TwinCAT with TR88.00.02 Methodology

Project Planning

Version: 3.0.0
Date: 27.09.2022

BECKHOFF

Contents

1 Foreword 5

1.1 Notes on the Documentation 5

1.1.1 Trademarks 5
1.1.2 Patent Pending 5
1.1.3 Copyright 5

2 Overall Layout 6

2.1 There is no one single “right way” to write code 6
2.2 Plan the code 6
2.3 Layers and interfaces 6
2.4 Interfaces allow for Exchange 7
2.5 Machine Built from Layers and Interfaces 7

2.5.1 Abstraction 8
2.5.2 Implementing Interfaces 9

3 Getting Started 10
Machine vs Equipment Module vs Component 10

3.1 Where does one level end and another begin? 10
3.2 Figuring out which block is responsible for what is important. 10
3.3 What should each layer do? 10

3.3.1 Base Components 10
3.3.2 Equipment Module Level 11
3.3.3 Machine Level 11
3.3.4 Module Separation 11

3.4 Error response/Error Propagation 12
3.5 Error Logging vs Alarm Display 13
3.6 Error/Fault Categorization/Classification 14

4 Layout 15

4.1 Interfaces 15

4.1.2 Interface Data Type 18

4.2 Extending Components 19
4.3 Start Small and Extend to Add Features 19

5 Define the Functionality 20
6 Build a Small Machine 21

6.1.1 Safety Circuit 22

6.2 Components 23

6.2.2 Table of all Control Module Functions 23

6.3 Equipment Modules 24

6.3.2 Equipment Modules Production 25
6.3.3 Equipment Modules Maintenance 25
6.3.4 Equipment Module Manual 26

6.4 Machine 27

6.4.1 Machine Production Mode 28
6.4.2 Machine Maintenance Mode 28
6.4.3 Machine Manual Mode 29

7 Implementation 30
7.1 Library code 31
7.1.1 SPT Base Types 31
7.1.2 SPT Component Base 31
7.1.3 SPT Components 32
7.1.4 SPT Event Logger 36
7.1.5 SPT PackML Base 36
7.1.6 SPT Utilities 37
7.2 Start at the Top and work Down 38
7.2.1 Main 38
7.2.2 Machine 39
7.2.3 Pull Wheels 42
7.2.4 Sealer 44
7.2.5 Unwind 45
8 Components 48
8.1 FB_Cylinder Component 48
9 TwinCAT Event Logger 49
9.1 Excel Add-in 49
9.1.1 Installation 49
9.1.2 Tool bar 49
9.1.3 Tabs 49
9.1.4 Example for the Sealer 50
9.1.5 Translations 50
9.1.6 Generate TMC-File 51
9.1.7 TMC File in the PLC Program 51
9.2 PLC Program 51
9.2.1 Defining Alarm Arrays 52
9.2.2 Create Events 52
9.2.3 Raising/Clearing Alarms 52
9.2.4 Monitoring the Alarms (Equipment Module) 53
9.2.5 Monitoring the Alarms (Machine Module) 53
10 Appendix 54
10.1 Beckhoff Support and Service 54
10.2 Beckhoff Worldwide Headquarters 54

List of Figures

Figure 1 Code Hierarchy 8
Figure 2 Communication Between Layers 9
Figure 3 Error Propagation and Reaction 12
Figure 4 Module Hierarchy 15
Figure 5 I_Axis Interface 18
Figure 6 Component Basic Axis 32
Figure 7 Basic Axis 33
Figure 8 Component Basic Slave Axis 34
Figure 9 Basic Slave Axis 35
Figure 10 Digital Sensor 35
Figure 11 SPT Event Logger Table 36
Figure 12 Main 38
Figure 13 Pull Wheels 43
Figure 14 Sealer 45
Figure 15 Unwind 46
Figure 16 FB_Cylinder 48

1 Foreword

1.1 Notes on the Documentation

This documentation is intended only for the use of trained specialists in control and automation engineering who are familiar with the applicable national standards. It is essential that the following notes and explanations are followed when installing and commissioning these Components.

The responsible staff must ensure that the application or use of the products all safety requirements described, including any applicable laws, regulations, codes and standards.

1.1.1 Trademarks

Beckhoff®, TwinCAT®, EtherCAT®, Safety over EtherCAT®, TwinSAFE® and XFC® are registered trademarks of and licensed by Beckhoff Automation GmbH. Other designations used in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

1.1.2 Patent Pending

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents: EP1590927, EP1789857, DE102004044764, DE102007017835with corresponding applications or registrations in various other countries.

The TwinCAT Technology is covered, including but not limited to the following patent applications and patents: EP0851348, US6167425 with corresponding applications or registrations in various other countries.

1.1.3 Copyright

2 Overall Layout

The ideal approach is to have program Components that are standardized, easily transferred to different machines, flexible enough to be customized and modified, and re-usable so that core Components can be readily re-used amongst multiple machines and allow for future updates.

2.1 There is no one single “right way” to write code

There are however many best practices and different design approaches for writing software, the key item is consistency, when all programmers in a group follow the same approach to writing code, code is more readable, re-usable, and robust. Software development has not changed much since its inception. Software development tools have gotten a lot better and easier to use but base concepts such as Object-Oriented Programming have not changed since its creation in the 1960’s. The problem is many of these ideas and tools passed the PLC world by until now, now the PLC world is catching up. Cabinet designers and mechanical designers are very familiar with revision control. Checking parts of drawings out, editing them and checking back in is common practice. These processes originated in the programming world, and they were so successful, drawing packages have implemented them. PLC Software development tools now have these same capabilities (often free of charge) that the rest of the programming world has had for decades. Take advantage of these tools. The aim of this document is to present a few different ways how things can be done. It is up to the individual companies/teams to decide which ideas (if any) they like and will implement. PLC software development is no different to any other software development and it should be treated as such. All the tools available to the software development world are also available for PLC software development. When code is written in a consistent documented manner it will be easier to read, easier to modify, easier to re-use and easier to update. This saves money.

2.2 Plan the code

Very few physical things are built by trial and error, not unless there is lots of material to destroy and time to rebuild. So why does it happen with software? No one would ever say to the mechanical department, “Hey we've had some steel here for an hour someone should start building the frame. At least put the feet on the frame, we know the machine will need feet, that can be done right now, you weld those feet on to that beam”. Everything is first planned, then built, after that, it mostly fits together and there is very little re-work. No plan? Then in the beginning lots of things get built, it looks like progress is fantastic as things are being completed, but when it’s time to bring them together, they don't fit and they must be re-built and re-worked and all that perceived progress is gone, lots of time and material is wasted re-doing things. Code is no different, the more complete the plan the less re-work later.

2.3 Layers and interfaces

Everything, absolutely everything is accomplished via smaller Components that work together. Break down any problem into small manageable complete pieces. Components interact with each other in layers via interfaces. Banking and bank machines are a perfect example. There are three layers. Customer, Bank Machine and the Bank’s Account Database. The three talk to each other via interfaces. An interface is the simply the common connection at the boundary between two Components. Where each Component can say, “My responsibility ends at the

interface" or "You don't get to see anything past the interface". The interface between the bank machine and the customer consists of a bankcard and a pin. To take money out, a customer puts the card into the machine and enters a pin. The bank machine's interface to the bank is a secure network connection. Via the network connection the bank machine provides the account number, pin and how much money has been requested. The bank verifies the account, pin and availability of funds. If funds are available, the bank tells the bank machine to go ahead and release the money. The bank machine counts the money, presents the money, confirms the user has taken the money and tells the bank to update the balance. Any bankcard and pin, any bank machine, any bank, it all works. Pesos, Dollars, Euro it does not matter, the interfaces are clear, well defined, and everything interchangeable.

2.4 Interfaces allow for Exchange

A key Component of Interfaces is that they allow a Component to be exchanged without affecting the other side. Take for example a car. The interface for using a car is the steering wheel, gas pedal, and brake pedal. Every licensed operator can use this interface, what goes on past that, does not matter and should not concern the user. A diesel car can be exchanged for a gasoline or electric car, and it does not matter any, any operator can drive the car. When the interface is different, such as manual gearbox with a clutch, then not every operator can use, only operators that understand the extra interface can use the car.

The same works in machines, developing controls or modules with interfaces allows Components to be exchanged without affecting other parts of the machine. Easier exchange of modules allows for more options with lower engineering costs and less risk.

Code should always be written in such a way that there is a defined interface to separate the inside of a piece of code from the outside. Interfaces should be as similar as possible. This makes it easier to change things later. If the interface to the car was to directly adjust the carburetor's fuel/air mix, then switching to an electric car is going to be significantly more work as the interface between the engines is drastically different.

2.5 Machine Built from Layers and Interfaces

A machine can be programmed in single block of code. It will be a nightmare to maintain, adjusting will break existing functions, and virtually none of the code would be re-usable. Every machine built in this manner starts from scratch. Programming, as with any problem, is best solved by breaking it down into manageable pieces. Dividing a problem into layers allows for abstraction (interchangeable parts), and breaking Components up into various pieces makes it easier for multiple programmers to handle simultaneously. Smaller Components are more likely to be re-used. Three levels appear to be about right number, and they are Machine Module, Equipment Module and Component. The Machine Module coordinates multiple Equipment Modules. Equipment module coordinates multiple Components, and a Component performs a single specific function.

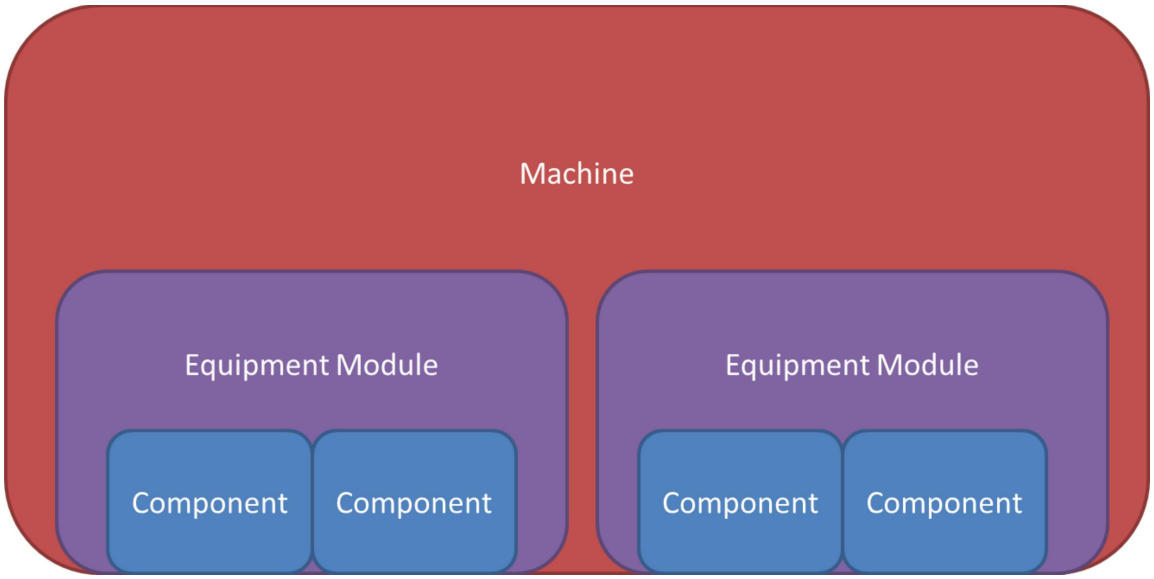


Figure 1 Code Hierarchy

2.5.1 Abstraction

One of the key reasons for having layers is abstraction. This is the idea of "I don't need to know how" or "I don't want to know how X works I just need it to go". For the car, "I push on the gas pedal for the car to go". I do not care how it goes; my job is to deliver the pizza not to figure out how to burn fuel to turn the wheels. This is abstraction; nothing past the interface concerns me as long as it does what the interface says it will. Give me a car and I will deliver the pizza. It is up to the people building and maintaining the car to ensure that when the gas pedal is pressed, the car will go.

For the machine to “do something”, it will issue commands to the Equipment Modules in whatever order is required for the machine to operate. The machine also responds to the status of the Equipment Modules, if an Equipment Module has a problem, the machine will determine what all other Equipment Modules must do. The machine only has to deal with Equipment Modules. The machine does not care which or how many Components the Equipment Modules have. Equipment Modules can be exchanged for other Equipment Modules and the machine can deal with this easily.

Abstraction can make things very efficient. If all Equipment Modules have the same interface, IE they accept the same commands and return the same status, then Equipment Modules become interchangeable. For example, A company has built two Unwind Equipment Modules, one using Servo Motors and one using DC motors, provided both Unwind Equipment Modules use the same interface, one can be replaced with the other without any changes to the programming of the machine. The machine programmer can focus on coordinating the modules to make everything run, rather than wasting time figuring out how to get a module to perform its function.

Equipment Modules work with Components in the same way. The Equipment Modules coordinate the Components to perform a machine function. The Equipment Module issues commands to the Components and monitors the status of the Components to perform the function requested by the Machine.

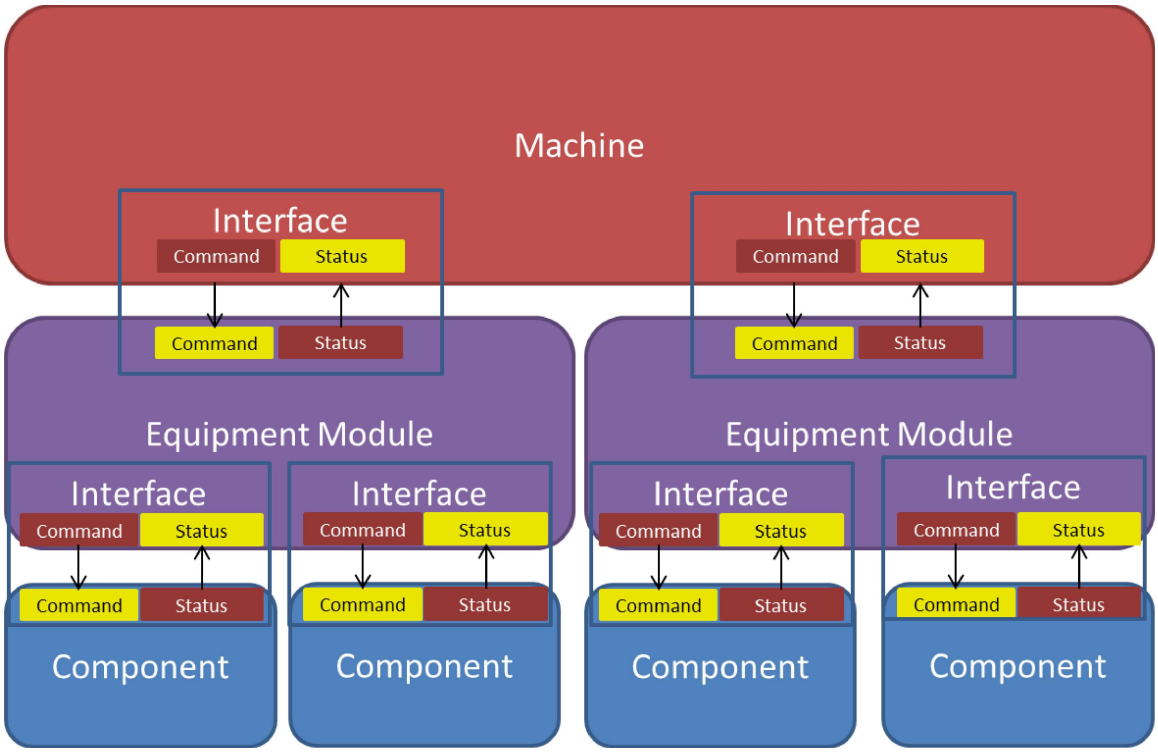


Figure 2 Communication Between Layers

Interfaces between layers should be as standardized as possible. For example, all axes should have the same commands and status. It would make no sense to have some axis to use the command "Reset" to clear faults and another axis to use the function "ClearError". It is far faster and simpler to build an Equipment Module if the Components have similar commands and status and it is faster and simpler to build the machine if Equipment Modules have similar interfaces.

2.5.2 Implementing Interfaces

There are many ways to implement interfaces, things like Variables, Methods, and Properties, but there is no single right way. All roads lead to Rome. The most important item is consistency. All roads might lead to Rome, but planes, trains and cars do not mix. Code should be written with the expectation that someone else (a user) will take the completed Component and implement it. The "user" never has to look inside a Component to have to get it to work. If a user must look inside a block to figure out how to use it, the user will generally throw the block out and write a new one. This new block now must be tested and debugged, a huge waste of time and money. Command Structures, Status Structures, Error Messages and Variable Naming should all be consistent between levels/modules. Exceptions to the standard, or blocks of code that behave differently require more time to implement, have more errors, take more time to debug/commission, are less likely to be re-used, and at the end of the day cost more money.

