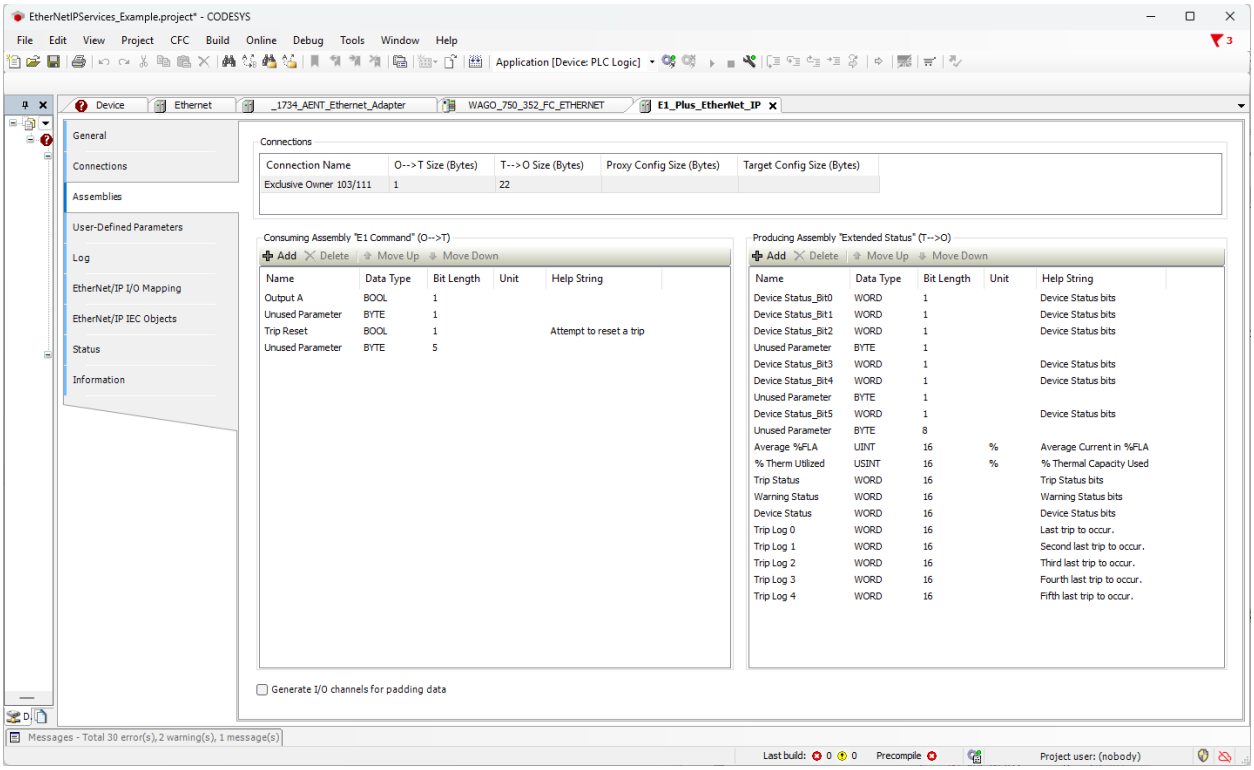


Figure 1 Input and Output Assembly Structures as Presented by a Connection Originator Tool

EDS files are authored by device vendors and must conform to the structural and content requirements defined in the CIP specification. The development of a compliant EDS file requires careful attention to specification requirements, making dedicated authoring and validation tools an essential part of the device development workflow. EZ-EDS addresses this need by providing an integrated environment in which EDS files can be created, edited, and validated against current specification requirements.

EDS Improvements and New Specification Alignment

Historically, adoption of optional CIP functionality was slowed by a circular dependency between control and device vendors: control vendors were reluctant to implement support for a feature until enough devices offered it, while device vendors saw little reason to implement a feature that control-side tools did not yet support. Neither side was willing to move first, and the feature stalled regardless of its merits.

Prior to the EDS improvements initiative, no standardized minimum requirements existed for EDS Interpreters. Each vendor independently determined which keywords to support, leading to inconsistent implementations across the industry. Device vendors faced a difficult choice: either invest significant effort

in researching the capabilities of each EDS Interpreter they needed to target, or restrict their EDS files to only the most basic and widely supported keywords. The latter approach, while pragmatic, limited innovation and often resulted in suboptimal device configurations.

To address this, the EDS improvements initiative established a minimum set of requirements for EDS Interpreters used to configure connection originators. This baseline is designed to cover approximately 90% of products on the market, striking a balance between broad applicability and implementation simplicity. The requirements were developed with the following device characteristics in mind:

- **Non-modular design:** Devices that do not include modular components.
- **No bridging or routing capabilities:** Devices that operate as standalone units without acting as gateways or bridges in a network.
- **Static assemblies:** Devices with fixed assembly configurations that do not vary in size based on runtime or user-defined settings.

These requirements address two primary use cases. The first is parameter access and visualization, where tools must allow users to monitor real-time parameter values and configure them as needed. Parameters should be organized into logical groups, with tools providing clear visualization of those groups to aid the user in understanding the device's structure and capabilities. Accurate data representation is essential in this context — a 16-bit unsigned integer must be presented as such, enumerated values should be displayed using their defined labels rather than raw numeric values, and bit-level parameters should present the names and meanings of individual bits rather than raw binary values.

The second use case concerns support for Class 1 connections. Devices typically include one or more Class 1 connections with defined input, output, and configuration assemblies. Tools and originators must be capable of parsing and interpreting these assemblies, presenting the data in a manner consistent with the intended usage and meaning as defined in the EDS file. Configuration assemblies must also be supported, enabling end users to configure assembly content while respecting the structure and dependencies described in the file.

