ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUSDSC03

INTEGRATING TOOLS TO AID THE AUTOMATION OF PLC DEVELOPMENT WITHIN THE TwinCAT ENVIRONMENT

N. Mashayekh*, B. Baranasic, M. Bueno, T. Freyermuth, P. Gessler, S. T. Huynh, N. Jardón Bueno, J. Tolkiehn, L. Zanellatto, European X-Ray Free-Electron Laser, Schenefeld, Germany

Abstract

Within the myriad of day to day activities, a consistent and standardised code base can be hard to achieve, especially when a diverse array of developers across different fields are involved. By creating tools and wizards, it becomes possible to guide the developer and/or user through many of the development and generic tasks associated with a Programmable Logic Controller (PLC).

At the European X-Ray Free Electron Laser Facility (EuXFEL), we have striven to achieve structure and consistency within the PLC framework through the use of C# tools which are embedded into the TwinCAT environment (Visual Studio) as Extensions. These tools aid PLC development and deployment, and provide a clean and consistent way to develop, configure and integrate code from the hardware level, across to the Supervisory Control And Data Acquisition (SCADA) system.

Keywords: PLC, TwinCAT, C#, software tools, extension development

INTRODUCTION

Whilst there were tools [1] previously developed in order to aid Programmable Logic Controller (PLC) project generation at the European X-Ray Free-Electron Laser (EuXFEL), overtime, these tools became harder to manage. Many of the tools were developed in an array of programming languages, and require the PLC developers to become adept in multiple Integrated Development Environment (IDE) and languages. In turn, in order to enhance or add to an existing tool or function, edits would have had to be made across the multiple applications to ensure consistency. This approach can work within a diverse and well integrated team, however, also caused bottle necks and had a high dependency on all of the tools being kept up-to-date. This constriction was highlighted as an area which could definitely be improved upon.

A new approach was envisioned where all of the function previously being performed either manually, or via some means of automation, was collated together into a Single Point of Contact (SPoC). Provided with the backdrop of the TwinCAT environment, it was a logical step to build upon this platform by integrating this new functionality and interface into TwinCAT itself via the means of Visual Studio extensions.

THE NEED FOR TOOLS AND WIZARDS

TwinCAT's integration with Visual Studio makes it convenient to combine Visual Studio extensions with the TwinCAT Automation Interface library. This combination is highly beneficial for automated PLC project generation, code injection, and hardware linking. Visual Studio extensions enable custom tools and workflows that seamlessly integrate into the TwinCAT environment, while the TwinCAT Automation Interface library provides programmatic access to TwinCAT's features; allowing for the automation of tasks and the adaptation of useful features associated with modern programming languages. Together, these tools enhance productivity and reduce the need for switching between different applications whilst reducing the potential for errors within automation workflows at EuXFEL.

VISUAL STUDIO EXTENSIONS

A solid foundation provides the ideal canvas upon which to develop auxiliary tools. By utilising Visual Studio extensions within the realm of C#, it becomes feasible to import an in-house configuration seamlessly and to also dynamically modify both the project node, and the hardware node of the project in an adhoc manner.

This capability enables developers to make on-the-fly adjustments and closely monitor the success or failure of each step throughout the process. Consequently, this real-time feedback loop facilitates efficient and flexible project management, allowing for swift adaptation to changing requirements in addition to a more streamlined development process. Also in response to TwinCAT's lack of support for generic data types, developers can create Visual Studio extensions as a workaround to generate duplicated function blocks (Fig. 1) for different data types (Fig. 2).

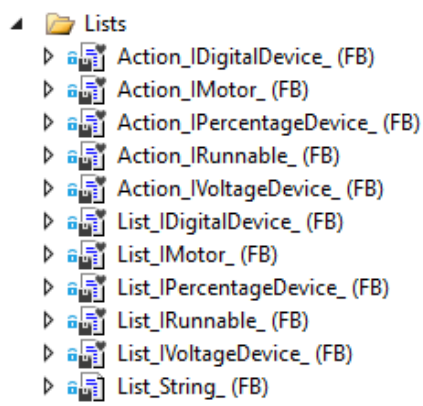

Figure 1: List implementation for multiple interfaces.