3 Getting Started
Machine vs Equipment Module vs Component

3.1 Where does one level end and another begin?

Determining what each level will do takes some time and must be well thought out. This is the core design, think it through, time spent here saves time later.

What happens when the layers are not well defined, or the interface is not clear?

Back to the Bank Machine example, there is an easy way to build a bank machine. Take a box of money, embed it in concrete and put a bunch of unlocked combination locks next to it. Give the serial numbers and combinations of the locks to the bank. To use the “bank box”. The user goes to the “bank box” and phones the bank. The user gives the bank their account number, pin, the serial number of the combination lock currently on the “bank box”, and how much money they want to withdraw. The bank looks up how much money is in the box and how much money is in the account and says OK here is the combination to that lock. Take your money, then take a new lock, and put it on the box. The bank then subtracts the amount of money the user told them from their account balance and updates how much money is left in the box. When the bank finally gets around to putting more money in the “Bank Box” the balance will almost certainly be wrong. At which point the “Bank Box Company” says, “It’s not my fault the user took out more money than they said they did, they didn't use it as described in the terms and conditions"

When the Components or layers are not properly defined and the responsibility of tasks is not correctly assigned, the phrase: “X works fine, they didn’t use X correctly” is commonly heard. If this is happening, either it is not clear how to use the interface, or the user of the Component is performing tasks that the Component should do itself. In the “Bank Box” machine example the user and the bank are doing the "Bank Box’s” tasks.

3.2 Figuring out which block is responsible for what is important.

The problem “you are not supposed to take out more money than you said you would” is easy to avoid when the bank box is responsible for dispensing the money and keeping track how much money is in the box. After a bit of examination, anything happening inside the box should be the boxes responsibility. The user should never be able to "see inside". The Bank should not have to keep track of how much money remains in the box before approving the withdrawal. The bank is worried about how much money is in the bank, not the box. If the user needs to look inside the Component to use the Component, the interface is incomplete. If someone needs to know how much money is in the box, the box should keep track of this and provide this value in the interface. If the user needs to provide external logic/functionality in order to get the block to do what it is supposed to, the functionally is incomplete and tasks need to be re-assigned.

3.3 What should each layer do?

3.3.1 Base Components

Components are individual devices that perform a task, but they do perform a “Machine function”. For example, a solenoid can move forward and back, and axis can spin and stop at a position but these things are not a machine function. An Equipment Module will implement Components such as axis and solenoids together with sensors, and other devices to perform a machine function such as a “Back Gauge” or “Unwinder” “Unloader or “Filler”

A Component must be usable by any Equipment Module. For Example, in an axis Component, the Equipment Module function “Move Back Gauge” is implemented by commanding it to go to a specific position. The axis Component itself has no idea that it is controlling a Back Gauge or a Spindle or a Tension Roller, it simply has commands like “go to position X”. The Equipment Module defines that this axis Component is driving a Back Gauge.

Only Components talk to Hardware. Why? Abstraction. If only the Component talks to hardware, hardware can be replaced/exchanged without having to re-write the Equipment Module. If a DC motor Component has the same interface and functions as a Servo motor Component, then inside the Equipment Module one can be swapped for the other without making any coding changes to the Equipment Module. The Equipment Module’s only concern is the command “Go to position X” results in the Position X being reached. Servo may be faster and more precise, but so long as it goes to position X, the Equipment Module doesn’t care about the “How”.

3.3.2 Equipment Module Level

Equipment Modules, this is the important one, but it is also usually a more obvious one. The machine will be set up mechanically into individual modules. It is important to get the Equipment Module level right. If the Equipment Module is too low level, it is like driving the car by adjusting the airflow and fuel injection. The Machine level is doing things best done by the Equipment Module. Too high of a level and the Equipment Modules is doing too much work that the Machine should be doing. The car is doing everything including the navigation. One key to getting Equipment Modules correct is; If this module is removed from the machine, which Components need to be included to dry cycle it? Can this module perform its function without other Components? Can it be removed or replaced? An Equipment Module should perform a specific but complete function. An Equipment Module should not talk directly to hardware. It might need a sensors value and status, but a Component should be scaling the sensors value and monitoring the sensor to determine if the value is in an error range. A system can have as many Equipment Modules as it needs. Equipment modules might be simple, or they might be complex.

3.3.3 Machine Level

Determining the scope of the machine level is straightforward, it is the top layer, and it is what drives all other Components. When a machine is provided with materials, it can complete its entire function. The machine however is not just the Main Program. The machine will have to talk to an operator. The Human Machine Interface (HMI) or Operator Interface is often a GUI (Graphical User Interface), but it does not need to be. The Operator interface could be a push button and a light. For example, to start, turn the key, wait for the green light, and then press the green button. To stop it, push the red button. The main program may contain other programs to do other administrative tasks. For example, there may be a program to exchange data with the Operator Interface / HMI. There might be programs that connect to databases or IOT programs logging information etc. The machine level program as far as this document is concerned is dedicated to running the machine. The machine level block accepts commands, sequences the Equipment Modules, and reacts to alarms/errors of the Equipment modules.

3.3.4 Module Separation

As stated, the programmer at each level is concerned with their level and only their level. That is why the interfaces exist, they give a clear definition of where responsibility ends. A module

communicates only with one layer above and only with one layer below. For example, in the Equipment Module, it receives commands from the Machine Layer via it's inputs/method calls. The Equipment Module then communicates to the Components which operate their hardware. The Machine has no business communicating directly with a Component. This abstraction allows Components to be replaceable without requiring the machine to be re-written. For Example, replacing a Servo drive with a DC Motor in the Unwind. The Equipment Module simply says “Component” run at this speed. The machine is none the wiser.

3.4 Error response/Error Propagation

Part of the separation of modules is the error handling. Compartmentalizing functions also means the errors can be compartmentalized. Errors have different meanings at different levels, and each level gives the error context. "Axis 5 lag distance fault" is meaningless at the machine level. The machine wants to know, “Was this error severe enough to stop the entire machine?” and the operator wants to know “how do I get back in operation”. When a Component has an error, it reports its specific error. The Equipment Module sees the Component error and determines what it means for the Equipment Module and reports an Equipment Module error. The machine sees the Equipment Module error and finally determines if the entire machine must be stopped or not. Axis 5? Lag distance fault? The back gauge is not in position. Do not allow machining of the part to start, the part isn’t in the right place.

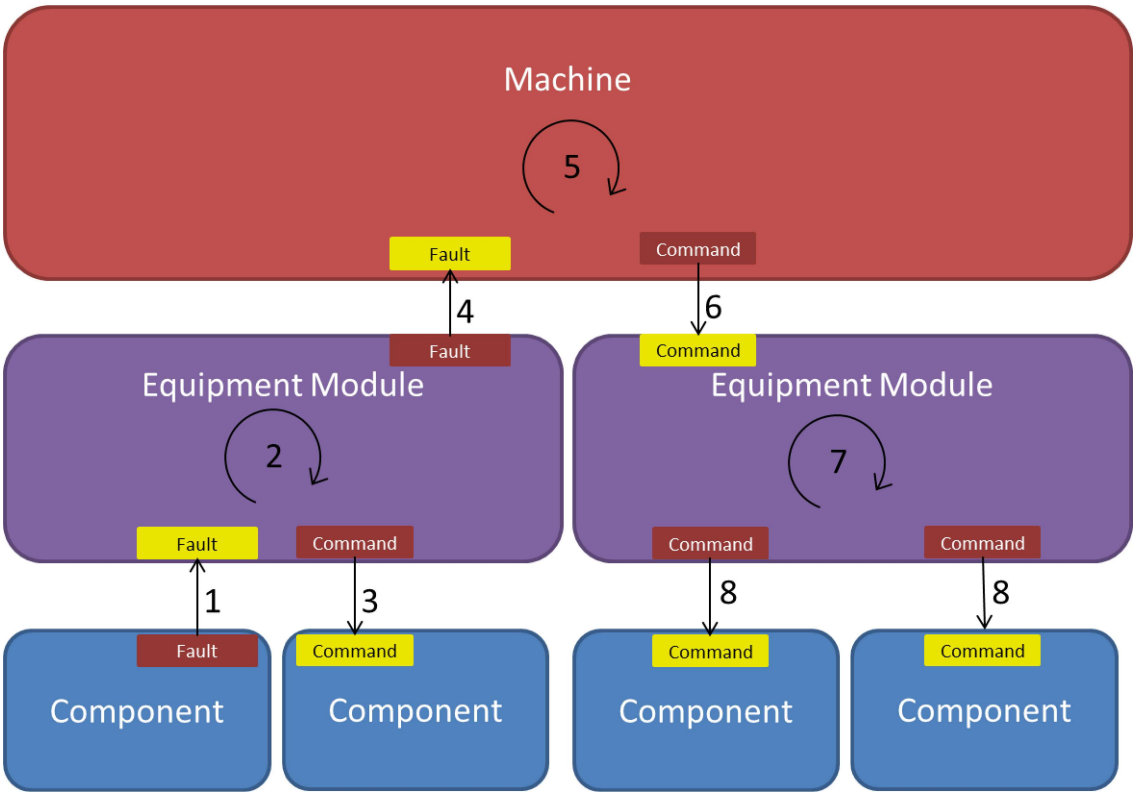


Figure 3 Error Propagation and Reaction

The complete sequence of error reporting and reaction is as follows:

1) Component reports error to the Equipment Module
2) Equipment Module determines the appropriate reaction for its other Components
3) Equipment Module issued commands to the Components
4) Equipment Module Reports the Equipment Modules Error to the Machine.

5) Machine determines appropriate reaction for the other Equipment Modules
6) Machine issues commands to all Equipment Modules
7) Equipment Modules process the commands
8) Equipment Modules issue commands to the Components

For example, a Machine has an Equipment Module that is an X-Y table. Within the Equipment Module are two Axis Components, X and Y. The Component for the X-axis reports a lag distance fault. The Equipment Module sees this lag fault on the X-axis, determines that the Y-axis should be stopped immediately; it stops the Y-Axis and reports to the machine that the X-Y table is no longer operational due to a problem with the X-Axis. The machine sees that the X-Y Table is no longer functional and, in this case, determines that without the X-Y table nothing should operate. The Machine commands all Equipment Modules to stop and cannot be re-started until the X-Y Table Error is fixed, and the operator is informed the entire systems stopped due to the X-Y Table.

3.5 Error Logging vs Alarm Display

Everything should be logged, but not everything should be displayed. Each Module must log all its errors and timestamp it. Logs are for Service and the Development Engineers; Alarms are how the machine tells the operator what needs to be done in order to run. Alarms must be clear and relevant, with the most urgent displayed first. The operator will generally not be looking at the screen when the alarm was raised. When the operator arrives at the machine it must be readily apparent where the problem lies and what to do. Like everything, there are several ways to handle this. This is an important topic some of the first things written in the control modules will be the error/fault reporting.

Following the error propagation sequence, a root cause error at a control module will generate 3 messages, one at the control module, one at the Equipment Module, and one at the machine level. There are some options for handling this.

1) Each module reports its own error and via ID numbering schemes it can be determined all 3 errors are associated and either the HMI or the PLC ties makes the association to tie them together.

2) The lowest level module generating the error reports it, higher levels add information to the same alarm.

3) Only the machine layer reports the errors.
    a. The Machine layer knows “is told” via either configuration or programming of every possible message from the Equipment Modules and control modules and manages them. This is often the “old school” way of doing things, every module/device provides a list of all possible errors and the machine and global error numbers are assigned.

4) Component faults are captured by the Equipment Module and only the Equipment Module provides the alarm up to the machine.

A traditional alarm handler looks through all the devices’ errors and determines what to report. With the core idea being modularity and interchangeability of Components, this ends up being a duplication of effort. Every module within the machine has already had to recognize and react to the alarm, now a full second set of logic must look thorough all the faults on all modules and determine what messages to display. It can generally be surmised that the Component faults are the root cause but that may not be the case, if a safety system is tripped many Components will fault and it can all happen in a single PLC cycle. Did the safety drop the power or did the

power drop cause the safety system to trip? Properly timestamping and raising the fault/error/alarm directly makes sequencing easy.

Another option is alarms “drill down” when looking at the machine layer, Equipment Module errors are displayed, when looking at an Equipment Module, errors from the Components are displayed. X-Y table has an error, drill down to the X-Y table and the X axis jammed. To see the X axis jammed, the operator will have to look at the X-Y table Equipment Module. As Operators become more familiar with the machine, they will learn that the X axis lag distance fault means a jam they need to go fix. The operator doesn’t want to have to click on everything to see the root cause.

For modularity, it is very easy to have every device write to the list in the order things happen. The problem is if there is text associated with each alarm. This text must be translated; this means list of text must be exported/imported. If each device has its own list, then a bunch of different lists need to be sent for translation. This is not a significant problem, text for the messages can be stored in files or databases, but it must be managed. If the text is handled by the HMI, then it will need to be provided with these files or databases so that it can perform the text lookup.

3.6 Error/Fault Categorization/Classification

Regardless of the alarm handling methodology employed for modules are going to have unique faults and need to be readily identifiable and probably lots of them. Developing an error reaction to every single fault is not practical. Fault Categorization/Levels can help with this. In order to react to a fault, the severity of the fault is important, not the specific error message. Is the fault bad enough that this module must stop? Do other control modules also need to stop? Setting up Levels for faults greatly reduces the programming/implementation effort and ensures consistency between Modules. Note levels of the faults can also indicate how the fault should be reacted to. For Example, Critical/Abort the Machine or Equipment Module must be stopped immediately. Cycle Stop faults allow the Equipment Module to "finish" but they cannot be restarted. There seems to be no definitive standard on alarm categories and classes. ISA 18.2 does cover alarms but for plant alarms with things like Category 1 is severe, risk of loss of entire plant or environmental contamination outside the facility a little beyond our scope. Some systems implement 4 categories some recommend more. The TwinCAT Event Logger logs 5 "PC event types" natively.

0. Verbose
1. Info
2. Warning
3. Error
4. Critical

These categories could be used to determine the severity of the alarm or to differentiate between debugging/logging and errors and what the reaction should be. For example, Critical means the module has aborted and cannot receive new commands.

Other systems use error categories that determine the course of action the machine should take.

1. Critical/Immediate Stop/Abort the Module/Component cannot/does not function
2. Severe/Cycle Stop The Module should be stopped and not commanded again.
3. Idle / machine functions hold preventing a next part to be started
4. Hold/local condition. The machine can function but is low on a material
5. Suspend/External. This Module/Machine is stopped waiting on other Modules/Machines from running.

6. Warning Machine/Module operates but the user requires information about a condition that could lead to a higher category fault
7. Info

A traditional variation on the theme is to allocate X bits to each module for each level where each bit represents an error. This makes some things easy, no bits set, no error but this only really works well up to 64 bits after that OR'ing multiple LWORDS gets complicated. 32 bits can be used to either represent 32 errors, or used as a UDINT, which can now represent 4.2 billion errors and via module ID's a scheme of errors can be configured to cover nearly every possibility. One advantage of this methodology is that all errors can be displayed simultaneously any error that is active is reported. If Error ID's are used, one possibility is the module only reports the first or highest priority error often this is the only one that matters. A second option is the module could log to each error to an error handle as they appear even multiple errors in the same cycle. The third option is that the Module contains a list of sufficient length to report all currently active errors.

Regardless of the scheme is chosen, be consistent and clear the modules need to log their errors and keep track of their active errors so the alarm handler can report/acknowledge the alarms accordingly.

Having levels/categories can make the Equipment Module and machine level programming significantly easier. If category/level 1 faults mean the module will no longer work, then it's simple at the machine layer to say, for any level 1 error from this Equipment Module stop the entire machine. Level one error from this module? Finish the cycle and don't start a new one.

4 Layout

Once it is known what everything must do, how to implement it?