Edition 3.40 of the CIP Networks Library Volume 1 [1], published in April 2026, put this minimum bar into effect by tightening how support obligations are specified. Four changes underpin the update:

- Specification text was reviewed and clarified throughout, on the premise that ambiguous requirements lead vendors to skip a feature entirely or implement it inconsistently.
- The specification now states explicitly what level of support is expected from devices and from EDS Interpreters separately, rather than leaving the split implicit.
- Where a feature benefits the system as a whole, the specification now requires it in device EDS files but leaves interpreter support optional — recognizing that tool vendors will not prioritize a feature devices rarely expose, and that they are best placed to judge whether their customers need it.
- Where a feature only matters for certain applications or device types, the specification instead makes device support optional while requiring all EDS Interpreters and their connection originators to handle it if present.

These requirements are defined separately for EDS Interpreters and Device EDS files, reflecting the different obligations placed on each. This distinction is illustrated by the handling of the ConstructedParamN Entry Keyword in the Parameter Section. As shown in**Error! Reference source not found.**, EDS Interpreters are required to support the ConstructedParamN Entry Keyword, while it is optional for Device EDS files. This asymmetry is intentional — not all devices will have data that needs to be represented using ConstructedParamN, so device vendors are not obligated to include it. However, since EDS Interpreters must be capable of working with any compliant device, they are required to

support it unconditionally. More generally, if a higher-level section or keyword is not supported by a device, any required subsequent keywords or fields within it are not required to be supported either.

Entry Name	Entry Keyword	EDS Interpreter Requirement	Device EDS Requirement
Parameter	ParamN	Required	Optional
Proxy Parameter	ProxyParamN	Optional	Optional
Proxied Parameter	ProxiedParamN	Optional	Optional
Enumeration	EnumN	Required[1]	Conditional[1]
Proxy Enumeration	ProxyEnumN	Optional[1]	Conditional[1]
Proxied Enumeration	ProxiedEnumN	Optional[1]	Conditional[1]
Fixed	FixedN	Required[1]	Conditional[1]
Constructed Parameter	ConstructedParamN	Required	Optional
Proxy Constructed Parameter	ProxyConstructedParamN	Optional	Optional
Proxied Constructed Parameter	ProxiedConstructedParamN	Optional	Optional
Ranges	RangesN	Required[1]	Conditional[1]
Proxy Ranges	ProxyRangesN	Optional[1]	Conditional[1]
Proxied Ranges	ProxiedRangesN	Optional[1]	Conditional[1]

Table Footnotes:

1. A corresponding ParamN, ProxyParamN, or ProxiedParamN keyword shall exist.

Table 1 Parameter Section Entry Keyword Requirements for EDS Interpreters and Device EDS Files

Where a device does provide a ConstructedParamN Entry Keyword, a separate set of requirements governs the fields within that keyword. As shown in **Error! Reference source not found.**, certain fields within ConstructedParamN are marked as required for Device EDS files. While this obligation only applies when the keyword is actually present, it provides an important guarantee to EDS Interpreters: if a ConstructedParamN Entry Keyword is encountered, the required fields will always be present. An EDS file that includes the keyword but omits any of those fields is considered non-compliant; however, in line with the grace period established alongside the update of CIP Networks Library Volume 1 [1], this requirement does not become mandatory for compliance until April 2027.

Field Name	Field Number	Data Type	EDS Interpreter Requirement	Device EDS Requirement
Link Path	1	Padded EPATH	Required	Conditional [7]
Descriptor	2	WORD	Required	Optional
Data Type Code	3	WORD	Required	Required
Data Size	4	UINT	Required	Required
Name	5	STRING	Required	Required
Units String	6	STRING	Required	Conditional [2]
Member Unit String	7	{STRING, STRING, …} [3]	Required	Conditional [2]
Help String	8	STRING	Required	Required
Member Help String	9	{STRING, STRING, …} [3]	Required	Optional
Minimum Value	10	See Data Type Code field [1]	Required	Optional
Maximum Value	11	See Data Type Code field [1]	Required	Optional
Default Value	12	See Data Type Code field [1]	Required	Required
Type Specific Field Count	13	UINT	Required	Conditional [4]
Type Specific Field	14, 15, …	See Constructed Data Type Definition [5]	Required	Conditional [6]

Table Footnotes:

1. The Data Type of Field Numbers 10, 11 and 12 are based on the value of Field Number 3.
2. At least one of these fields must be specified.
3. The number of STRINGs is determined by the number of members in the Constructed Data Type.
4. The Type Specific Field Count shall be present if the Constructed Data Type defines Type Specific fields otherwise this field is not allowed.
5. The Data Type of Fields 14, 15 and so on is based on the definition of the Constructed Data Type indicated by the Data Type Code in Field 3. See section 7-3.5.2 Defined Constructed Data Type Encodings for type specific definition.
6. Field is required if Field 13 is not equal to zero. The number of Type Specific Fields is dependent on the value of field 13.
7. Required if Not Addressable (Bit 11) of Descriptor (Parameter Object Instance Attribute 4) is zero.

Table 2 Field Requirements within the ConstructedParamN Entry Keyword for EDS Interpreters and Device EDS Files

Interacting with an EDS File

The Developer Use Cases

The are several ways in which a developer might create an EDS file – from creating a file from scratch, copying and modifying an existing file through to the generation of a file from the development environment. If an EDS file is created by hand, the process is likely to be laborious and error prone. For those developers creating files manually, the strength of EZ-EDS as it is today is that it reflects the structure of the file, as well as providing the rules to check for the syntax. However, there is no linkage between EZ-EDS and the requirements of the specification and conformance test: it is entirely the responsibility of the developer to create the parts of the file that they believe will be needed both to meet conformance requirements and to expose the capabilities needed to make use of the device.

For developers that are generating their EDS file from an IDE, the workflow today may look like that shown in Figure 2.