* navid.mashayekh@xfel.eu

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUSDSC03

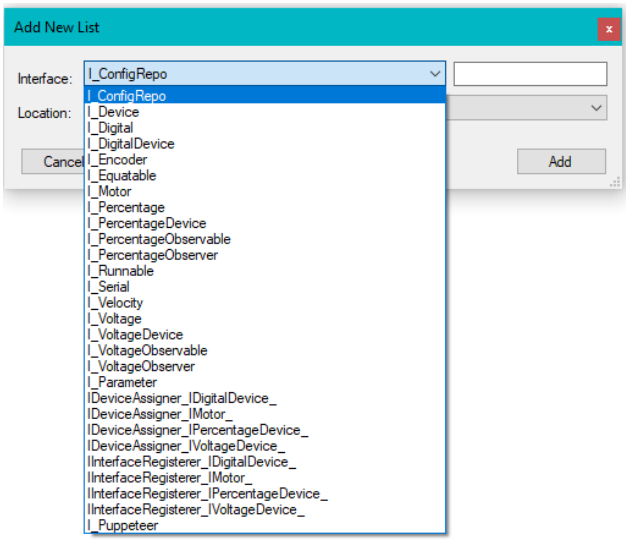

Figure 2: A list of all available interfaces created by user.

These extensions enable automation of the code generation process by allowing developers to define templates that can be customized for specific data types. When invoked, the extension generates multiple copies of function blocks with the necessary modifications to accommodate different data types, effectively emulating generic functionality. This approach streamlines the development process, reduces code duplication, and ensures consistency while working within the constraints of TwinCAT's limitations.

Developing Extensions

Developers can utilize the Visual Studio Extensibility (VSIX) [2] project template in Visual Studio to create extensions. There are multiple alternatives to customise and enhance the functionality of Visual Studio to their specific needs. Amongst all of the available customisations, the tool window and toolbar button are two which are noteworthy.

A tool window is a dockable window in Visual Studio that can host various controls and components, offering a customized workspace within the Visual Studio. To create a tool window, typically a Windows Presentation Foundation (WPF) user interface is defined [3] which implements the necessary logic to handle user interactions. Toolbar buttons on the other hand, are UI elements placed on toolbars within Visual Studio. These buttons trigger specific actions or commands when clicked, therefore they are suited for simpler actions which do not need complex user interface.

The implemented Visual Studio extensions for TwinCAT are structured into three distinct subsets (Fig. 3). Encapsulation and Library Handling, core logic, and lastly, user interface and panels.

Encapsulation and Library Handling In the first subset, the handling of the TwinCAT3 Automation Interface Library [4] is performed, in addition to the the encapsulation of the data to a new model format. This ensures that a streamlined and structured flow of data is adhered to. This

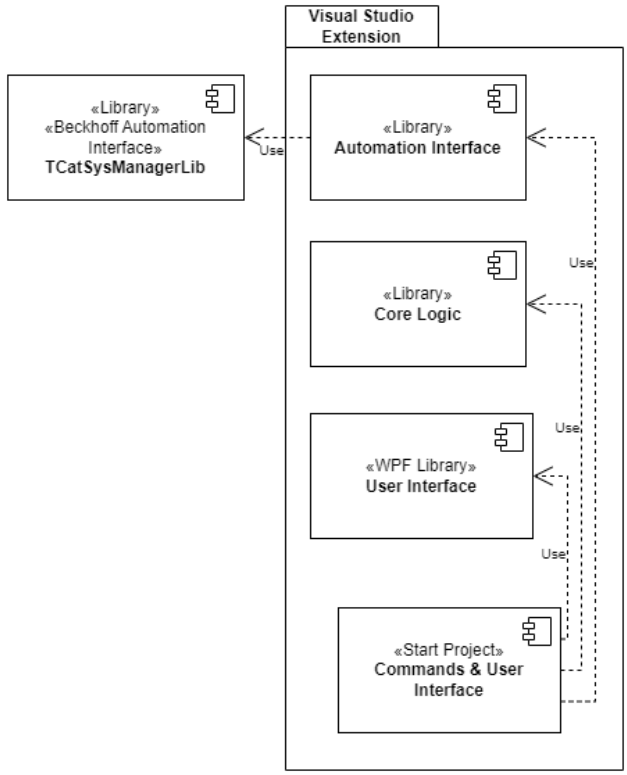

Figure 3: Design Structure.

foundational layer acts as the backbone, providing a reliable interface between the extension developed and to TwinCAT's core functionalities.

Core Logic The second subset contains the core logic responsible for driving the various actions and features of the extension. By keeping this logic separate, we maintain a clean separation of concerns, making the code base modular and adaptable to changes. One example of this includes the helper classes to parse EPLAN or ESI files.

User Interface and Panels Lastly, the third subset focuses on the user interface and panels, providing a polished and intuitive experience for users. Merging core logic and user interface into a simple toolbar button interface, can be an efficient coding practice, minimizing complexity and improving readability.

This well-organized structure not only enhances the maintainability of our code but also sets a solid foundation for future expansions and feature integrations.