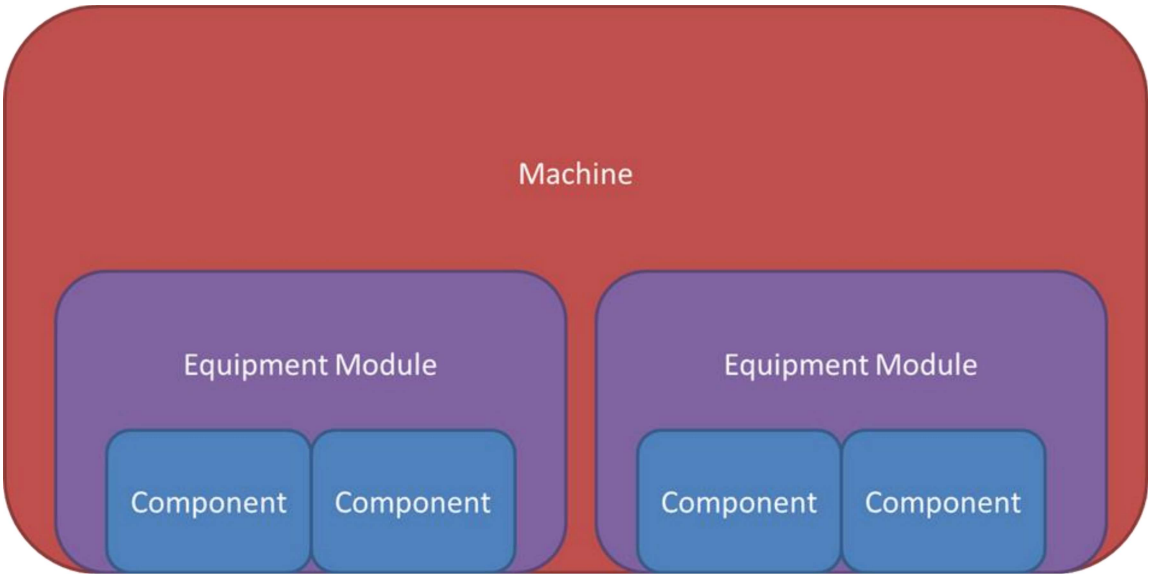


Figure 4 Module Hierarchy

Items at each level must talk to one level up and one level down. All blocks, regardless of what they are communicate via an interface.

4.1 Interfaces

As indicated before an interface is simply a defined way of two things interacting with each other. For code, that means how a piece of code A talks to with another piece of code B. A well-defined interface allows B to be replaced by C without requiring any changes to A. Back to the earlier car example, the diesel car can be replaced by electric car and the Intern is still able to go pick up lunch. Telling the Intern to take the skid steer https://en.wikipedia.org/wiki/Skid-steer_loader and go get lunch is a recipe for disaster. This vehicle's interface is drastically different and requires specialized training.

How do interfaces apply to PLC Code? All functions, function blocks, and add-on instructions have an interface. The input, output, and in/out variables define the interface. In TwinCAT 3 function blocks also have the capability of implementing Methods and Properties. Methods and Properties have been a staple of nearly major programming languages since the 1970's. Methods are functions/subroutines for the Function Block called by the user. Methods are things that the Function Block can "do". Method names should always be verbs. Properties are characteristics of the block, i.e. what the block "is" or "has" and should always be nouns. Back to the car example, methods would be Accelerate, Decelerate, Turn Left, and Turn Right. Properties would be things like Color, VIN Number and Number of seats. For the Bank Machine, Properties would be things like "CashInBox", and methods would be "CallBank" and "DispenseCash".

For blocks to be interchangeable, they must use the same interface. When building an interface it needs to be determined what functionality must be provided and what data will be available. Which methods and properties each block must support and which variables will be passed in and out. This process is a discussion, the user of the block and the builder of the block together need to determine what the interface will be. There will be changes as deficiencies are found.

Point to Point axis Components should be interchangeable. The ability to swap a stepper for a servo or a DC motor without changing the Equipment Module code is highly desirable. The same for the ability to swap one servo drive type/manufacturer for another. Different hardware will do things in different ways. The reset sequence for a stepper motor is very different from the reset sequence for a fieldbus connected servo motor. The Equipment Module that asks for a reset, does not care about "how" the reset is done, it just wants a reset done. Having separate Components that are interchangeable but "perform the same function" allows lower cost machine variation. Some customers may be willing to pay for high performance servo axis and some customers would prefer lower cost stepper motors. The easier it is to swap Components the lower the engineering cost for modifications.

For example, a simple Point to Point axis block could have the following interfaces:

4.1.1.1 Option 1: Methods, Properties, Inputs, Outputs and Var_In_Out**

Method	Property	Input	Output	InOut (cyclic)
Reset	Position mm	Enable	Error	Axis Reference
Move(Pos,Vel)	Position inch		Error ID	
Stop			Busy	
			Ready	

The interface defines the rules for the Axis block. The "user" of the block reads the interface and knows what an axis can do, the person who builds the Axis block reads the interface and knows what features/functions to provide. If all different versions of axis blocks use the same interface, then anyone who knows how to use one axis block knows how to use all axis blocks regardless of whether it is a Linear, Servo, Stepper, or Hydraulic axis. Same interface, same usage, directly exchangeable.

It is up to the company/programming team to define rules for creating interfaces. Should all inputs and outputs be handled by Methods and Properties, or via Input and Output variables or both? It is just as easy to write Axis1.Reset := TRUE; as it is to say Axis1.Reset(); The same for outputs, Axis1.Error or Axis1.GetError(). Position could easily be an output, instead of a property.

Here are 4 more options for the Point-to-point Axis Block interface. Each has its own advantages and disadvantages. Choosing a consistent interface layout will be key. Options 3 and 5 below have special advantage in that they can be declared as a type "Interface" which is described in the next section.

4.1.1.2 Option 2: Inputs and Outputs Only

Input	Output
Enable	Position Inch
Reset	Position mm
Move	Error
Move Position	Error ID
Move Velocity	Busy
Stop	Ready

This is the traditional approach for function blocks

4.1.1.3 Option 3: Methods and Properties Only

Method	Property
Reset	Position Inch
Mover(Pos,Vel)	Position mm
Stop	Error
Enable	Error ID
Disable	Busy
	Ready

4.1.1.4 Option 4: Inputs and Outputs With Command and Parameters**

Input	Output
CMD	Position Inch
P1	Position mm
P2	Error
Abort	Error ID
Reset	Busy
Start	Ready
	CMD Done

This Interface requires a documentation for what each CMD number is what the Parameters P1 and P2 represent for different commands. For example, CMD = 1 could be a Move command where P1 is position and P2 is Velocity. CMD = 2 could be stop where P2 is the deceleration.

4.1.1.5 Option 5: Methods and Properties with Commands and Parameters

Method	Property
Reset	Position Inch
CMD(P1,P2)	Position mm
Abort	Error

	Error ID
	Busy
	Ready
	CMD Done

Just like Option 4, Option 5 would need documentation about what each command is and what the parameters do for each Command.

4.1.2 Interface Data Type

For TwinCAT 3 there is an interface data type. Interfaces defined by a data type are much more rigid. When a defined data type interface is implemented, all Methods and Properties MUST be included. The code will not compile if a block implementing an interface does not do it fully. All blocks implementing an interface data type will be interchangeable the compiler forces this. This significantly improves the reusability and consistency of the code.

Variable Inputs, Variable Outputs and Var_In/Outs are not included in the Interface data type, and therefore the compiler does not force the programmer to use the same variable names. External documentation is required if programmers are to use the same variable names. Someone must document what the interfaces are to be (this can be done directly in the code) and the rules must then be enforced when documented interfaces are not being followed. "You didn't follow the interface." "You changed the name of the Execute variable to Go." "Go back, look at the documentation and fix it". When an Interface Data type is used, the code simply will not compile if the names of the properties and methods do not match to the declared interface. Again, there is no right/wrong choice here, what is important is that all interfaces are designed in a consistent way. Implementing some blocks with data type defined interfaces and others via variables, reduces readability, reduces re-usability, increases confusion, which ultimately requires more time/money than having a consistent system. Two formats may be appropriate, Blocks using inputs and outputs for the interfaces must follow a specific set of rules, as new blocks must be property and method only and follow another set of rules. Programming convention documents are not big documents, 3-5 pages is generally plenty.

A second huge advantage of using a Type Declared Interface is the ability to "code against the interface". When "coding against an interface" the programmer can declare an instance of the interface and write all code using that interface only. This is extremely advantageous when Interfaces have been defined but no code has been built. The Equipment Module programmer can write all the code for a Component before the Equipment Module is even started. The interface says it has a reset command and when it is complete it will be done or have an error. Eventually a Component that implements the interface will be completed. When it is ready to use, simply assign the instance of the block to an instance of the block. Now all commands to the interface are issued to the actual block.

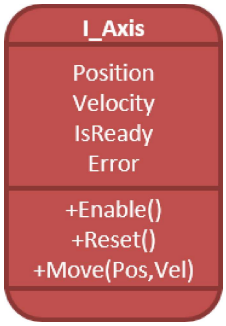


Figure 5 I_Axis Interface

With such an interface, there can several axis blocks built for handling different hardware provided all axis blocks implement this interface, any axis block can be replaced with any other axis block and the higher-level program requires no changes. Want to change a Stepper to a

Servo? Change one assignment statement or add an IF condition.

4.2 Extending Components

One “simple” option to handle the challenge of interchangeable Components is to build one block that can do “everything”. For the axis block example, a single block can be built that has various options. The options would then indicate what hardware the block is to be connected to and what features it has. It could have options for steppers, options for different servos and options for different functions.

Great everything is in one block that can do “everything”. The problem is this block is never finished. It is continually being updated and its interface is continually changing and there’s so many options no one understands it. For Example, A new very low-cost version of stepper motor was sold as an option for an existing machine. It requires a digital output to enable it, and a digital output to reset it. “Cheap Digital Stepper” option is now added to the block, the block is given 2 more output variables, and the “Axis Type” which is likely an enumeration is extended for the new Cheap Digital Stepper type. This block will require a lot of documentation. When an instance of this block is created, it has every variable of every possible axis type included. This causes confusion. The same for if functionality is to be added. “Why are these inputs and outputs not linked, oh those are only there in the case Axis X was a cheap stepper motor we used one time 6 years ago”. When a new type of Cam Table is developed, this functionality must be added into the “do everything” axis block. Now all blocks even ones that do not use the new cam table function will have to contain large cam tables. If no one is taking care of ensuring this block is in a library there will end up being several partial complete versions of the block. One version might allow stepper control but not the new cam table. One might allow for a new cam table but not for a servo on fieldbus X.

Then end result no one uses the “does everything” block because it’s too complicated.

4.3 Start Small and Extend to Add Features

Many blocks start with very high goals. For this project a new Servo block will be built, it will be able to do point to point motion, flying shear functionality, and handle multiple cam table switching. Every Axis in every project will use this block and all axes will have all possibilities.

Maybe 2 of the axes in a project will need Cam Table switching. Someone is going to take a significant amount of time building this block. Most axis will likely only need basic point to point motion, but this all this functionality needs to be programmed, tested and debugged before anyone can use it.

A better solution: Build a basic block for point-to-point motion. This block has one function, point to point. This block is great for any Equipment Module that needs point to point motion and doesn’t require any information about a master axis. This block is quick to build, test and debug and covers the basics, enable the axis to reset and move point-to-point, simple to build, simple to use.

Axes that require synchronization will also need point-to-point motion. The point-to-point block can be extended. The extended block contains everything the original has, it has an axis reference and point to point motion, Components that have already been tested and work. For synchronization it needs the master axis information. When the block is extended a master axis reference is added, and basic camming/gearing can be added. If an error is found is ever found in the point-to-point block, fixing it in the point-to-point block automatically fixes it in the

extended Synchronization block. If a “Shortest way” option is added to the point-to-point block, it automatically appears in the Synchronisation block. The programmer of the extended block can focus on the extension knowing the rest already works.

This idea of small blocks and extension, results in the most code re-use. The base axis from one project is now “in demand” for the next project because the block has been significantly tested and does not include things that are not necessary. For point to point, it is perfect. When the complexity is necessary, the Extended block is used. However, when the complexity is not required the simple base block can be used without the “clutter” of unused functionality.

If the basic block was chosen, and later it’s determined the extended block is necessary, no problem, switch the basic block for the extended version. The Extended block still contains the basic block and all the existing code that used the basic block still works for the extended version.

This is inheritance and this is probably the single most powerful feature available in TwinCAT 3. Extending blocks is always available, even if Interface Data types are not used. Blocks in libraries can also be extended. The extended block cannot break the base block, it can only add to it and only the new functionality requires testing and debugging. In the event the new block needs to change how a basic function works, the methods can be extended or overridden. If the new block has new hardware and the reset must be different? No problem, override the base block’s Reset method to include the code for handling the new hardware.

5 Define the Functionality

Now it is known which Equipment Modules and Components will exist and what the interfaces will look like, time to start writing code.

Not so fast! Before writing a single line of code, either on paper or in a spreadsheet determine what functions the Machine each Equipment Module must be able to do in each mode of operation and what happens in the case of an error level of error of each Module. Modules always has more than one mode of operation, it is going to require a manual mode for jogging axis after a jam, it may have a clearing mode to clear out partially completed products, and it will very likely have a dry cycle mode for operating without product. Everyone (management) is always concerned about Automatic/Producing mode; this is theoretically, where the machine makes money. This mode is important, but it is generally "easy" because everything is running. All the other modes, functions, and features take time to build, and they need to planned in at start. Adding core functions later without planning has the most risk of “breaking” Components already completed.

Create a Document/Spreadsheet that does the following:

1) Defines all modes of operation of the machine and each Equipment Module. It is common that different Equipment Modules will be in different operation modes at different times. Putting a single Equipment Module into manual while the machine is in production is fairly standard.
2) Determines what each Module must do in each mode
3) Contains a list of what Errors can occur each Module (this list will be added to as construction progresses)

Creating a table for each Equipment and Component and laying out the functionality in each mode does two things.

1) It provides a complete scope of work of how many functions really must be implemented
2) It clarifies the operation of the entire machine.

If the machine is built without a full plan, after the machine produces the first product comes the statement, "Great that works. Now during product change over before we load a new material, these two axes have to decouple and this one has to stop in order to release tension here." Whoa, wait, what?!? If I had known that I would have done all this slightly differently to accommodate that. Patching in unplanned functionality later typically causes the most unintended consequences. After patching in the new function, the next problem report is, "Uh, somehow while running the machine the operator started the load process, that decoupled the axes, released the tension, the film wrapped around the dancer and bent the arm, we need a new part machine will be down for a week".

Laying out the functionality for each Module takes time, it may take a couple of days or a week or more depending on the complexity of the machine. However, when completed the project timeline will be more accurate, it will save days/weeks of effort at the end of the project when time is most precious. The phrase "If we knew that, we would have done this instead" is a huge problem before running off. Now comes the debate, re-do a major Component, which will prevent the machine from running until it is completed, or try to somehow shoehorn in the extra functionality. Usually, the shoehorn method is employed until it's finally determined that the block must be re-built, and even more time is lost.

Producing mode is important but **the customer only complains when the machine is not producing.** The end user paid for production mode, every minute the machine is not in production mode is costing money, getting the machine back into production quickly is where the money is.

There will be a sequence to start up the machine and there will be different functionality in different modes of operation. Commissioning, Maintenance, unsticking things after a jam, there's a lot of functionality needed that is not "automatic production". No problem, build a Mode handling and a state machine to help ensure functions from Manual mode are not accidentally triggered in Automatic mode. There are now two options, come up with your own Mode and State sequencing system, or follow a standard. For those who build/have their own Mode/State handling system, that is perfectly acceptable. Doing it your own way comes with the task of building/debugging your system, documenting your mode/state handling system, and teaching your system to other programmers, service people, and the end user. Having the ability to say this machine conforms to ISA TR88.00.02 makes things a lot simpler. The TR88.00.02 Implementation guide from OMAC describes the implementation of TR88 very well.

6 Build a Small Machine

A Vertical Form Fill Seal Machine
There's tons of vertical form fill seal machines and documents out there. Wikipedia does a poor job of describing one. https://en.wikipedia.org/wiki/Vertical_form_fill_sealing_machine but doing a search for VFFS machine returns hundreds of machines.

For simplicity, this machine will be intermittent and have 3 Components, Unwind/Tensioning, Pull Wheels, Sealing Bars. These items logically become Equipment Modules. Now to figure out what the Components will be. Unwind has an axis and a dancer (sensor) to adjust speed and tension. Pull wheels have 2 axes coupled together. Sealer has a temperature-controlled bar and an axis to move it in and out. The Equipment Modules are nearly small enough to become control modules but looking through there are 4 axes. If the Unwind, Pull Wheels and Sealer Axis are all Components, each one is going to have to have code to deal with and resetting and enabling axis and fault messaging for each axis, sure copy and paste works. Followed by the copy and paste errors along with fixing 3 but forgetting 1 and spending an hour to figure out what is different. Google copy and paste programming there will be thousands of hits as to why this is bad i.e. expensive. Much better to deal with it once, do it right and make

the axis a Component and re-use it 4 times. Then should the unwind ever need to change the Axis from an AC Motor to a Stepper or a Servo, the Axis Component can be switched out without effecting the Unwind Equipment Module. Breaking down the machine in Equipment and Components, we have the following.