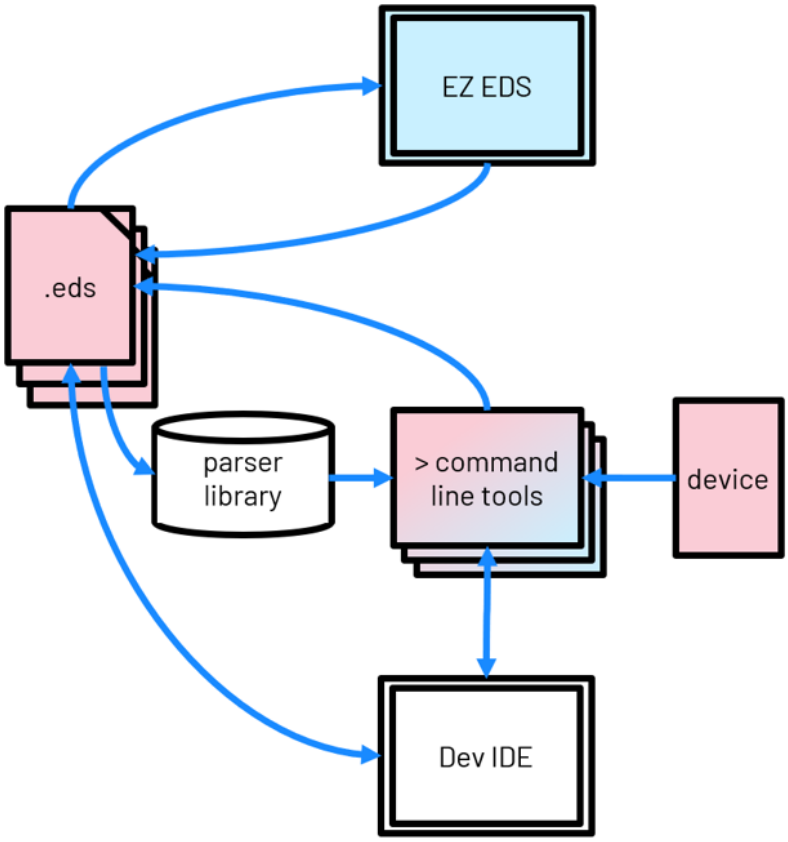


Figure 2 EDS workflow

This shows that there are multiple linkages between the EDS file itself and the tooling. At one level, EZ-EDS gives the ability to both create and review a file. The linkages to the Development IDE are likely to make use of a parser library and command line tools.

Any tooling provided by ODVA for developers needs to reflect the diversity of the developer community – which encompasses licensed vendors who may have limited resources and are working on a small number of devices, through to large members who create multiple, complex devices.

A related and increasingly common case is EDS generation as part of an automated build pipeline. Here, a developer's continuous integration (CI/CD) system generates an EDS file automatically, typically as part of a nightly build, using a command-line interface rather than manual authoring or GUI-based tools. The generated EDS file then needs to be validated on two levels: syntactically, to confirm it conforms to the structural and content requirements of the specification, and functionally, to confirm that what it describes actually matches the capabilities implemented in the device. Supporting this use case requires that both EDS generation and EDS validation be available as command-line operations, so they can run unattended as part of the automated build and test process.

The Device Integrator Use Cases

There are two main workflows related to the use of EDS files today:

1. Using an EDS file to configure a connection in a scanner

2. Reading or writing information from/to a parameter with a link path

The expectation of a user wanting to configure a connection in a scanner is that the programming tool for the scanner will be able to parse the information in the EDS file and to present the options available to the user. This will be primarily to show the list of available class 1 connections that are defined in the file. If the connection information has been defined in line with requirements of the April 2026 specification update then the programming environment also has the potential to read the structure of the data. The requirement for a simple device is that the connection manager section is present, that an assembly is referenced and that the assembly in turn has been structured with parameters.

Furthermore, if individual parameters are defined with a link path, configuration tools can set values (naturally subject to support for the appropriate services). Software tools can also potentially read values, giving the ability for information to be presented to CIP Clients without the need for this information to pass through a connection originator or PLC.

Today, the role of EZ-EDS is limited in terms of helping to meet these use-cases. The tool allows a developer the ability to enter the relevant information but there is no linkage between the appropriate parts of the file. The Connection information, the assembly information and parameters all need to be entered individually. This requires the developer to test the end result outside EZ-EDS. Additional EDS testing outside EZ-EDS is the basis of the EDS test in Plugfest. The Conformance Test Tool already performs some EDS testing today, but this will be expanded to cover and match the EDS requirements that have recently been added to the CIP Networks Library Volume 1 [1].

The Debugger Workflow

Aside from these customer workflows, there may be cases in which someone may wish to use an EDS file to understand more about the CIP implementation – for example, in testing, and if using tools that are not able to parse EDS files and obtain the information directly.

Changes to Validation Rules Driven by the EDS Improvements

The EDS improvements initiative introduces requirements that have direct implications for how EDS files are developed and validated. It is important to note that EZ-EDS is a tool for developing Device EDS files and is therefore primarily concerned with Device EDS file requirements rather than EDS Interpreter requirements. Nevertheless, understanding the distinction between the two requirement categories is relevant context for appreciating the validation changes described in this section.

Both Device EDS file requirements and EDS Interpreter requirements are subject to verification through ODVA Conformance Testing, albeit in different contexts. For EDS Interpreter conformance testing, ODVA verifies that the interpreter can consume EDS files and correctly implements all functionality marked as required for EDS Interpreters. For device conformance testing, the focus shifts to the EDS file itself — the conformance test verifies that the EDS file accurately describes all features implemented in the device that are marked as required for Device EDS files. In addition, the conformance test is expected to flag warnings where a feature is implemented in the device but absent from the EDS file, even where that feature is marked as optional. This serves to make device vendors aware of any discrepancy between the device's actual capabilities and what is represented in the EDS file, giving them the opportunity to address the gap before submission.

Prior to the EDS improvements initiative, the CIP specification did not explicitly define a minimum set of requirements for Device EDS files in the structured and comprehensive manner that it does now. The introduction of clearly designated required keywords and fields in Device EDS files therefore represents a meaningful expansion of the validation landscape that EZ-EDS must address.

To align with these updated requirements, EZ-EDS must extend its validation capabilities to incorporate awareness of the Device EDS file requirements introduced by EDS improvements. Specifically, EZ-EDS should continuously evaluate the EDS file currently open in the tool against the full set of Device EDS file requirements and present the user with a clear, actionable list of required elements that are not yet present in the file. This gives EDS authors an unambiguous view of what remains to be addressed to produce a compliant Device EDS file.