Visual Studio Versioning

Due to the necessity of creating distinct projects to fully support different versions of Visual Studio can lead to significant duplication of effort. As a result of this, a strategic decision was made. The approach taken is to focus solely on supporting TcXaeShell v15, which is built on Visual Studio 2017. This decision was made with careful consideration to optimize development efforts and resources. By concen-

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUSDSC03

trating on this specific version, the development team can avoid the complexities associated with managing multiple codebases tailored for different Visual Studio versions.

This choice is temporary and pragmatic, intended to streamline development workflows. It ensures efficiency and avoids unnecessary duplication until Beckhoff officially releases a new version of TcXaeShell based on Visual Studio 2022. Once the new version becomes available, the development efforts can be expanded to support the latest Visual Studio iterations, ensuring compatibility with the most up-to-date technologies while minimizing the complexity and maintenance challenges associated with managing multiple codebases simultaneously.

Deployment Process

By the TcXaeShell restrictions where extensions can only be manually added by copying and pasting files into specific folders, achieving continuous deployment poses challenges. The lack of automated deployment mechanisms means updates cannot be pushed seamlessly to other systems. Consequently, for now, we are constrained with implementing continuous deployment practices.

Once the anticipated update by Beckhoff is released, enabling compatibility with Visual Studio 2022, our development team plans to initiate an internal extension gallery. This gallery will serve as a centralized hub where all our developed extensions will be hosted. With this infrastructure in place, deploying new releases will become more streamlined and efficient. Colleagues will have access to the latest versions of each extension, ensuring everyone benefits from the most up-to-date features and improvements. This also adds a level of consistency across the board.

GENERATION OF PLC CODE USING EXTENSIONS

Due to the high repetition of code usage within the PLC code base, the extensions developed are predominately catered the generation of PLC code, to both adapt and create new functionality, and to also generate a PLC project, with all the various libraries embedded and their methods and/or functions referenced.

Current Extensions

There are two fundamental approaches to generating PLC code via the Beckhoff Automation Interface. Those that develop or generate new Program Organization Units (POUs) to provide new functionality, and those that utilise existing POUs.

Generating New POUs The first method involves utilising predefined templates, where the developer creates a new POU by modifying keywords within these templates. This method is considered best practice when creating entirely new POUs and then adding them to the project. This approach is particularly useful in scenarios where complex data structures such as lists and dictionaries, or implementing finite state machines are being developed. In such cases, conventions are generally followed, and similar structures can be used repeatedly. With the ability to do this via a toolbar button, developers can initiate the creation of these objects, input necessary data through a user interface (such as specifying data types), and have the required POUs generated, and injected into the project's appropriate folders and sub-folders.

Modifying Exisiting POUs On the other hand, the second method is more convenient when dealing with existing POUs. This approach is ideal when developers need to modify existing POUs by injecting new code lines, instantiating new function blocks, or making changes in the hardware tree.

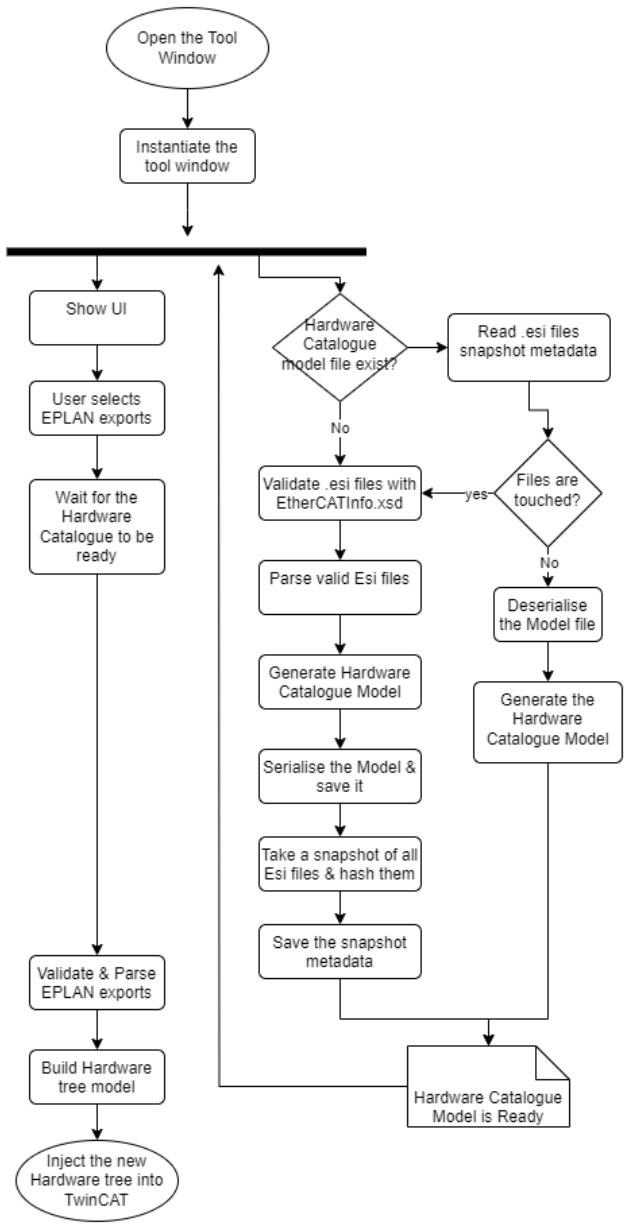

Figure 4: Generating the hardware tree.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TUSDSC03