1) Unwind Equipment Module
    a. Unwind Axis (instance of Axis) Component
    b. Dancer Sensor Component
2) Pull wheel Equipment Module
    a. Pull Wheel Axis (instance of Axis) Component
3) Sealer Equipment Module
    a. Seal Bar Axis (instance of Axis) Component
    b. Seal Bar Temperature Component
4) Safety System While not a module per say, it needs to be monitored by the Machine to determine when to abort devices.

The Seal Bar could be its own Equipment Module. Generally, the sealer is a single unit, bar and blade. The sealing bar is not easily mechanically separated from the cutting mechanism, thus one Equipment Module. It is however perfectly valid to implement the Seal bar as its own Equipment Module in which case we would have the following breakdown.

1) Unwind (Equipment Module)
    a. Unwind (instance of Axis) (Component)
    b. Dancer Sensor (Component)
2) Pull wheel (Equipment Module)
    a. Pull Wheel Axis (Component)
3) Cutter (Equipment Module)
    a. Cutter Axis (Component)
4) Seal Bar (Equipment Module)
    a. Seal Bar Temperature Control (Component)
5) Safety System (Special Component)

6.1.1 Safety Circuit

The safety circuit it should be monitored. If the safety logic is as modular as the machine is, then each module should monitor it's own safety module and report it back to the machine. If there's only one safety logic controller then it belongs to the machine level.

It should be possible to know not just the status of every single safety input, output, and logic Component. Most safety systems will be able to provide all the diagnostic information as to the status of every safety device, E-Stop buttons, Light Curtains, Doors, etc. It should be immediately available to the operator which safety device was tripped to stop the system, or what is preventing the system from starting.

The Safety Circuit could be a module, there's no real reason it can't be. For error handling it will need an ID or some way of providing an alarm that indicates it is the safety system that has triggered the alarm.

6.2 Components

Component functionality is generally always the same regardless of machine mode. Jog functionality won't likely be called in Automatic mode, but it has the capability. Component can have state engines but will not follow the TR88 method.
What can each block do and what alarms will they have?

6.2.1.1 Axis

For the Axis they have 2 possible commands, Move to Position and Move at Velocity, and accept the values Target Position, Velocity, Acceleration, Deceleration and perhaps Jerk. The axis should return status Current Position, and Current Velocity, In Position, and Time for last move.
Faults: Axis Hardware Fault go to aborted (Level 1 clear required) invalid commands go to stopped (level 2 reset required). There's not much else that can go wrong with the axis, the command could be is invalid (too fast), the hardware can have a fault and can't run, or the axis cannot physically make the move and creates a position fault.

6.2.1.2 Temperature control

The Temperature Control should have the following parameters a set point, alarm levels for High-High, High, Low, and Low-Low. It must accept a command to actively control its temperature or to run an Auto Tuning Sequence. The Temperature Control should return its current temperature and status as to whether it is above High High, above High, between low and high (ie operating properly), below Low, and below Low-Low.
Faults: Control loop faults to aborted (level 1 clear required) invalid commands to stopped (level 2 reset required)

With the faults we can immediately see that different faults result in different actions. When faults are assigned levels, it becomes very easy to determine the higher levels reaction.

6.2.1.3 Dancer

The Dancer will have a sensor and values for scaling. The Dancer will scale the sensor value and output an angle in degrees.
Faults:
Sensor Fault go to Aborted (level 1 requires clear)
Invalid parameters to stopped (level 2 requires reset)

6.2.2 Table of all Control Module Functions

State	Axis	Dancer Sensor	Temperature Control
Aborting	Emergency Ramp	Provide Scaled Value	Output Off
Aborted	Power Off	Provide Scaled Value	Output Off
Stopping	Halt	Provide Scaled Value	Output Off
Stopped	Power Off	Provide Scaled Value	Output Off
Resetting	Rest then Enable	Implement/Adjust Scaling parameters	Reset Control Loop
Idle	Enabled and Ready	Provide Scaled Value	Output Off
Starting	Perform Move Command according to parameters	Provide Scaled Value	Accept Command
Execute	Moving as per Parameters	Provide Scaled Value	Auto Tune or Maintain Temp

Complete	Move Complete	Provide Scaled Value	Auto Tune Complete

Temperature control could be a tricky one, it could be said, OK, start the Temp Control and when it is within range, it is complete and maintain temperature, but what happens when it's out of range? Does it go to stopped or aborted? If it stops or aborts, then it has to go back into execute somehow which will require a command. Therefor, for Temp Control, start either triggers the Auto Tune command or activates the temperature control loop. The feedback parameters will provide the current temperature and if it is within the Set point range as defined by the parameters provided. If the start command was given to Autotune, then execute runs the auto tune process and when complete goes to the complete state.

Should the temperature control loop be active in Idle?? No. If the temperature controller is in idle, nothing should be done. What if the machine is in Idle?? If the machine is in Idle the temperature controller should be active, so shouldn't the temperature control also be active in Idle?? Again No, if the Temperature Control should be active when the machine is in Idle, then the Temperature controller should be in the Execute State while the machine is in Idle. This is perfectly fine. The state of the module is not dependent upon the state of the machine.

6.3 Equipment Modules

Now for the Equipment Modules, these are going to be a little more interesting as these will likely have different modes of operation. This machine is going to stay "simple" and motions are intermittent.
It is important to define the Components first because the Equipment Module functionality is going to be determined by what state commands it issues to the control modules and what each control module does in which state. It is very likely that Components will be re-worked while doing the Equipment Module design.

6.3.1.1 Unwind

When requested the unwind will deliver 1 bag length of material. As material is unrolled from the unwind, the diameter will decrease and the feedback from the dancer will adjust the number of rotations for the unwind roll.
In Manual Mode the unwind axis can be jogged for loading and aligning
Faults: Axis Fault go to aborted (level 1 requires clear) Sensor Fault to aborted (level 1 required clear)
Material Low Level 3/4 (machine can continue) Material Empty: Level 2 (no fault but cannot accept a new command)

6.3.1.2 Pull Wheels

When requested the pull wheels will pull 1 gab length of material. The ratio of pull wheel axis rotations to material length is constant. In manual mode the pull wheel axis can be jogged for alignment.
Axis Fault go to abort

6.3.1.3 Sealer

When enabled, the sealer will maintain temperature of the seal bar to the setpoint parameter. When requested the Sealer will move to the close position, and dwell for a specified seal time, then return to the open cycle at which point a cycle is complete. In Manual Mode the Seal Axis can be jogged and the Temperature control loop can be tuned.

Faults:
Temperature control above High High abort (fire hazard)
Temperature High, Low, Low Low, warning
Axis fault abort, requires clear

6.3.2 Equipment Modules Production

Production Mode is easy. Equipment modules could use the same state model as the control modules

State	Unwind	Pull Wheels	Sealer
Aborting	Abort Axis Abort Dancer	Abort Axis	Abort Seal Axis Abort Temp Control
Aborted	Axis Aborted	Axis Aborted	Axis Aborted Temp Control Aborted
Clearing	Clear Axis If Aborted	Clear Axis If Aborted	Clear Axis If Aborted Clear Temp Control if Aborted
Stopping	Wait for Axis Complete then Stop Axis	Wait for Axis Complete Then Stop Axis	Wait for Axis Complete then Stop Axis and Stop Seal Bar
Stopped	Axis In Stopped	Axis In Stopped	Axis in Stopped, Temp Control in Stopped
Resetting	Reset Axis and Dancer, accepts Recipe Parameters	Reset Axis	Reset Axis and Temp Control, Start Temp Control
Idle	Axis in Idle, Dancer In Idle	Axis in Idle	Axis in Idle Temp Control in Execute
Starting	Start Axis for 1 Bag Length	Start Axis for 1 Bag Length	Start Axis to CutPosition
Execute	Axis in Execute	Axis In Execute Execute	Wait for Axis to complete at Cut Position, Dwell, Start Axis to Open Position wait for Axis Complete
Completing	Not Used	Not Used	Not Used
Complete	Axis in Complete	Axis In Complete	Temp Control Execute Axis in Complete at Open Position
Holding	Not Used	Not Used	Not Used
Held	Not Used	Not Used	Not Used
Unholding	Not Used	Not Used	Not Used
Suspending	Not Used	Not Used	Not Used
Suspended	Not Used	Not Used	Not Used
Unsuspending	Not Used	Not Used	Not Used

Note the Seal system does not wait or abort if started and the temperature is not in range. If the Sealer is commanded to make a sequence, it makes a sequence, the bag should be marked as bad production, but it should be produced. The machine layer should be worried about issuing a command if the temperature is in spec though perhaps

6.3.3 Equipment Modules Maintenance

Maintenance Mode is going to allow full cycling of the Equipment Module but without material or with Components disabled. Unwind, Pull Wheel, Temperature Control and Seal Axis can all be disabled, Unwind can be run without dancer control, Seal Axis can be run without temperature control, Seal Bar can also be temperature controlled without the axis.

State	Unwind	Pull Wheels	Sealer
Aborting	Abort Axis, Abort Dancer	Abort Axis	Abort Axis Abort Seal
Aborted	Axis Aborted Dancer Aborted	Axis Aborted	Axis Aborted, Temp Control Aborted
Clearing	Clear Axis if Aborted and Enabled Clear Dancer If Aborted Enabled	Clear Axis If Aborted and Enabled	Clear Axis if Aborted and enabled, Clear Temp Control if Aborted and Enabled
Stopping	Wait for Axis Complete then Stop Axis and Stop Dancer if Enabled	Wait for Axis Complete then Stop Axis if Enabled	Wait for axis Complete in Open Position then Stop Axis if Enabled and Stop Temp Control if Enabled
Stopped	Axis and Dancer In Stopped if Enabled	Axis In Stopped if Enabled	Axis in Stopped if Enabled and Temp Control in Stopped if Enabled
Resetting	Implement Parameters, Reset Axis and Dancer if Enabled	Implement Parameters Reset Axis if Enabled	Implement Parameters, Reset Axis and Temp Control if Enabled, Start Temp Control if Enabled
Idle	Axis In Idle if Enabled	Axis In Idle if Enabled	Axis In Idle if Enabled, Temp Control in Execute if Enabled
Starting	Start Axis for 1 bag length if enabled Start Dancer	Start pull for 1 bag length if enabled	Axis to Close Position if enabled Temp Control in Execute if Enabled
Execute	Axis in Execute if Enabled, Dancer in Execute if Enabled	Exis in Execute if Enabled	Maintain Temp Control if Enabled, If Axis Enabled, wait for Axis to Complete at Close position, dwell, Start Axis to Open Position
Completing	not used	not used	not used
Complete	One bag length unwound and Axis Is Complete if Enabled	One Bag length Pulled and Axis Is Complete	Temp Control Maintained in Execute, Axis is at open position and Complete if Enabled
Holding	Not Used	Not Used	Not Used
Held	Not Used	Not Used	Not Used
Unholding	Not Used	Not Used	Not Used

6.3.4 Equipment Module Manual

In the manual mode all manual commands can be given to the modules and in Manual Mode, the Equipment Modules allow commands directly to their Subcomponents. Manual has very few states enough to get the module into a state where the Subcomponents can accept commands

directly. It is fully acceptable to put an Equipment Module in Manual mode and maintain the machine in production mode.

State	Unwind	Pull Wheels	Sealer
Aborting	Abort Axis, Abort Dancer	Abort Axis	Abort Axis, Abort Seal Aborted
Aborted	Axis Aborted Dancer Aborted	Axis Aborted	Axis Aborted, Temp Control Aborted
Clearing	Clear Axis and Dancer If Aborted and enabled	Clear Axis If Aborted and Enabled	Clear Axis if Eborted and enabled, Clear Temp Control if Aborted and Enabled
Stopping	Wait for Axis Complete then Stop Axis and Stop Dancer if Enabled	Wait for Axis Complete then Stop Axis if Enabled	Wait for axis Complete in Open Position then Stop Axis if Enabled and Stop Temp Control if Enabled
Stopped	Axis and Dancer In Stopped if Enabled	Axis In Stopped if Enabled	Axis in Stopped if Enabled and Temp Control in Stopped if Enabled
Resetting	Implement Parameters, Reset Axis and Dancer if Enabled	Implement Parameters Reset Axis if Enabled	Implement Parameters, Reset Axis and Temp Control if Enabled, Start Temp Control if Enabled
Idle	Axis In Idle if Enabled	Axis In Idle if Enabled	Axis In Idle if Enabled, Temp Control in Execute if Enabled
Starting	Start Axis for 1 bag length if enabled Start Dancer	Start pull for 1 bag length if enabled	Axis to Close Position if enabled Temp Control in Execute if Enabled
Execute	Axis in Execute if Enabled, Dancer in Execute if Enabled	Exis in Execute if Enabled	Maintain Temp Control if Enabled, If Axis Enabled, wait for Axis to Complete at Close position, dwell, Start Axis to Open Position
Completing	not used	not used	not used
Complete	One bag length unwound and Axis Is Complete if Enabled	One Bag length Pulled and Axis Is Complete	Temp Control Maintained in Execute, Axis is at open position and Complete if Enabled
Holding	Not Used	Not Used	Not Used
Held	Not Used	Not Used	Not Used
Unholding	Not Used	Not Used	Not Used

6.4 Machine

The full machine is just coordinating the states of the Equipment Modules. First figure out what the machine is going to do in each state and think about which states will allow Equipment

Modules to switch modes, and in which state the machine is going to allow mode changes

6.4.1 Machine Production Mode

State	Machine
Aborting	Unwind Abort Pull Wheel Abort Sealer Abort
Aborted	Unwind Aborted Pull Wheel Aborted Sealer Aborted
Clearing	Clear Unwind Clear Pull Wheel Clear Sealer
Stopping	Stop Unwind Stop Pull Wheel Stop Sealer
Stopped	Unwind Stopped Pull Wheel Stopped Sealer Stopped
Resetting	Implement Parameters, Reset Unwind Reset Pull Wheel Reset Sealer
Idle	Unwind Idle Pull Wheel Idle Sealer Idle
Starting	not used but could start the first cycle
Execute	If Sealer not at Temp, go to Trigger Hold, Otherwise Start Unwind, Start Pull Wheel, wait for Unwind Complete, Pull Wheel Complete, start Sealer and Reset Unwind and Reset Pull Wheel, wait for sealer complete and Unwind Idle, Sealer Idle, then repeat, Unwind and Pull then Seal until product count is reached then Complete
Completing	Run Unwind Pull Wheels and Sealer until Sealer is complete
Complete	Waiting on User
Holding	Run Unwind, Pull Wheels and Sealer to a complete Cycle
Held	Waiting on Operator or Temperature Control Operator allowed to Switch Equipment Modules to Manual to remdy problem
Unholding	With all Modules back in Production, reset each Module and go to Execute (start could be issued to Pull and Unwind
Suspending	Run Unwind Pull Wheels and Sealer to complete a Cycle
Suspended	Wait on upstream down stream
Unsuspending	When free to continue go back to Execute

6.4.2 Machine Maintenance Mode

Effectively the same as production mode but now allow Equipment Modules to be configured for running without material take the above table and add "If enabled" to everything.

State	Machine
Aborting	Unwind Abort Pull Wheel Abort Sealer Abort
Aborted	Unwind Aborted Pull Wheel Aborted Sealer Aborted
Clearing	Clear Unwind Clear Pull Wheel Clear Sealer
Stopping	Stop Unwind Stop Pull Wheel Stop Sealer
Stopped	Unwind Stopped Pull Wheel Stopped Sealer Stopped
Resetting	Implement Parameters, Reset Unwind Reset Pull Wheel Reset Sealer
Idle	Unwind Idle Pull Wheel Idle Sealer Idle
Starting	not used but could start the first cycle
Execute	If Sealer not at Temp, go to Trigger Hold, Otherwise Start Unwind, Start Pull Wheel, wait for Unwind Complete, Pull Wheel Complete, start Sealer and Reset Unwind and Reset Pull Wheel, wait for sealer complete and Unwind Idle, Sealer Idle, then repeat, Unwind and Pull then Seal until product count is reached then Complete
Completing	Run Unwind Pull Wheels and Sealer until Sealer is complete

Complete	Waiting on User
Holding	Run Unwind, Pull Wheels and Sealer to a complete Cycle
Held	Waiting on Operator or Temperature Control Operator allowed to Switch Equpiment Modules to Manual to remdy problem
Unholding	With all Modules back in Production, reset each Module and go to Execute (start could be issued to Pull and Unwind

Fault/warning handling
Hold: Unwind material empty will cause a hold. Sealer Temperature out of range will cause a hold.
Stop: Stop button causes a stop, invalid commands to an Equipment Module will cause a stop
Abort: Any module that aborts will abort the machine
Abort: Any interruption in the Safety System will immediately abort the machine.