This validation capability is particularly valuable during the early and iterative stages of EDS file development, where the file may be incomplete by design as the author works through the device description progressively. Rather than relying on a final validation pass to surface compliance gaps, authors benefit from continuous feedback that allows issues to be identified and resolved in context. This reduces the risk of discovering significant structural omissions late in the development process, where the cost of remediation is higher.

The approach also aligns with the intent of the EDS requirements themselves. By making the required elements visible and trackable within the authoring environment, EZ-EDS reinforces the baseline that the EDS improvements initiative has established, supporting device vendors in producing EDS files that will meet conformance test expectations and contribute to a more consistent and interoperable ecosystem.

Implications for EZ-EDS

These updated requirements create both opportunities and challenges for EZ-EDS. As a tool that spans both authoring and validation functions, EZ-EDS must address the obligations placed on EDS Interpreters while also providing the means for device developers to produce compliant Device EDS files. The enhancements described in this paper are a direct response to these demands.

Workflow Considerations for Compliant EDS Authoring

This section explores how EZ-EDS can be extended to guide EDS authors through the process of meeting the Device EDS file requirements introduced by the EDS improvements initiative. Rather than placing the burden on the author to independently interpret and apply the specification requirements, EZ-EDS should provide structured workflows and contextual guidance that make compliance a natural outcome of the authoring process.

The following aspects will be addressed in this section:

- The specific workflows and authoring patterns that EZ-EDS should support to help users meet EDS requirements
- Mechanisms for guiding users toward required elements and flagging gaps in a way that is actionable and integrated into the authoring experience
- How EZ-EDS can present requirement context to users at the point of authoring, reducing the need to cross-reference external specification documents

Usability Improvements

EZ-EDS was created in response to a gap in the available tooling for EDS file development. At the time of its creation, a small number of commercial alternatives existed, but none were actively maintained or kept current with the EDS keyword additions that had accumulated over time. A dedicated tool was needed that reflected the full and current state of the specification.

Prior to EZ-EDS, the predominant approach to EDS file development was to use a standard text editor. While this was a practical workaround, it was inherently error-prone. Text editors provide no syntax checking or structural validation for EDS files, meaning that errors could only be discovered by importing the file into an existing engineering tool and observing whether it reported a problem. This workflow introduced several significant inefficiencies. Engineering tools are not dedicated EDS checkers and may report only a single error before aborting, forcing the developer to iterate through a slow and fragmented

correction process. Furthermore, many engineering tools do not support the full range of EDS keywords — they may silently ignore unknown or malformed keywords, or in some cases actively reject them, forcing developers to remove valid content from their files. Perhaps most problematically, the extent of an engineering tool's EDS keyword support is rarely well documented, meaning a developer may receive no errors and incorrectly conclude that the file is fully compliant when in fact unsupported keywords have simply been overlooked.

EZ-EDS addressed these shortcomings by providing an integrated environment for EDS authoring and validation that was current with the specification and capable of checking the full range of supported keywords. This represented a substantial step forward from the text editor workflow, and at the time of its creation the tool was well received for precisely this reason.

However, the usability standards against which software tools are measured have evolved considerably since EZ-EDS was first developed. At the time, user interface design and user experience engineering were disciplines that received little attention in tools developed by engineers for engineers. Functionality was added to ensure coverage rather than to support coherent workflows, and the resulting interface reflects those origins. Error handling is an area where EZ-EDS falls short of current expectations — error messages and validation feedback are not always presented in a way that is clear, actionable, or well integrated into the authoring workflow.

Since its initial release, no systematic effort has been made to revisit the usability of EZ-EDS. In the intervening years, engineers' expectations of the tools they use rose considerably, shaped by broader trends in software design and the proliferation of well-designed development environments across the industry. The enhancements described in this section represent the first structured effort to bring EZ-EDS in line with those expectations, addressing both the interface and the workflow experience it provides to EDS authors.

Interface Refinements

From Multiple Document Interface to a Tabbed Interface

At the time EZ-EDS was originally developed, the Multiple Document Interface (MDI) was a widely adopted application pattern. MDI organizes documents as child windows contained within a single parent application window, a model that was common across many professional tools of the era, including early versions of Microsoft Word and Excel. While MDI offered advantages such as keeping related documents grouped within a single window and making it straightforward to tile or cascade multiple files side by side, it has since fallen out of favor as a design pattern. In 2013, Microsoft transitioned its Office applications from MDI to the Single Document Interface (SDI), where each document operates in its own independent window. This shift reflected broader changes in how users interact with modern operating systems, particularly in multi-monitor environments where the confinement of child windows within a parent window becomes a practical limitation.

Figure 3 shows the current MDI-based interface of EZ-EDS, illustrating the nested child window structure that characterizes this design pattern.