An illustrative example is at EuXFEL, where the hardware layout is designed using EPLAN [5], generating exportable data that can be parsed and converted into the TwinCAT hardware tree definition (Fig. 4). Meanwhile, all of the EtherCAT Slave Information (ESI) files are internally parsed by the extension to create a comprehensive device catalog model. Parsing these files involves extracting vital information about the EtherCAT devices, which is essential for configuring communication and interaction within the system. However, this parsing step is both time-consuming and resource-intensive, consuming significant machine resources during execution.

To optimize performance and conserve resources, the entire device catalog model, once parsed, is serialized into JavaScript Object Notation (JSON). This serialized data is then saved locally on the user's computer for future use. By storing the parsed information, the extension avoids the need to repeat the resource-intensive parsing process each time the program runs, ensuring a more efficient user experience.

Additionally, to maintain the accuracy of the catalog model, metadata for all parsed files is generated and saved alongside the model file. This metadata serves as a reference, documenting the source and structure of the parsed data. Importantly, in case a new device is added to the TwinCAT catalog, the existing models are deleted, and the extension automatically regenerates the catalog model. This process ensures that the catalog remains up-to-date, and accurately reflects the current hardware catalog. Finally hardware nodes can be created by modifying corresponding parent nodes and adjusting their data using generated hardware catalog model, ensuring seamless integration of modifications into the existing project structure.

Future Extensions

Looking ahead, the future promises even more transformative advancements. By integrating the capability to install in-house and required external libraries, our extension aims to create a seamless development experience. Parsing project configuration data ensures that the extension understands the specific requirements of each project, allowing it to intelligently utilize these libraries.

The introduction of our brand new framework (TcZookeeper Suite) design, adds another layer of innovation. As TcZookeeper Suite is still in progress, it is hoped that the extensions will be adaptable to the evolving developmental needs of the facility. Works are currently underway to ensure the ability to generate project source code automatically. This automation drastically reduces the time and effort traditionally spent on manual coding, enabling a much faster and error-free development process.

Furthermore, the automatic linking of hardware mapping adds another level of sophistication. By handling this complex task automatically, our extension ensures that the software and hardware components are seamlessly integrated. This not only reduces the chances of errors but also significantly accelerates the overall project development lifecycle.

CONCLUSION

In the realm of TwinCAT development, the evolution of Visual Studio extensions stands as a beacon of innovation and efficiency. Through this paper, we explored a meticulous approach to extension development, emphasizing structured organization and modular design. Dividing tasks into subsets, such as managing the Beckhoff Automation Interface Library and intelligently creating or modifying Program Organization Units (POUs), ensures a streamlined and error-free coding experience.

With these extensions at their disposal, the TwinCAT development landscape is poised for a revolutionary change, marking a new era of efficiency, precision, and creativity in industrial automation.

ACKNOWLEDGEMENTS

The PLC team and authors of this paper worked closely with other EuXFEL scientific support groups and acknowledge their continuous efforts, input and cooperation. We thank the rest of the Electronic and Electrical Engineering (EEE) group, the Information Technology and Data Management (ITDM) group and the Controls Software and Data Analysis groups.

REFERENCES

[1] S.T. Huynh, H. Ali, B. Baranasic, N. Coppola, T. Freyermuth, P. Gessler, *et al.*, "Automatic Generation of PLC Projects Using Standardized Components and Data Models", in *Proc. ICALEPCS'19*, New York, NY, USA, Oct. 2019, pp. 1532–1537. doi:10.18429/JACoW-ICALEPCS2019-THAPP01

[2] VSIX Project template, https://learn.microsoft.com/en-us/visualstudio/extensibility/getting-started-with-the-vsix-project-template?view=vs-2022

[3] Extend and customize tool windows, https://learn.microsoft.com/en-us/visualstudio/extensibility/extending-and-customizing-tool-windows?view=vs-2022

[4] TwinCAT 3 Automation Interface, https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_automationinterface/index.html&id=3954232867334285510

[5] EPLAN, https://www.eplan.de/