6.4.3 Machine Manual Mode

Manual Mode should put all Equipment Modules in Manual Mode

State	Machine
--	
Aborting	Unwind Abort Pull Wheel Abort Sealer Abort
Aborted	Unwind Aborted Pull Wheel Aborted Sealer Aborted
Clearing	Clear Unwind Clear Pull Wheel Clear Sealer
Stopping	Stop Unwind Stop Pull Wheel Stop Sealer
Stopped	Unwind Stopped Pull Wheel Stopped Sealer Stopped
Resetting	Implement Parameters, Reset Unwind Reset Pull Wheel Reset Sealer
Idle	Unwind Idle Pull Wheel Idle Sealer Idle
Starting	not used
Execute	Allow Individual Control of each Module

7 Implementation

Now finally it is time to construct the blocks. Every item has been defined and the machine functionality is clear. There will be adaptations as things are implemented but there is now enough present to lay out the individual function blocks and the exact interfaces each block will have with others.

With the blocks laid out, this has also laid out the effort and the project plan for what must be done. Yes, missed things will still occasionally be found, but the number and complexity of tasks are known. Some tasks will not be as clear as others. This is a clear indicator more effort will be required. E.g., Unwind, there is a relationship between the dancer and the unwind axis that could have various implementations. Marking it as a “development required task' will highlight that it will consume extra time and might need some revisions that it is a higher risk task and should be addressed as early as possible.

Common core Components can be identified in the Modules. These core Components should be built once, built right, and re-used. The more often a Component is used, the more debugging time it gets, the more robust it becomes the fewer problems it has and soon it “just works”. Building the same thing twice or copy and pasting doubles the chances of errors. There will be more total time debugging but the debugging is split amongst the copies of the same Component. Copy and paste errors eat massive amounts of time with no functionality gain other than two functional copies of the same code, which is what should have happened in the in the first place.

To start the project, we must first investigate the library of code that is being supplied for creating machines.

7.1 Library code

The SPT Group has created a library of base modules and Components to build an application. The library code consists of the following items:

1. SPT Base Types
2. SPT Component Base
3. SPT Components
4. SPT Event Logger
5. SPT PackML Base
6. SPT Utilities

7.1.1 SPT Base Types

At the core of the library are two function blocks FB_BaseFB and FB_CyclicFB.

7.1.1.1 FB_BaseFB

This block implements the interface I_BaseFB.

Type	Name
Properties	Busy, Error, ErrorID

The FB_CyclicFB will extend this block. This block uses ABSTRACT keyword which means that this Function block cannot have an instance of it created. Instead, other blocks will extend from this block and those can have instances made.

7.1.1.2 FB_CyclicFB

This block is an extension of FB_BaseFB and implements I_CyclicFB. This block is also ABSTRACT.

Type	Name
Properties	InitComplete
Methods	CyclicLogic

Many base modules discussed here will be an extension from this block.

7.1.2 SPT Component Base

This block is an extension of FB_CyclicFB and is used as the basic framework for all Components. This block implements the interface I_ComponentBase.

Type	Name
Properties	CurrentAlarmSeverity, InSimulation, Name, ParentResponseDefinitions
Methods	AllowHMIControl, BlockHMIControl. Reset

The block also has methods that can be overridden for extended functionality:

Type	Name
Overridable	CreateEvents, HMICommunication, Monitoring

These methods are PROTECTED which means that they can only be called from inside this block or inside any block that extends this one.

This block is also ABSTRACT.

At this low level of Component setup, the access with an HMI is taken into consideration. Also, in this library a data structure for the communications to the HMI is started:

HMI Type	Variable	Data Type
Config	Name	STRING
Command	SimulatedOperation	BOOL
Command	StandardOperation	BOOL
Command	Reset	BOOL
Status	InSimulation	BOOL
Status	Busy	BOOL
Status	Error	BOOL
Status	ErrorID	UDINT
Status	HMIControlAvailable	BOOL

7.1.3 SPT Components

This library contains 3 very common Components used in machine building, FB_Component_BasicAxis, FB_Component_BasicSlaveAxis and FB_DigitalSensorBase.

7.1.3.1 FB_Component_BasicAxis

This block extends from FB_ComponentBase and implements the Interface I_BasicAxis with the following methods and properties as shown in the UML Diagram

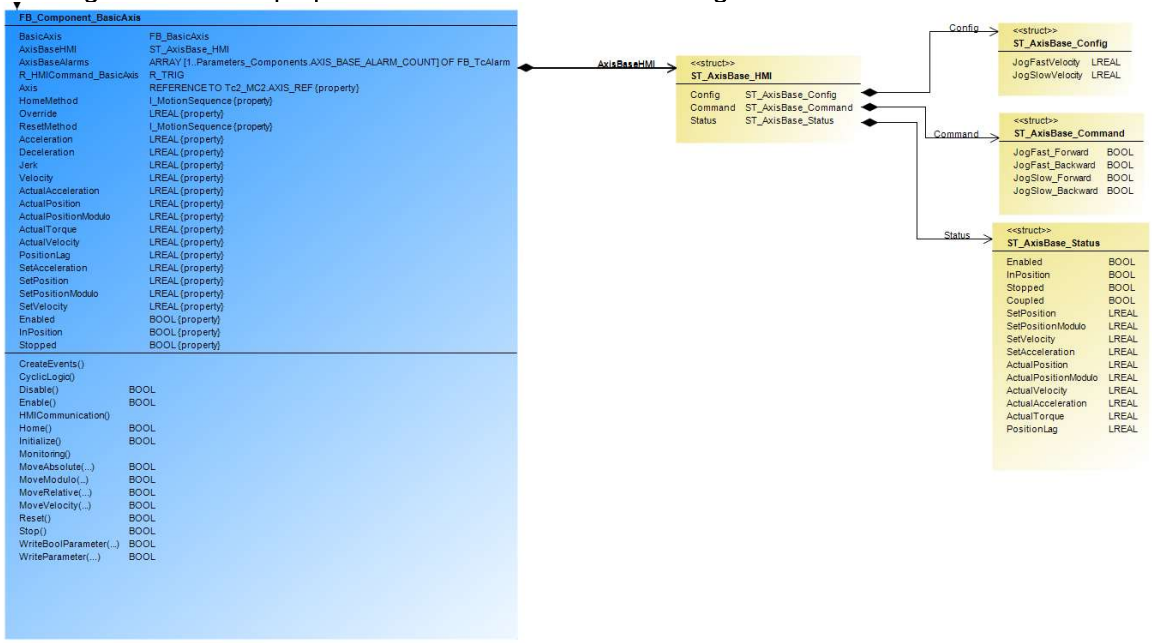


Figure 6 Component Basic Axis

7.1.3.2 FB_BasicAxis

This is the UML diagram for the FB_BasicAxis which has an instance in the above block.

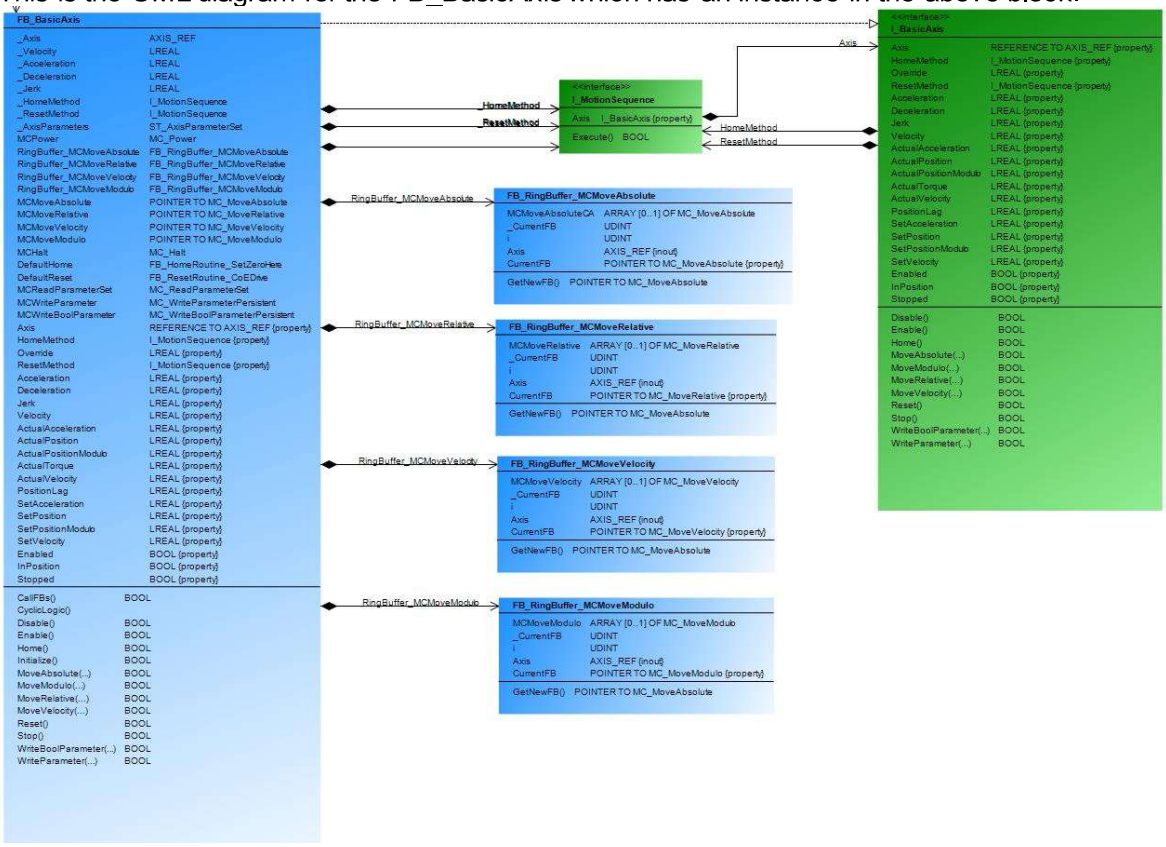


Figure 7 Basic Axis

7.1.3.3 FB_Component_BasicSlaveAxis

This block has the same functionality as FB_Component_BasicAxis but adds the ability for the axis to Gear to a master axis.

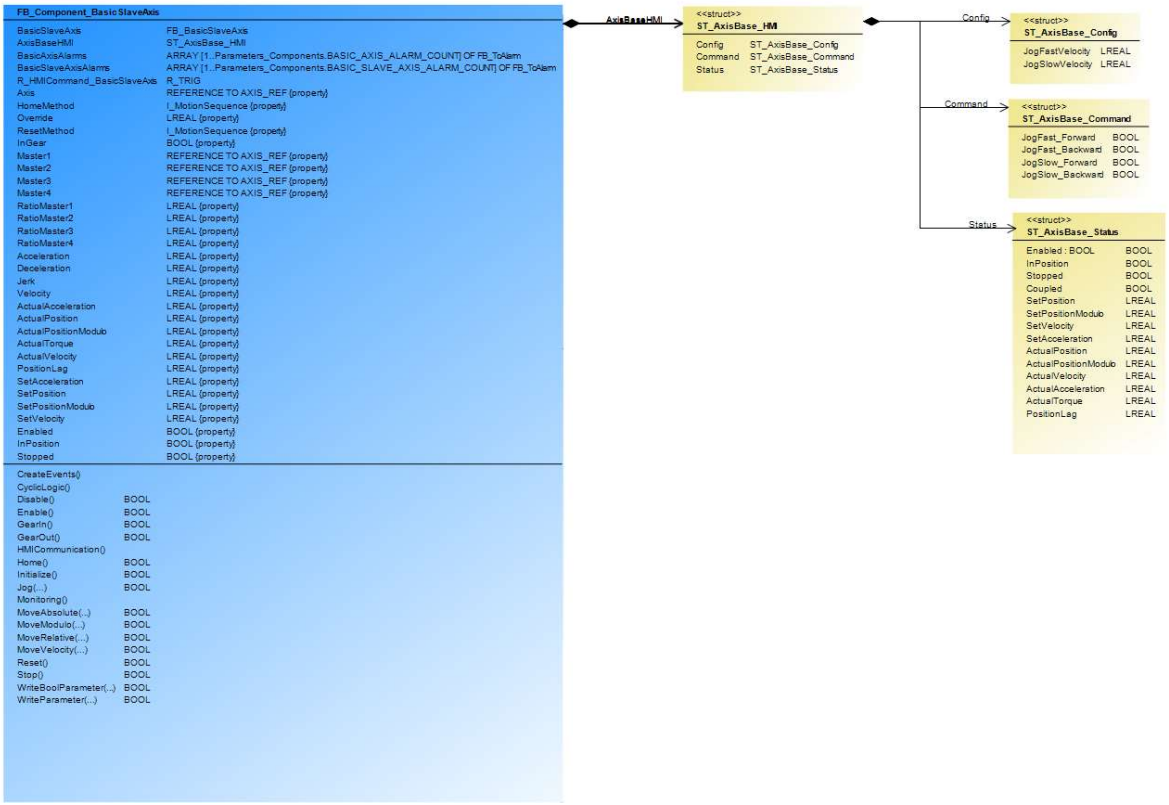


Figure 8 Component Basic Slave Axis

7.1.3.4 FB_BasicSlaveAxis

This block extends the FB_BasicAxis and, it adds the ability to gear to a master. The interface I_BasicSlaveAxis extends the I_BasicAxis.

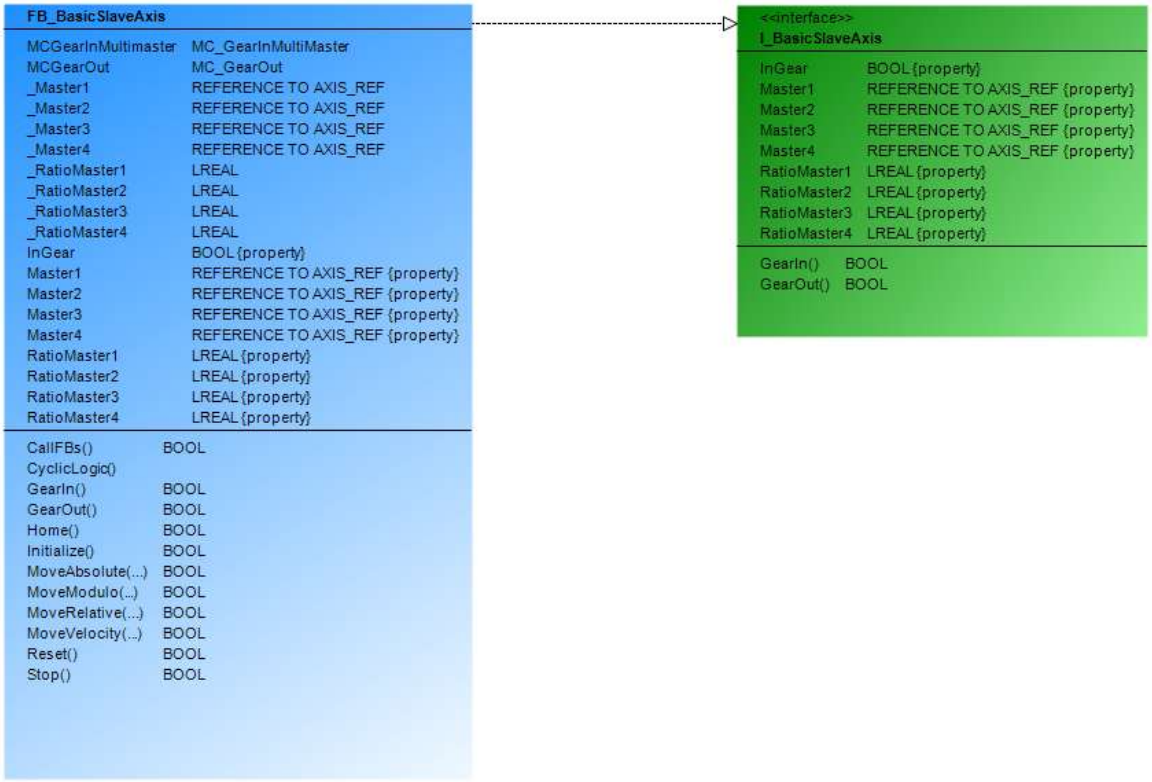


Figure 9 Basic Slave Axis

7.1.3.5 FB_DigitalSensor

This block extends the FB_ComponentBase and implements the I_DigitalSensorBase.