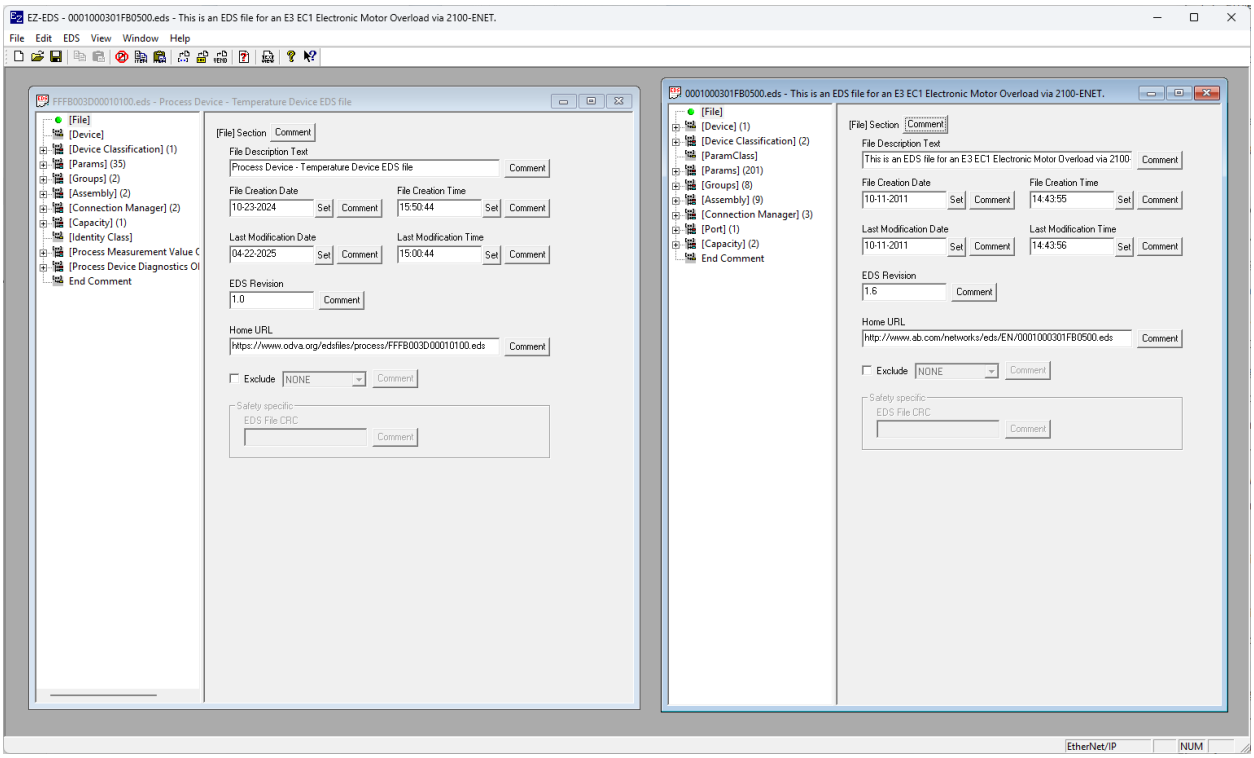


Figure 3 EZ-EDS Current Multiple Document Interface

A tabbed interface represents the natural evolution from MDI for a tool like EZ-EDS. Tabs have become the dominant paradigm in modern productivity software — adopted by web browsers, integrated development environments, terminal emulators, and the current generation of Microsoft Office applications alike. The familiarity of this pattern means that users encounter virtually no learning curve when working with a tabbed interface. Multiple open EDS files are represented as tabs within a single window, eliminating the visual complexity of nested child windows while avoiding the taskbar clutter associated with SDI. Individual tabs can be detached into independent windows when needed, providing the flexibility of SDI on demand and making the interface well suited to multi-monitor workflows. Related documents remain naturally grouped and visible at a glance, and the model scales gracefully whether two or twenty files are open simultaneously.

Figure 4 shows a mock-up of EZ-EDS with a tabbed interface, illustrating how this change would present itself to the user in practice.

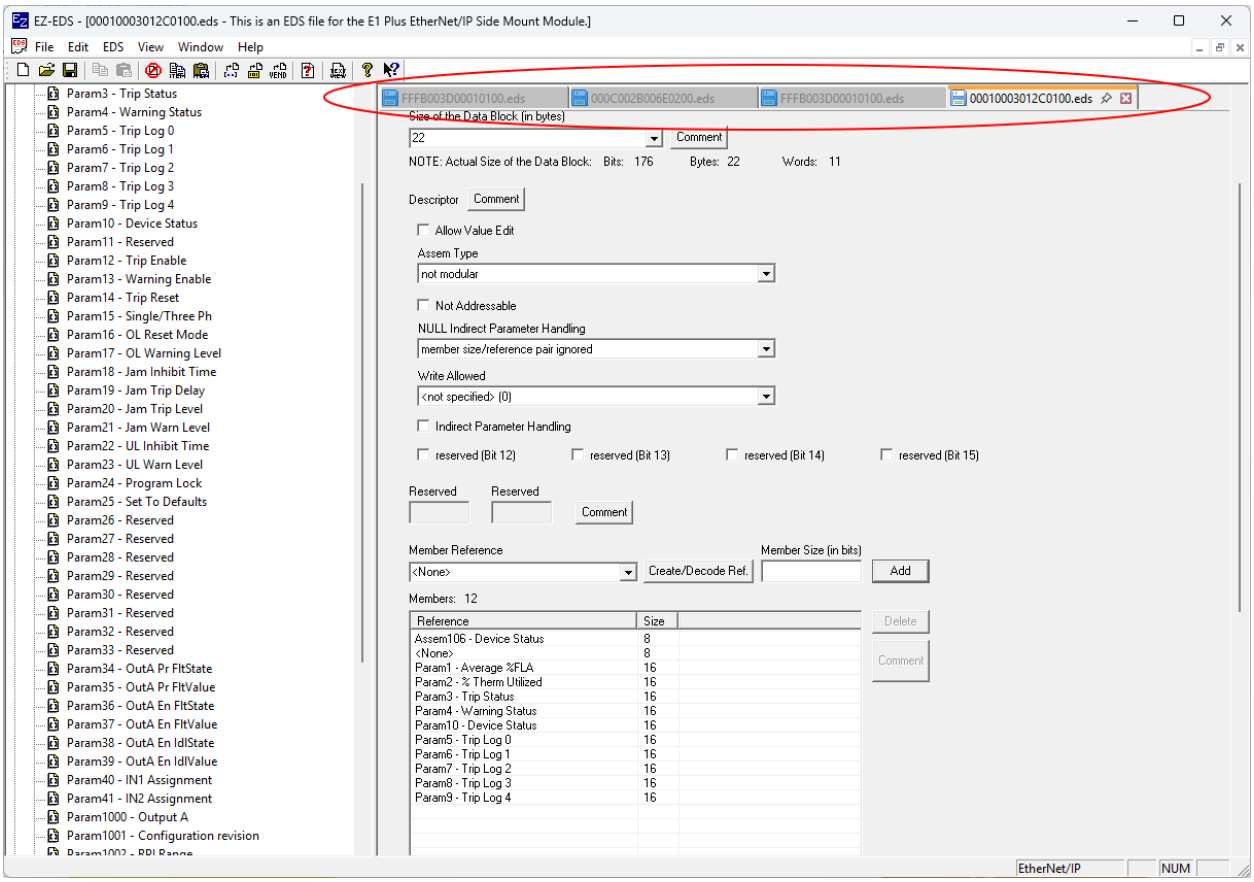


Figure 4 EZ-EDS Proposed Tabbed Interface Mock-up

Moving EZ-EDS to a tabbed interface therefore represents a straightforward but meaningful modernization — one that aligns the tool with the interaction patterns users already know from the software they use every day, while removing a structural characteristic that increasingly marks the application as dated.

From Modal Error Reporting to an Inline Diagnostic Panel

Currently, when any action is performed in EZ-EDS that triggers validation — such as loading or saving a file — any warnings or errors encountered are presented in a modal window. A modal window is a dialog that appears on top of the application and demands the user's attention before any further interaction with the underlying application is possible. While modal windows are appropriate for interactions that require an explicit decision from the user, they are poorly suited to presenting diagnostic information. The fundamental limitation of a modal in this context is that it is by definition blocking — the user cannot interact with the rest of the application while it is open. This means that it is not possible to click on a reported warning or error and navigate directly to the location in the interface where the issue originates.

Figure 5 shows the current modal error reporting behavior in EZ-EDS.

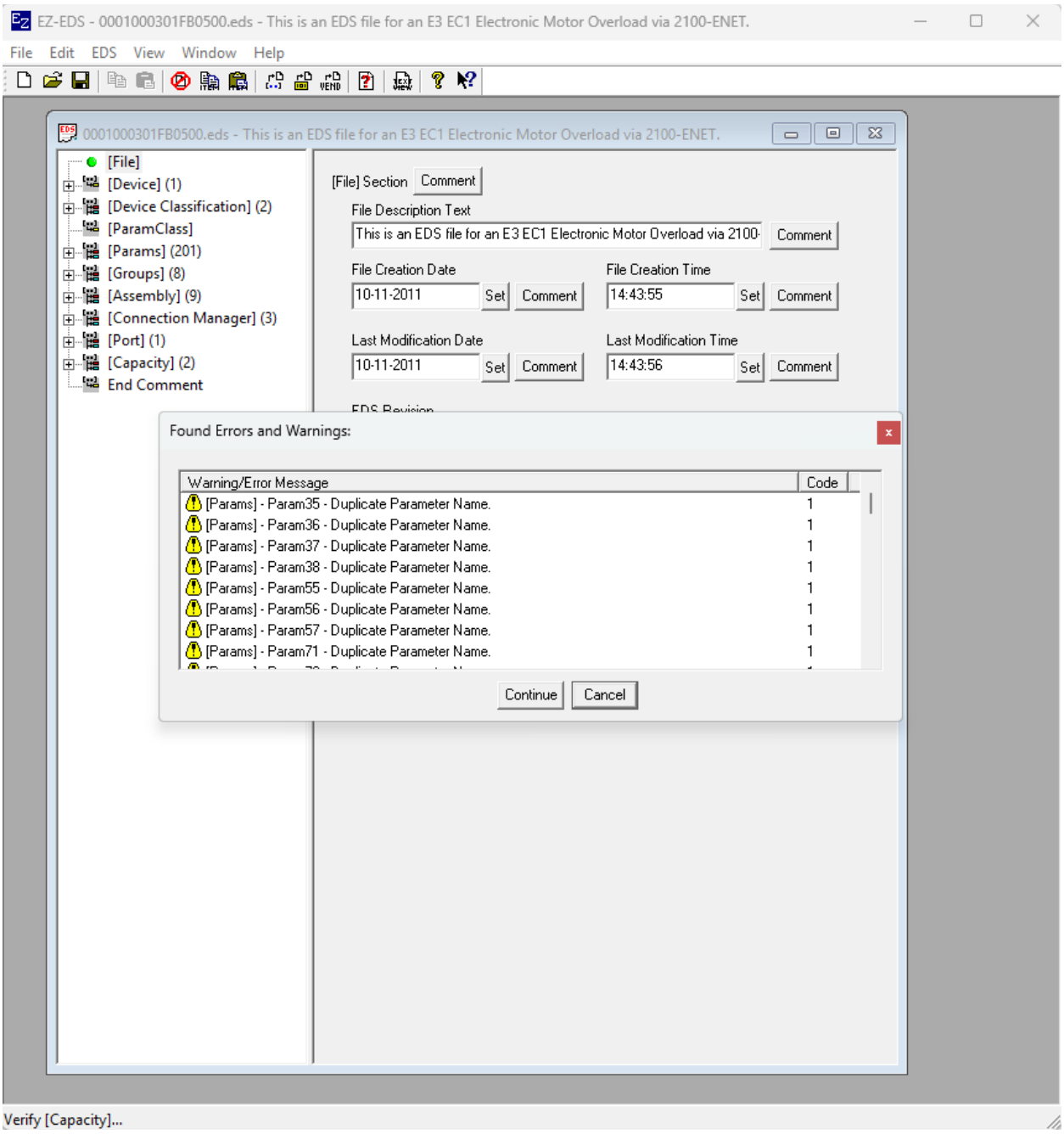


Figure 5 EZ-EDS Current Modal Error and Warning Dialog

The practical consequence of this limitation is a cumbersome and error-prone workflow. Upon encountering a list of warnings or errors, the user must read and memorize the reported issues, dismiss the modal, and then manually navigate through the interface to locate the source of each problem. If the issue is not immediately found, the user must trigger another save or load action to reopen the modal and re-read the original message. This cycle is particularly inefficient when multiple issues are present, as the user must work through them sequentially without any persistent reference to the full list.

A more effective approach is to surface warnings and errors in a persistent diagnostic panel docked at the bottom of the application window. This panel would remain visible at all times while the user navigates the interface, eliminating the need to memorize error details or repeatedly trigger validation actions to re-read them. Where possible, individual entries in the panel should be clickable, navigating the user directly to

the relevant location in the interface — the specific section, entry keyword, or field where the issue originates. This direct navigation capability significantly reduces the time and effort required to locate and resolve issues, particularly in larger or more complex EDS files.