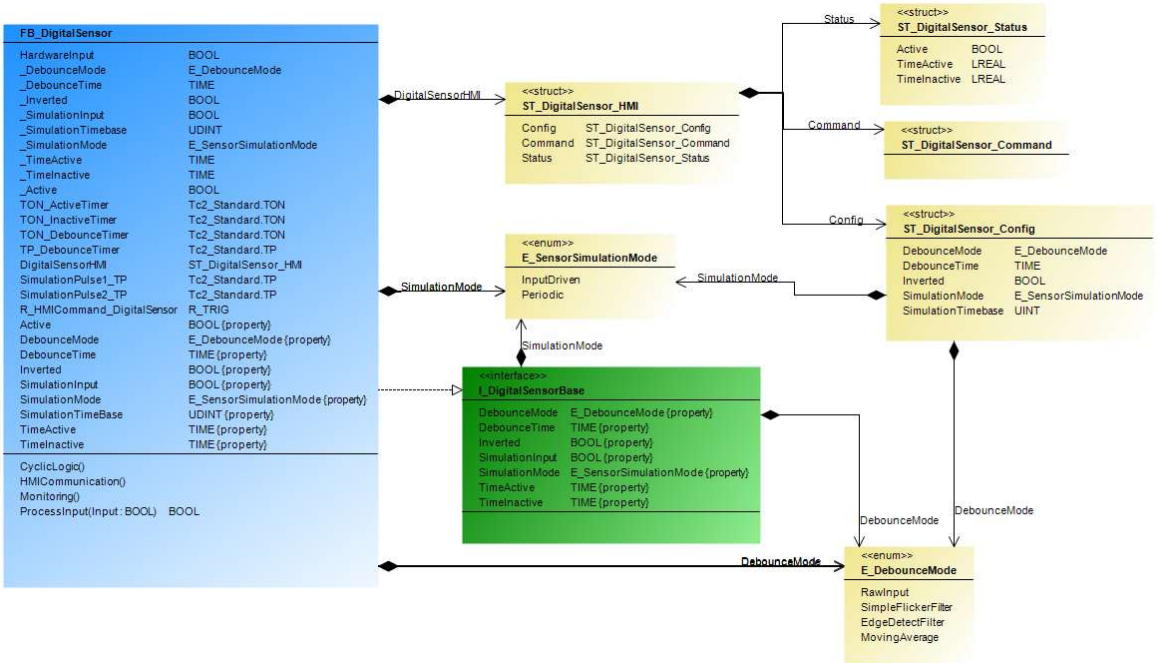


Figure 10 Digital Sensor

As seen in the properties, this digital sensor Component can be simulated.

7.1.4 SPT Event Logger

This library is a group of functions to support the TwinCAT Event Logger using FB_TcAlarms. An Excel spreadsheet and add-in are used to create the new classes for the application. The libraries have these alarms created already where the application will need to create new ones.

Function	Description
F_ClearAllEventsInClass	Clears all event function blocks for entire event class
F_CreateAllEventsInClass	Initializes event function blocks for entire event class using auto-generated source info
F_GetMaxSeverityRaised	Compares a collection of alarms to current alarm severity and returns the maximum severity of any raised alarm
F_RaiseAlarm	Raises an alarm and sets fault info to variables which are passed in
F_RaiseAlarmAndSetFlags	Raises an alarm and sets fault info to variables which are passed in and sets the flags
F_RaiseAlarmWithStringParameters	Raises an alarm and passes string arguments as parameters for extra fault context/info
F_RaiseAlarmWithStringsAndFlags	Raises an alarm and passes string arguments as parameters for extra fault context/info and sets the flags

Figure 11 SPT Event Logger Table

7.1.5 SPT PackML Base

This library has the PackML Base module, ModeLogger and State logger function blocks.

7.1.5.1 FB_PackML_BaseModule

This block extends FB_CyclicFB and implements I_PackML_BaseModule and I_PackML_Control. It is an ABSTRACT block. This block will be the base for all Machine Modules and Equipment Modules.

Type(I_PackML_BaseModule)	Name
Properties	AncestorIDs, CurrentAlarmSeverity, HMIControlAvailable, LogModeChanges, LogStateChanges, ModuleID, ModuleName, ParentResponseDef, ParentResponse_Critical, ParentResponse_Error, ParnetResponse_Warning

Type(I_PackML_Control)	Name
Properties	CurrentMode, CurrentState, ModeCommand, StateCommand
Methods	ChangeMode, ChangeState

HMI Data Structure

HMI Type	Variable	Data Type
Config	Name	STRING
Config	ModuleID	UDINT
Config	LogStateChanges	BOOL
Config	LogModeChanges	BOOL
Config	ModeNames	ARRAY[0..31] OF STRING
Command	Mode	E_PMLUnitMode
Command	State	Tc3_PackML_V2.E_PMLCommand
Command	ActivateStateLog	BOOL
Command	DeactivateStateLog	BOOL
Command	ActivateModeLog	BOOL
Command	DeactivateModeLog	BOOL
Status	Busy	BOOL
Status	Error	BOOL
Status	ErrorID	UDINT
Status	Mode	E_PMLUnitMode
Status	State	Tc3_PackML_V2.E_PMLCommand
Status	HMIControlAvailable	BOOL

7.1.5.2 FB_PackML_ModeLogger

This block will monitor and log all mode changes. An instance of this block is in the FB_PackML_BaseModule and can be configured to work by setting LogModeChanges to true.

7.1.5.3 FB_PackML_StateLogger

This block will monitor and log all state changes. An instance of this block is in the FB_PackML_BaseModule and can be configured to work by setting LogStateChanges to true.

7.1.6 SPT Utilities

This library has some function blocks and functions to help make things easier.

7.1.6.1 FB_FIFO_ULINT_Array

This block is a general purpose First In First Out buffer for ULINT type(64-bit Unsigned Integer). The block takes a POINTER TO ULINT.

Type	Name
Properties	EntryCount, LastInValue,NextOutValue
Methods	AddEntry, RemoveEntry

7.1.6.2 FB_Tree_IndexBased

This block maintains a hierarchy of a group of parent/child relationships. The block implements I_Tree_IndexBased.

Type	Name
Methods	AddNodeAsChildByName

7.2 Start at the Top and work Down

At the start of the project, virtually no Equipment Modules will be ready for use. The task to implement the Machine cannot continue until the other Components are running, or the other Components are "simulated" enough to function. Coding simulation takes time from completing the block but may be required to allow another programming tasks to continue.

If interfaces are implemented that is not the case. The Machine Programmer can already start implementing code against the interfaces alone. The user of the blocks does not care how the blocks do their thing, they care about what it does, and because all the Equipment Modules use the same base, it is known what they do, how they do it, and the interface can be used immediately without having any blocks that implement it. Starting with the machine and working down we can find anything that might have been missed during the project planning.

7.2.1 Main

The main program will have the instance of the Machine Module. This is needed to start the instance of the Machine. As we program further, we will add a couple more Function blocks to the main program to support the connection to HMI and Production Monitoring.

Below is a UML diagram of the VFFS Demo's final main program

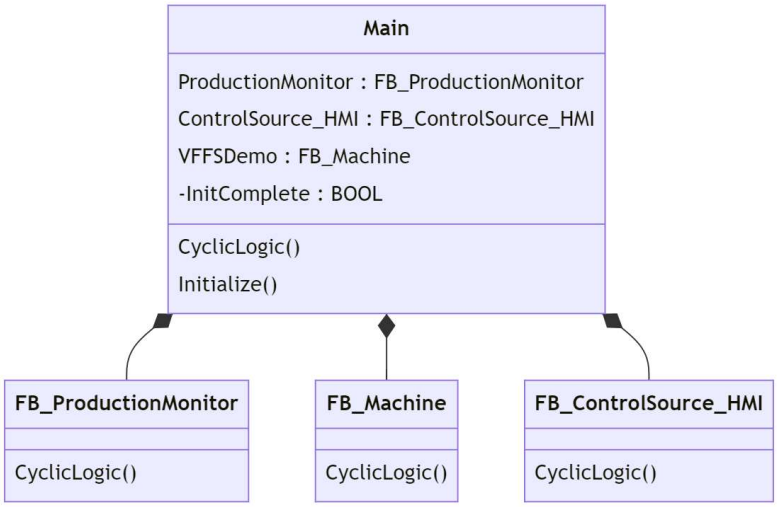


Figure 12 Main

7.2.2 Machine

FB_Machine contains several blocks that are not even started but this is not a problem. The programmer of the machine knows how the machine is supposed to behave and how to sequence the Equipment Modules so the coding of the machine can begin immediately. We know the machine requires the following items: Pull Wheels, Sealer, Unwind and Safety. The Equipment Modules first instantiated as the Base Block, then when the real blocks are ready, they can be switched to the proper new type.

The machine is going to have to be able to change states this is going to be driven by push buttons, HMI commands and Alarms in the Equipment Modules.

Within the FB_Machine it's now simply a matter of telling the base blocks what to do.
For example, the machine sequence might be:

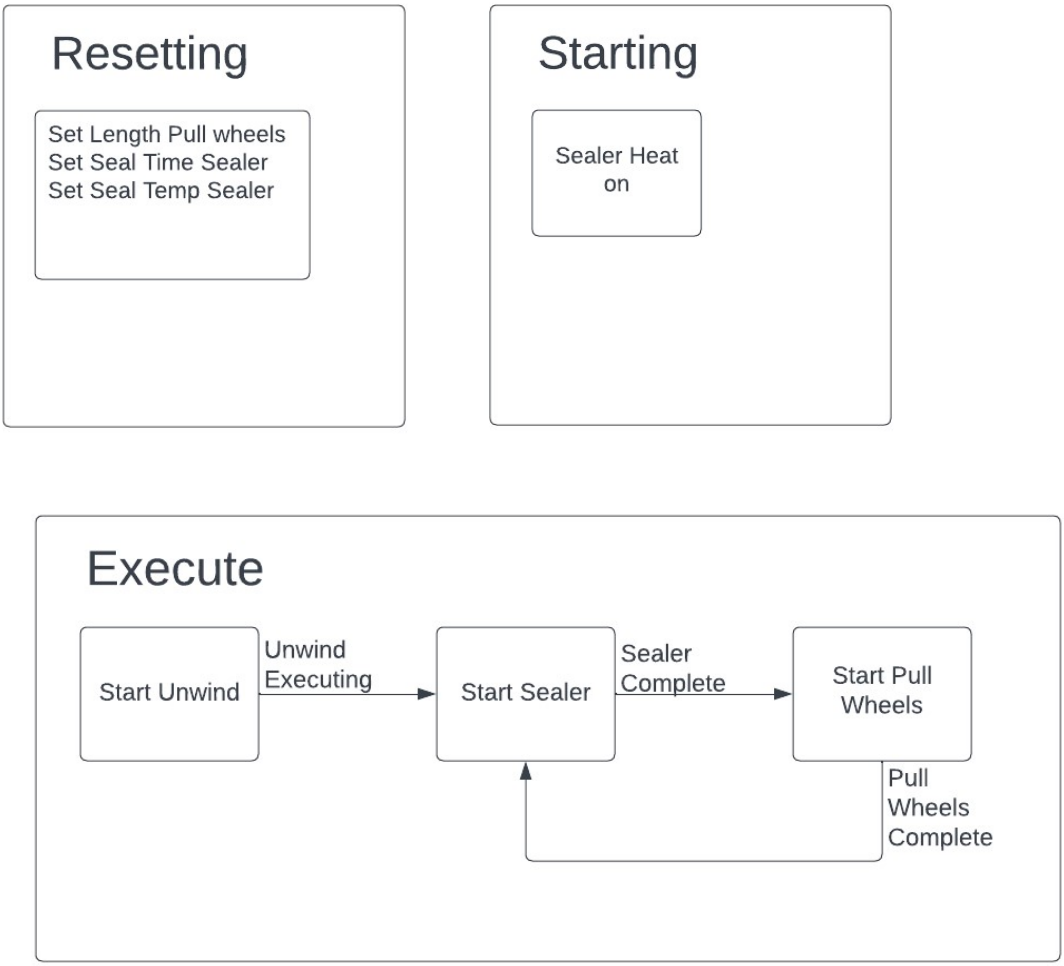
1) An Estop button is pressed and the machine must abort, FB_Machine sets itself and all Equipment Modules to Abort and now the FB_Machine and Equipment modules are in the Aborting State. There could very well be a sequence to this, Abort Modules X, Y and Z when they are aborted, Abort Module W.
2) Once all equipment Modules have reached the aborted state. The machine can switch from aborting to Aborted.
3) To restart operation the operator presses the start button and this will "Clear" the machine. FB_Machine will set all Equipment Modules to Clear, once each equipment Module has completed it's Clearing function the Module will transition to

the Stopped state. This state is where the safety is set and all components are enabled.

4) After all Modules are in the stopped state, the Machine can transition to the stopped state.
5) From Stopped pressing the Reset button switches the Machine and all Equipment Modules into the Resetting State.
6) Reset will put all the modules into the state needed to start the machine.
7) Once all Modules are reset and in the idle state, the machine can then switch to idle and is now ready for the start command to trigger starting and bring the machine into Execute and producing product.

The machine is simply sequencing the modules. It's not important how the modules complete their commands it is only important that they either complete their commands or throw an error, then the machine can determine how to react. Different modes are easy to build and to control.

This example code has the following logic in the given PackML states during Production Mode:



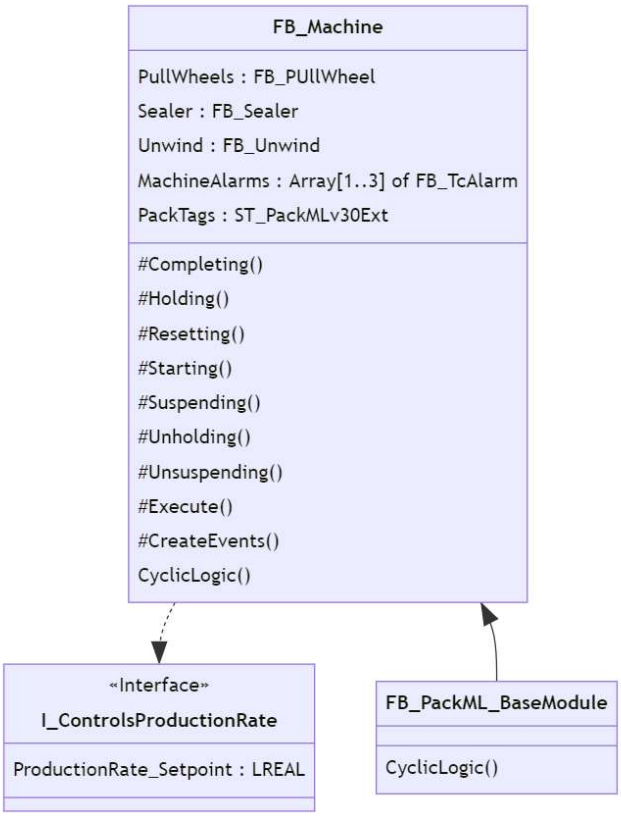
The machine module has no other specific code to run in the other states.

The FB_Machine also has to look at Alarms from each module and decide what it should do should it go to hold, cycle stop or abort? It depends on the module that threw the error and what "error level" the error was and how the machine is supposed to respond. If a Pull Wheel jams

this is a fault, but the Machine must decide how to handle this fault, should the machine abort or can it just do a stop.

This is where consistency in Error handling and reporting is important.