The diagnostic panel also enables filtering capabilities that are not practical in a modal context. Users would be able to filter the displayed issues by severity — separating warnings from errors — or by EDS section or entry keyword, allowing them to focus on a specific area of the file without being distracted by unrelated issues. This makes the panel useful not only during active problem resolution but also as a general overview of the current compliance state of the file.

Figure 6 shows a mock-up of the proposed diagnostic panel, illustrating how warnings and errors would be presented at the bottom of the EZ-EDS interface.

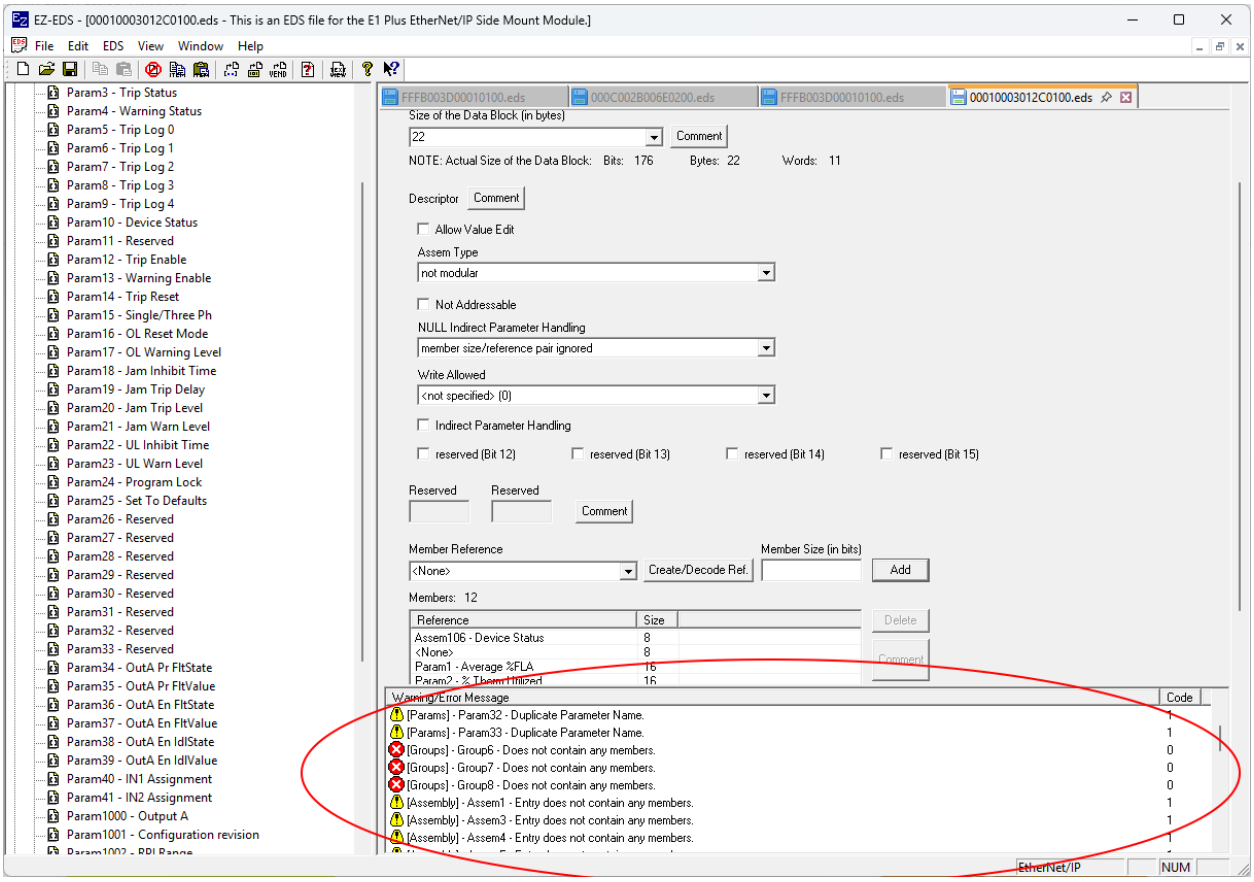


Figure 6 EZ-EDS Proposed Inline Diagnostic Panel

Automated EDS Generation

The manual authoring workflows discussed elsewhere in this paper address the case where an EDS author works directly within EZ-EDS to construct a device description. For many CIP product developers, however, particularly those supporting a family of related products or operating within a build automation environment, manual authoring is not the most effective approach. A recurring need among these developers is the ability to generate one or more EDS files programmatically from a structured input format, such as JSON or XML, rather than through direct manual editing.

This represents a fundamentally different mode of EDS development than the interactive, GUI-based authoring supported by EZ-EDS today. Automated generation treats the input file as the authoritative source of device description data, with the EDS file produced or updated as a derived artifact. Because this workflow is intended to be invoked programmatically — whether by a developer at the command line or by a build system with no user present — it cannot be delivered through an integrated development environment or similar interactive tooling. It requires a command line interface (CLI).

Figure 7 shows a general overview of how an automated EDS generation workflow could look in practice. Device information is captured in an intermediate format, such as JSON or AML, which can be produced or consumed by a CI/CD system — including through an EDS plug-in — transformed as needed, and ultimately used to generate or update .eds files. Command line tools sit at the center of this workflow, operating on the intermediate format directly and enabling both direct developer use and integration with upstream tooling, including GUI-based and AI-assisted development environments.

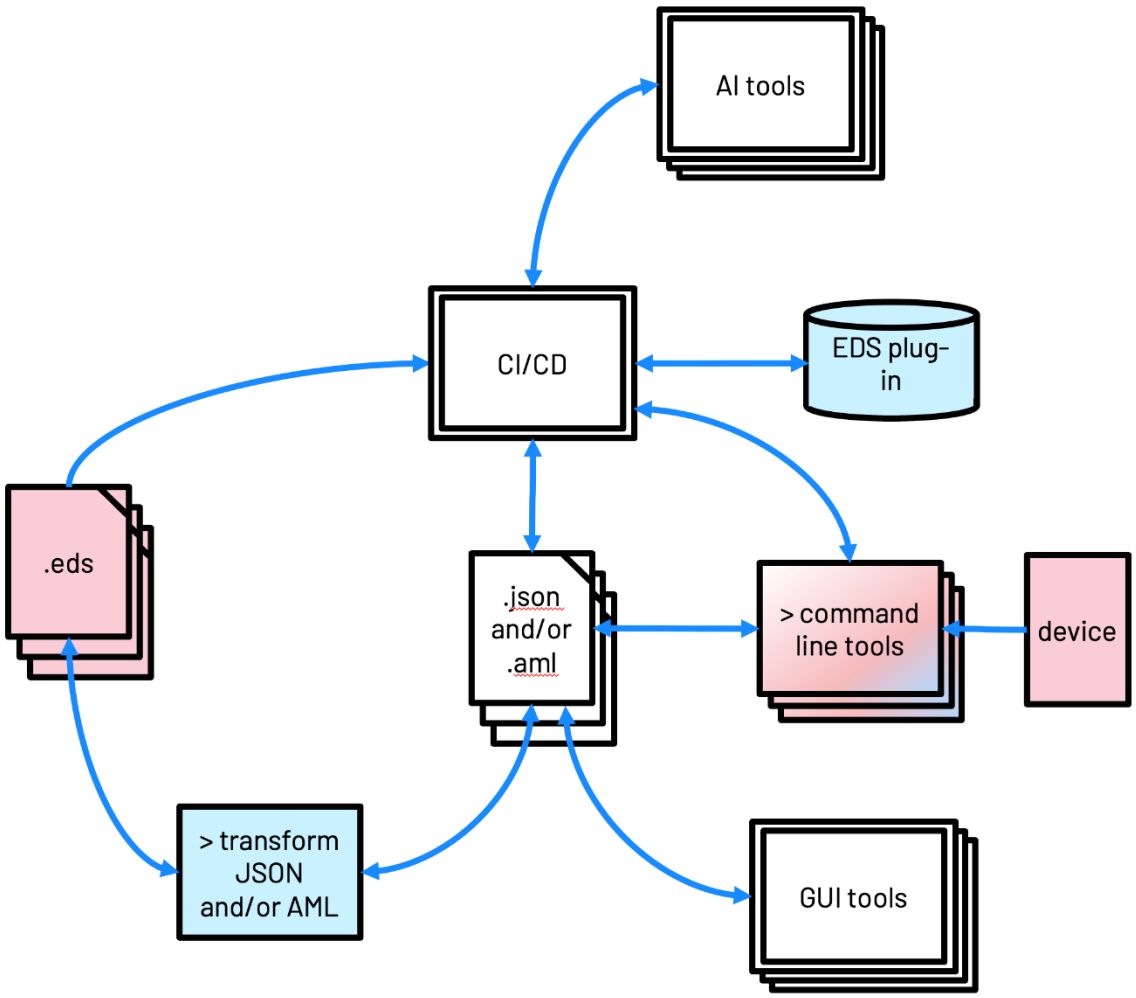


Figure 7 Overview of an Automated EDS Generation Workflow

Two related capabilities support this need.

The first is the batch update of existing EDS files. A developer or support engineer maintaining a family of CIP products often needs to apply a common set of changes across many existing EDS files at once. Rather than editing each file individually within EZ-EDS, a companion file — for example, a JSON file in a defined format — supplies the new or updated information. The tool applies this companion data against each target EDS file: content that does not yet exist in the EDS is added, and content that already exists is updated or replaced with the value found in the companion file. This allows coordinated, repeatable updates to be applied across a product family without manual intervention in each individual file.

The second is the generation of a new EDS file from an input specification. Rather than authoring a file from scratch, a developer working on a new CIP product can instead define the device description in a structured input file, such as JSON or XML, and have the tool generate a complete, valid EDS file from that input. As with batch updates, this generation capability is expected to be exposed through a command line interface, allowing it to be invoked directly by a developer during day-to-day development work and, equally, incorporated into a CI/CD pipeline. In the latter case, EDS files can be regenerated automatically with every build and made available to automated testing without any manual step in between.

In both capabilities, the underlying pattern is consistent: an automated, CLI-driven process replaces manual EDS authoring, using a structured input as the source of truth and producing or updating EDS files as output. This positions automated EDS generation as a more advanced complement to the manual authoring workflows described elsewhere in this paper — one that scales naturally to the needs of product families, continuous integration pipelines, and automated test environments.

Conclusion

This paper has outlined the challenges with the creation of EDS files, along with the gaps in tooling needed to align with developer and user workflows, and contemporary best practice for creation of device descriptor files. As the requirements for EDS files become more stringent with optional requirements starting to become mandatory for conformance, the tooling needs to evolve to make it easier for developers to create conformant EDS files. At the same time, improvements to the user interface present an opportunity to improve the experience of users wanting to make the use of EDS files.

We have examined two high-level approaches to EDS file development. The first is manual editing using an IDE or GUI-based tool, an approach that most developers will use at least initially and that, for many, remains the only tooling they need. The second is automated EDS generation — a more advanced case, oftentimes building on a file already developed in an IDE or GUI tool, that is used to update EDS files programmatically, potentially as part of a CI/CD environment. These are different use cases, requiring different tooling and, in most cases, serving different types of users.

In either case, the current EZ-EDS tool does not fully meet the requirements. It needs to be refreshed and expanded to support both manual and automated workflows going forward. However, the EZ-EDS code base, while well written, is closely tied to a very old development environment, and no straightforward migration path exists. As a result, any new tooling would likely need to be built on a modern programming language and development environment rather than extending the existing code base directly. A promising foundation already exists in this regard: the Conformance Testing tool has, over the past several years, been extended to include an EDS parser and validator. These components are fully separate from the rest of the Conformance Testing toolset and code base, and could be leveraged should the development of an EZ-EDS replacement move forward.

The question for the ODVA community, then, is what tooling ODVA should provide to make the development and maintenance of EDS files — for individual products as well as entire product families — easier going forward.

References

[1] ODVA, The CIP Networks Library, Volume 1: Common Industrial Protocol, Ann Arbor: ODVA, Inc., 2001-2026.

****

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.