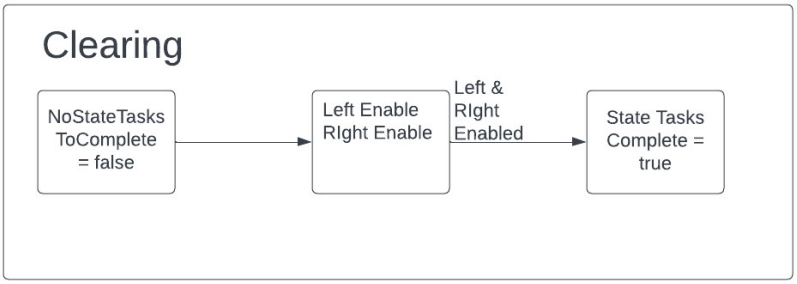
The final Machine Module will have the following structure. Below is the UML diagram of the Machine Module showing its links to the equipment modules:

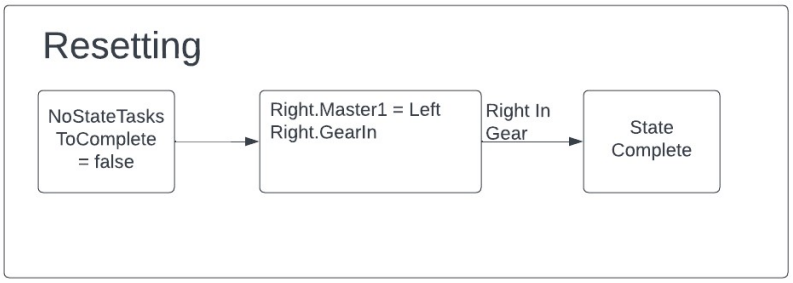


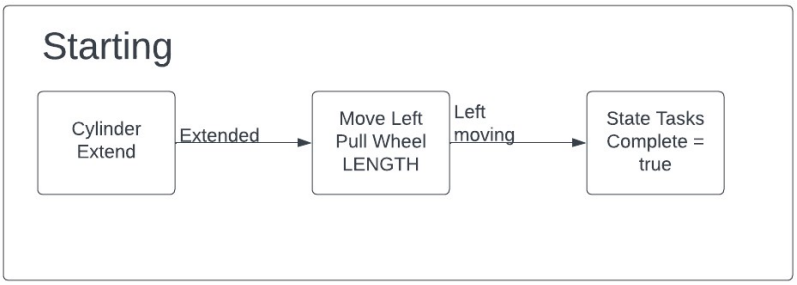
7.2.3 Pull Wheels

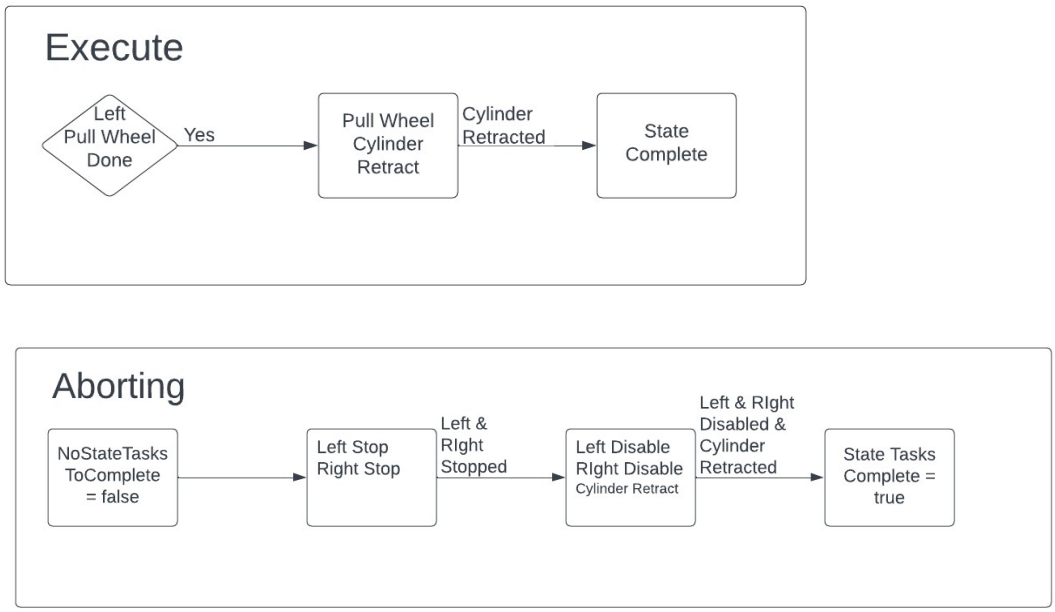
The pull wheels have 2 servo motors (Left, Right) and a cylinder as components in the Equipment Module.

The pull wheel module has the following logic in the following states and in Production mode:









UML Diagram of Pull Wheels

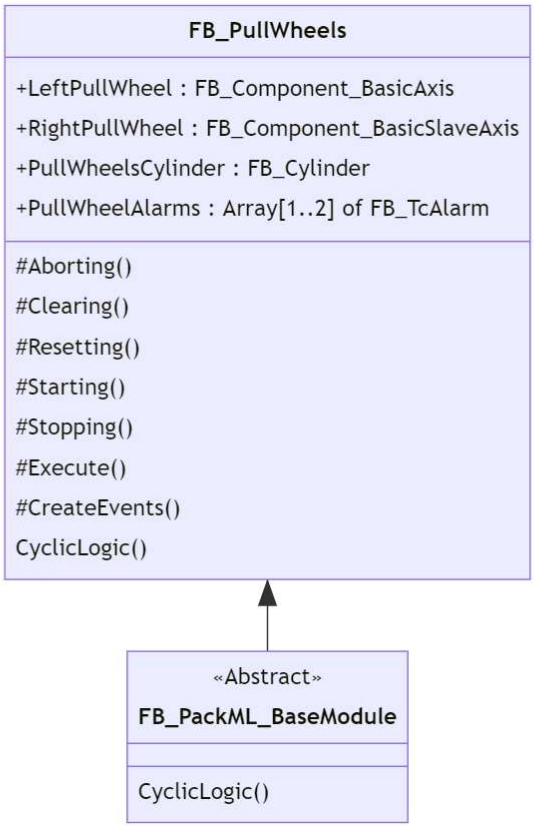
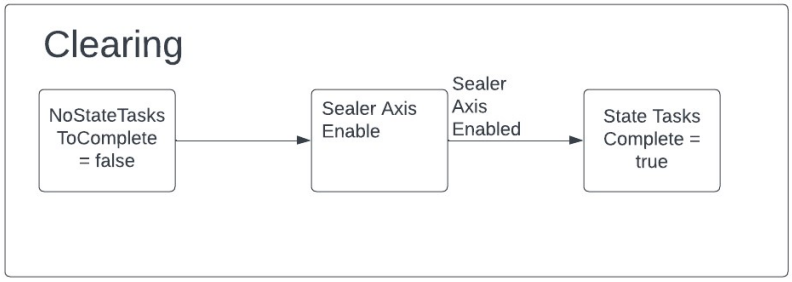


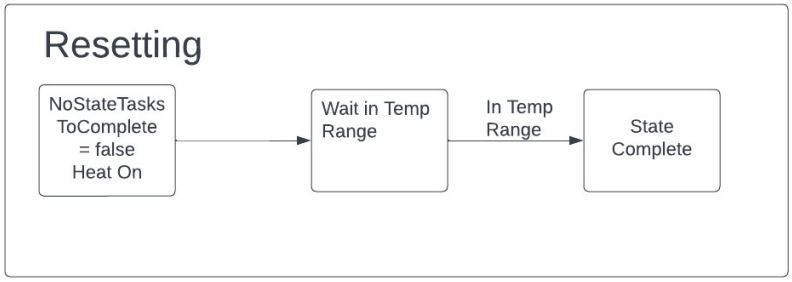
Figure 13 Pull Wheels

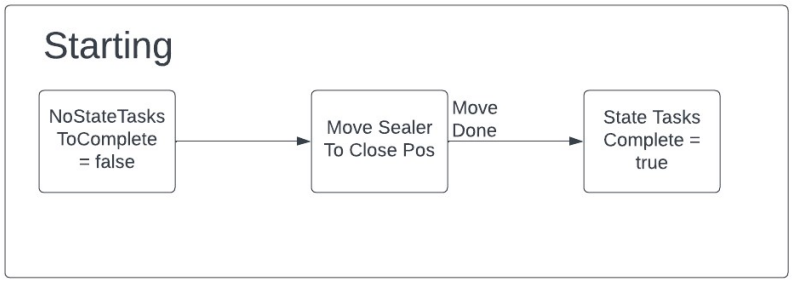
7.2.4 Sealer

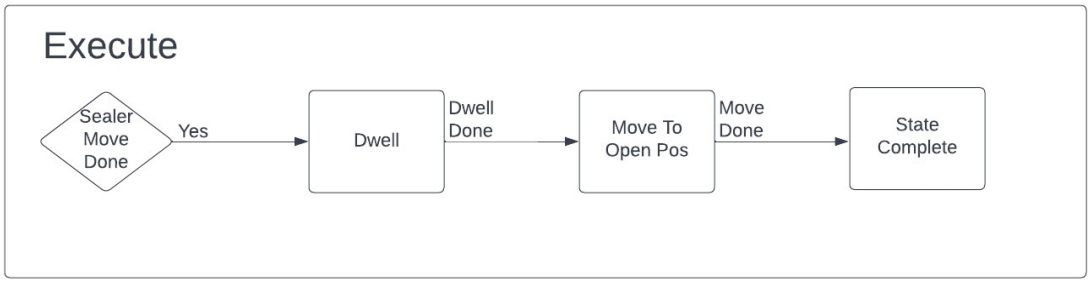
The sealer has one servo to move the seal bar in/out. The seal bar is a heating device for sealing the film, and it is new to this application.

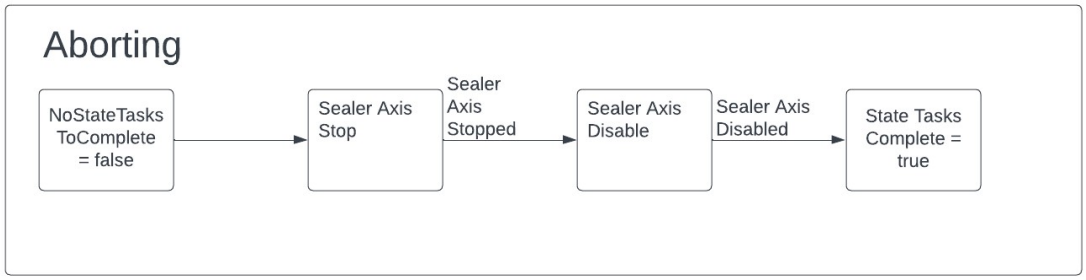
The Sealer module has the following logic in the following states and in Production mode











UML Diagram of Sealer

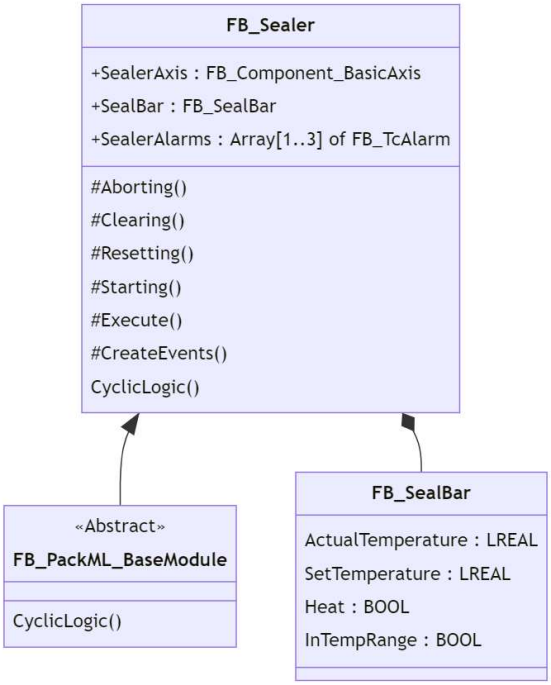
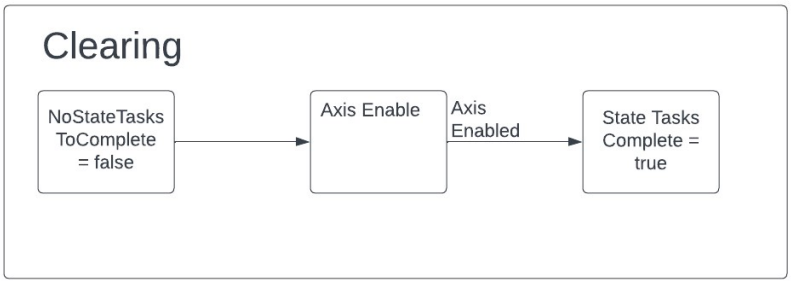


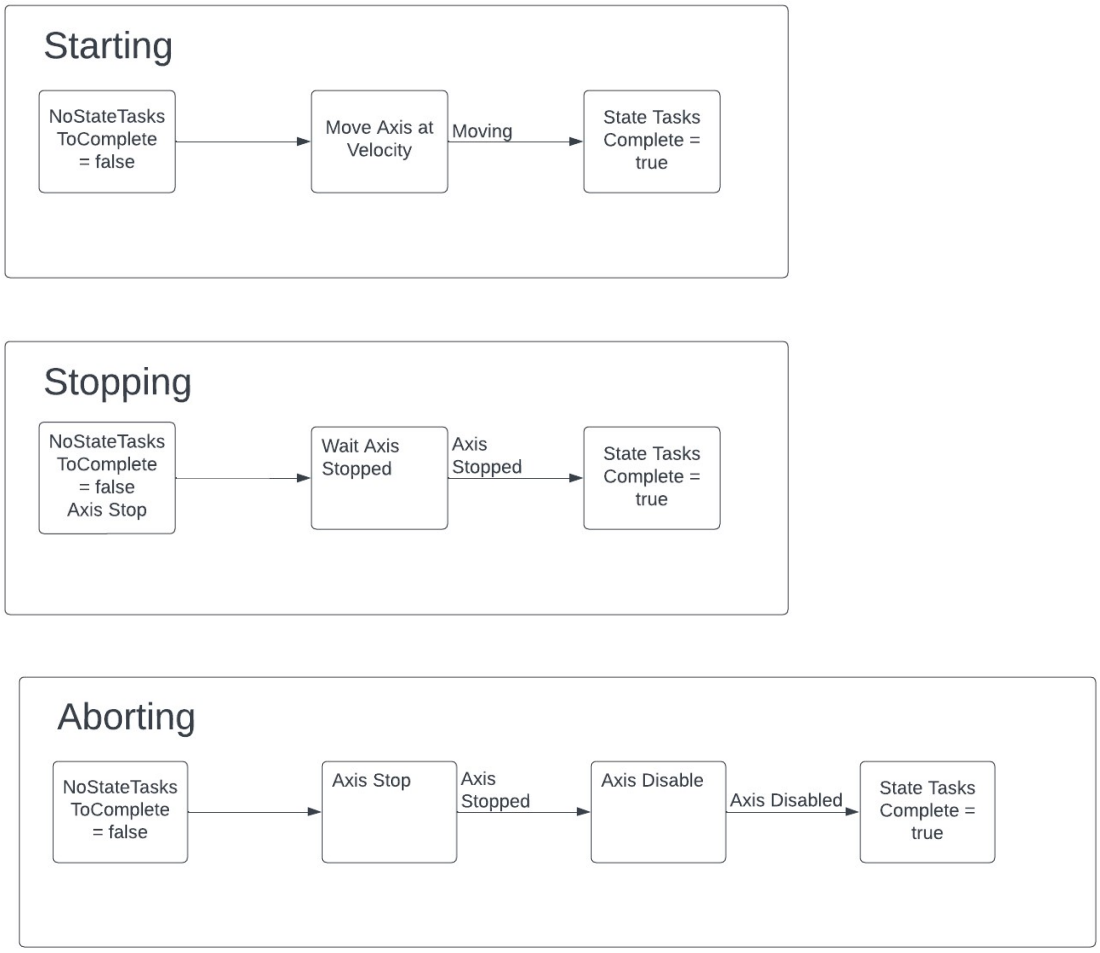
Figure 14 Sealer

7.2.5 Unwind

The Unwind module has one servo and a digital sensor.

The unwind module has the following logic in the following states and in Production mode





UML diagram of Unwind

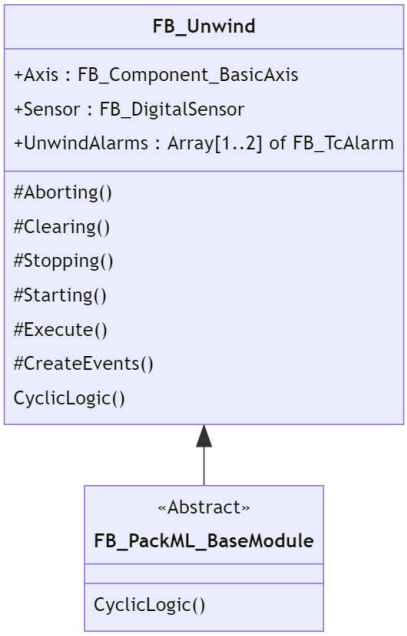


Figure 15 Unwind

8 Components

These are the components created in the application and not currently in the library.

8.1 FB_Cylinder Component

The cylinder component was made in the application as a simple cylinder with one output and no inputs.

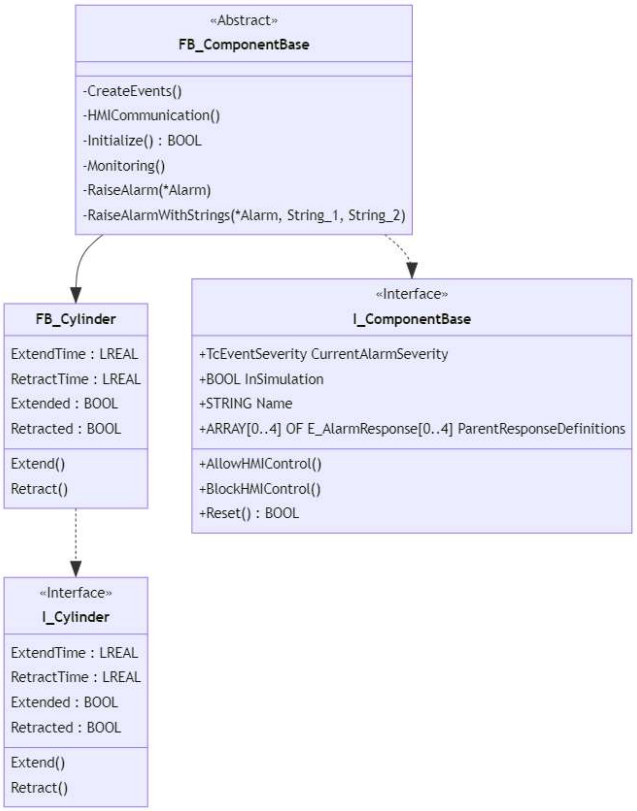


Figure 16 FB_Cylinder

9 TwinCAT Event Logger

This application will need alarms created that can be raised and cleared based on things happening in the application. As in the already created libraries, the application will need to create the list of alarms and get them into the Excel spreadsheet. The following sections will explain the Excel Add-inn, how to create event classes and how to translate the messages that will be needed for your application.

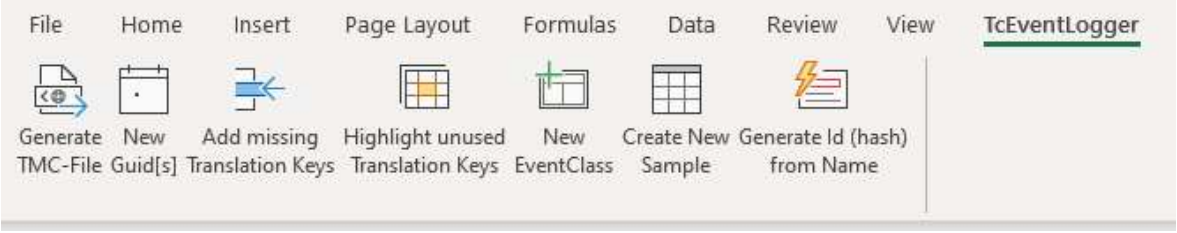
9.1 Excel Add-in

You will need to obtain a copy of the current Beckhoff.TwinCAT.EventLoggerExcelAddIn. While this was being written the version is 1.3.0.8. {How to obtain}

9.1.1 Installation

To correctly install the Add-in, any previous version must be uninstalled. To access Excel Add-in's, you will need to go to File/Options and select Add-ins.

9.1.2 Tool bar



In the TcEventLogger tool bar you will find the following buttons to help you create your Event classes and Alarms. This document does not go into all the features. Please read {document here for reference}

9.1.3 Tabs

The machine module and each equipment modules will have a tab in the spreadsheet. These tabs can be made by using the 'New Event Class' button and giving it a name.

In a module the first line is always:

Name	Id	Severity	Display Text
InitReferenceEvent	1	Info	DT_GLOBAL_INIT_REFERENCE_EVENT

This will be used in the PLC code to create the Events for each class of alarms. All lines following the first will be actual alarms that can be used in the PLC. The Id must be unique and go up in order. The severity is what you chose it to be. The common usages are Error, Warning and Info. The Display Text is a formula that takes the string 'DT_' and adds the class name, '_Event_', and the alarm name for a name like: DT_Sealer_Event_TemeratureOutOfRange. This name is then put onto the translation tab

where you can put your translation for the alarm into many languages. In the US, we use English (1033). There are more tabs, and they can be reference in the document listed above.

9.1.4 Example for the Sealer

Name	Id	Severity	Display Text
InitReferenceEvent	1	Info	DT_GLOBAL_INIT_REFERENCE_EVENT
TemeratureOutOfRange	2	Warning	DT_Sealer_Event_TemeratureOutOfRange
FeedbackError	3	Error	DT_Sealer_Event_FeedbackError

9.1.5 Translations

Key	German (1031)	English (1033)	French (1036)	Spanish (1034)	D
DT_GLOBAL_INIT_REFERENCE_EVENT		Init Reference should never be set			
DT_PullWheels_Event_StartTimeout		{0} - Start up timeout			
DT_Sealer_Event_TemeratureOutOfRange		{0} - Out of Temperature Range ({1}-{2})			
DT_Sealer_Event_FeedbackError		{0} - Sealer feedback error			
DT_Unwind_Event_StartTimeout		{0} - Start timeout			
DT_VFFSDemo_Event_EmergencyStop		{0} - Emergency stop			
DT_VFFSDemo_Event_LowAir		{0} - Low air			

The {0},{1} & {2} are holders for arguments that the PLC code can add at runtime. All the alarm text has a {0} which will be replaced with the module name where the error took place. On the temp range one, the lower and upper limits can be added. This is a nice feature if you want to

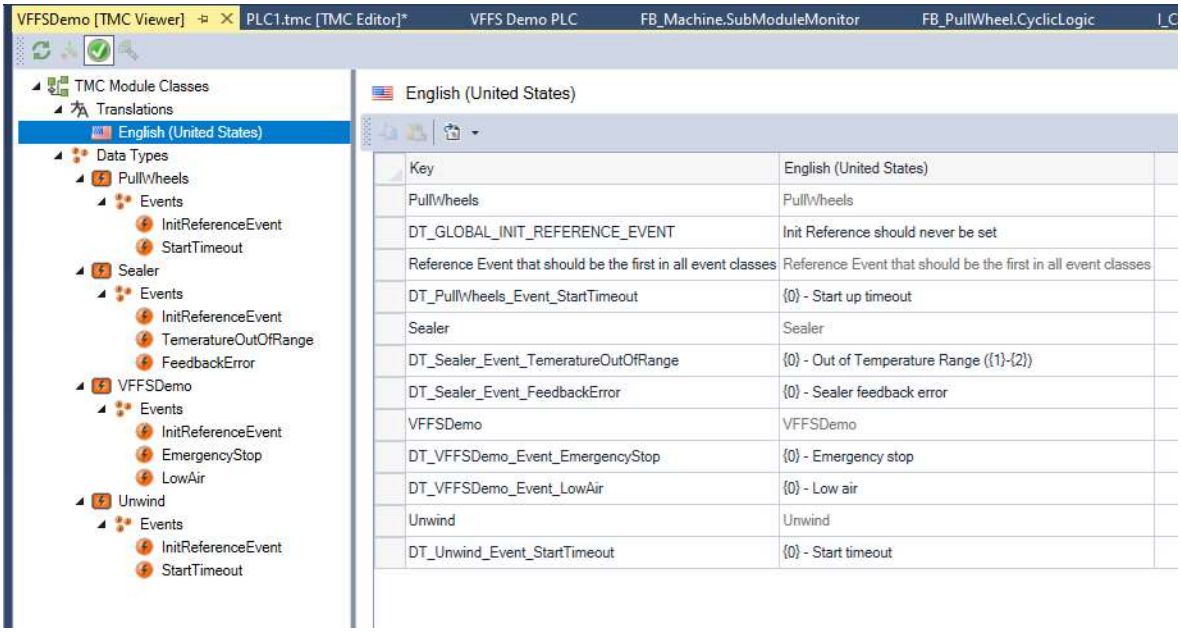
have one alarm that is an error with many reasons the error can happen so that at runtime you can tell the operator why.

9.1.6 Generate TMC-File

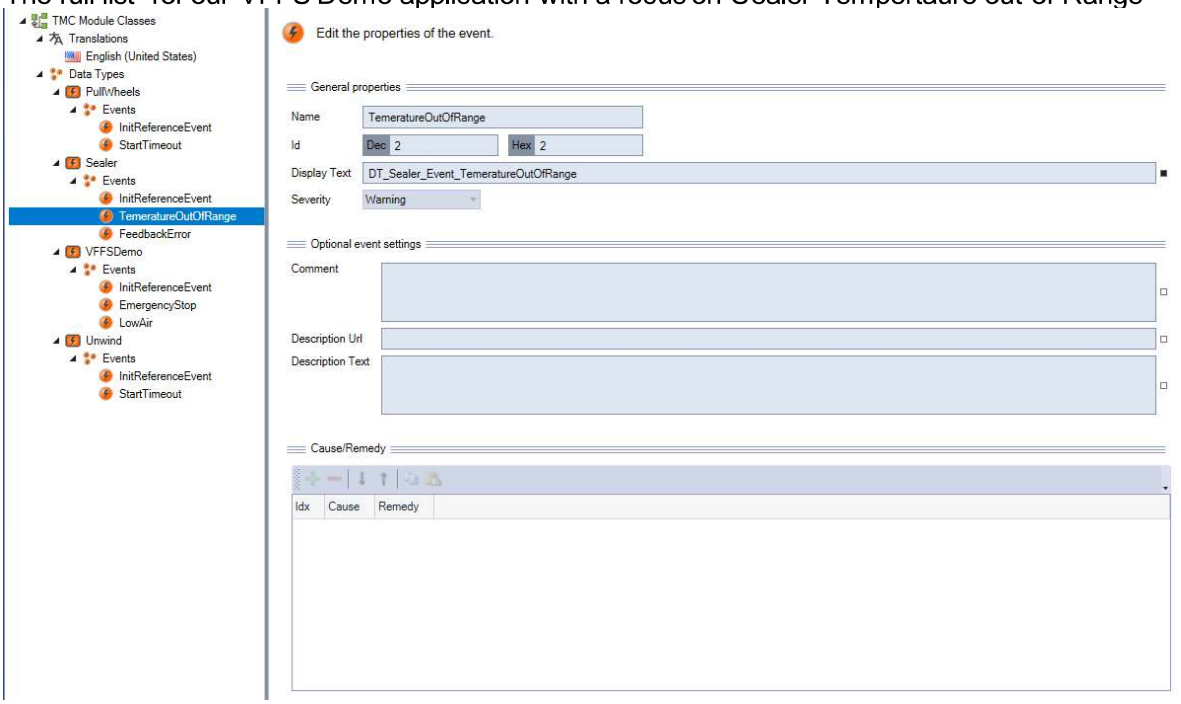
Once all alarms are added and translated, press the Generate TMC-File button to create the TMC that can be added to the PLC program.

9.1.7 TMC File in the PLC Program

Here is the output VFFSDemo.tmc file shown in TwinCAT Translations



The full list for our VFFS Demo application with a focus on Sealer Tempertaure out of Range



9.2 PLC Program

The PLC side of the TwinCAT Event logger consists of defining the alarm array for each module, creating the events at startup, and raising/clearing the alarms.

9.2.1 Defining Alarm Arrays

The alarms are an array of FB_TcAlarm fuction block.

```
//Alarms for this module
MachineAlarms : ARRAY[1..MACHINE_ALARM_COUNT] OF FB_TcAlarm;
```

The MACHINE_ALARM_COUNT is a constant that you set based on how many alarms you put into that tab for your module.

```
VAR CONSTANT
    MACHINE_ALARM_COUNT : INT := 3;
END_VAR
```

This example is the machine module definition, and it has 3 alarms.

9.2.2 Create Events

During the initializing of the machine, each module will call its own CreateEvents method. In this method, it will create the events into the TwinCAT event system.

```
METHOD PROTECTED CreateEvents
VAR_INPUT
END_VAR
```

```
// Add your event creation here
F_CreateAllEventsInClass(Alarms     := MachineAlarms,
                         ClassSize  := SIZEOF(TC_Events.VFFSDemo),
                         pInitEvent := ADR(TC_EVENTS.VFFSDemo.InitReferenceEvent),
                         Prefix     := _ModuleName);

// Super call to base
SUPER^.CreateEvents();
```

The F_CreateAllEventsInClass is from the SPT Event Logger library. The TC_Events.VFFSDemo is created in the tmc file that came from the Excel spreadsheet. Also, as part of creating the events, an enumeration is created that has a number for the array usage in using the alarms by name.

9.2.3 Raising/Clearing Alarms

After you have created the TMC-file from Excel and initialized the events on powerup, you can raise and clear the alarms in the PLC code. Here is an example of the machine module Estop alarm.

```
IF EstopInput AND NOT MachineAlarms[E_VFFSDemo.EmergencyStop].bRaised THEN
    RaiseAlarm2Args(MachineAlarms[E_VFFSDemo.EmergencyStop],'','');
ELSIF NOT EstopInput AND MachineAlarms[E_VFFSDemo.EmergencyStop].bRaised THEN
    MachineAlarms[E_VFFSDemo.EmergencyStop].Clear(0,0);
END_IF
```

9.2.4 Monitoring the Alarms (Equipment Module)

The PLC application code will need to monitor the severity of any raised alarms that are set in the module being worked on and act accordingly. The Machine Module will already be monitoring the CurrentSeverity of each Equipment Module, but we need to set it. So, in the CyclicLogic function in each Equipment Module we must add a function call to pass in the base class severity to check it against the current modules severity.

```
IF NOT _InitComplete THEN
    _InitComplete := Initialize();
    RETURN;
END_IF

SUPER^.CyclicLogic();

//Set the current alarm severity for this module
_CurrentAlarmSeverity := F_GetMaxSeverityRaised(Alarms := PullWheelAlarms, CurrentSeverity := _CurrentAlarmSeverity);
```

This code must be after the SUPER call to CyclicLogic since it will reset the value to Verbose which is the lowest alarm severity in the base class.

9.2.5 Monitoring the Alarms (Machine Module)

In the machine module, since there is no module above looking to monitor the severity of it, we will override the SubModulesMonitor method. First, we will make a call to the SUPER^. SubModulesMonitor to still check the sub modules, and then check the Machine Modules severity and act as needed. This is a sample of the Machine Modules overridden SubModulesMonitor().

```
METHOD PROTECTED SubModuleMonitor : BOOL
VAR
    AlarmResponses : ARRAY[0..4] OF E_AlarmResponse; //Temporary response array
END_VAR

// Check the sub Modules for Alarm Severity
SUPER^.SubModuleMonitor();

//Check the alarms at the machine level
_CurrentAlarmSeverity := F_GetMaxSeverityRaised(Alarms := MachineAlarms, CurrentSeverity := _CurrentAlarmSeverity);

//Check the severity of the machine level alarms
IF _CurrentState <> E_PMLState.ePMLState_Aborted AND _CurrentState <> E_PMLState.ePMLState_Aborting AND _CurrentState <> E_PMLState.ePMLState_
   _CurrentState <> E_PMLState.ePMLState_Stopped AND _CurrentState <> E_PMLState.ePMLState_Stopping THEN
    AlarmResponses := ParentResponseDef;
    CASE AlarmResponses[CurrentAlarmSeverity] OF
        E_AlarmResponse.Abort_ImmediateError:
            AbortImmediateError(ModuleName, TRUE);
        E_AlarmResponse.Abort_Immediate:
            AbortImmediate();
        E_AlarmResponse.Stop_Immediate:
            StopImmediate();
        E_AlarmResponse.Stop_Controlled:
            StopControlled();
        E_AlarmResponse.Hold_Immediate:
            HoldImmediate();
        E_AlarmResponse.Suspend_Immediate:
            SuspendImmediate();
        E_AlarmResponse.Suspend_Controlled:
            SuspendControlled();
        E_AlarmResponse.NoResponse:
            ;
    END_CASE
ELSIF _CurrentState = E_PMLState.ePMLState_Stopped OR _CurrentState = E_PMLState.ePMLState_Stopping THEN
    AlarmResponses := ParentResponseDef;
    CASE AlarmResponses[CurrentAlarmSeverity] OF
        E_AlarmResponse.Abort_ImmediateError:
            AbortImmediateError(ModuleName, TRUE);
        E_AlarmResponse.Abort_Immediate:
            AbortImmediate();
        E_AlarmResponse.NoResponse:
            ;
    END_CASE
END_IF
```

10 Appendix

10.1 Beckhoff Support and Service

Beckhoff and their partners around the world with the exception of the USA where the XTS system is not offered, comprehensive support and service, guaranteeing fast and competent assistance with all questions related to Beckhoff products and system solutions.
Beckhoff Support and Service representatives round the world and can be reached by phone, fax or e-mail. The contact addresses for your country, please refer to the list of Beckhoff's branch offices and partner companies.

Beckhoff Support
Support offers you comprehensive technical assistance, helping you not only with the application of individual Beckhoff products, but also with wide- ranging services

- Worldwide support
- Planning Programming and Commissioning of complex Automation Systems
- Extensive training program for Beckhoff System Components

Hotline: + 49 (0) 5246/963-157
Fax: + 49 (0) 5246/963-9157
E-Mail: support@beckhoff.com

Beckhoff Service
The Beckhoff Service Center offers support for all post sales service matters.

- On Site Service
- Repair Service
- Spare Part Service
- Hotline Service

Hotline: + 49 (0) 5246/963-460
Fax: + 49 (0) 5246/963-479
E-Mail: service@beckhoff.com

10.2 Beckhoff Worldwide Headquarters

Beckhoff Automation GmbH
Eiserstr. 5
33415 Verl
Germany
Telephone + 49 (0) 5246/963-0
Fax: + 49 (0) 5246/963-198
E-Mail: info@beckhoff.de
Web: www.beckhoff.de
Further Support and Service addresses and product documentation can be found at http://www.beckhoff.com