„Closed Loop Safety PLC Systems„

Concepts for safe IEC 61131-3 compliant PLC Systems

Dipl.- Ing. Stefan Angele
infoteam Software GmbH Bubenreuth Germany

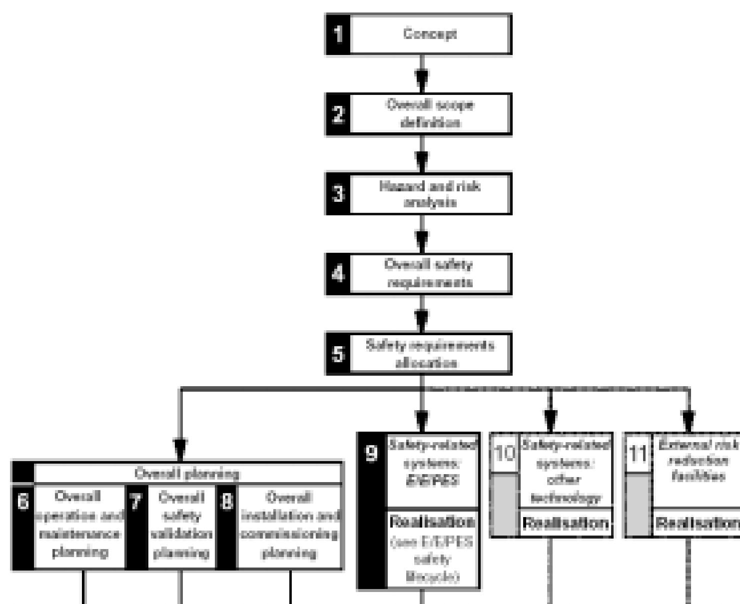
Abstract

The architecture of PLC systems for safety related applications is usually driven by hardware dependent safety issues. Programming and engineering software systems are developed with huge efforts to fulfill safety requirements focused on individual target environments.
infoteam Software GmbH as a provider of standard IEC 61131-3 programming and runtime environments now presents hardware independent concepts as a closed loop of safety mechanisms affecting all layers of a universal safety PLC system architecture. The selective usage of diversity is “closing the gap” of traceability between the user and the running system.

Introduction

Along with the IEC 61131-3 standard there comes a huge potential to build up component libraries and ready made solutions for PLC applications known from other high language environments like Pascal or C++.
Besides the benefits of saving engineering costs one of the biggest advantages is the possibility to continuously improve and validate software modules for specific tasks until they are high quality components that are proven in use. These components then can be saved from being changed unattended e.g. by encryption and can be easily reused in any safety related application domain offering their services only by an dedicated interface.

The IEC 61508 software lifecycle defines the mandatory need of an conceptual software design as well as the validation and verification against precisely outlined requirements.
IEC 61131-3 is the only standardized approach to an component oriented PLC software development. Requirements Management methods like VOLERE, as well as design methods like UML are mainly developed with respect to component or object oriented software development. To be able to benefit from the well validated and worldwide verified programming standard paradigms IEC 61131-3 compliant systems are rather suitable to develop safety PLC application programs.



Because the IEC 61508 is not only addressing the development of application software but is also beingconsidered during the development of system software like an programming environment, infoteam software as an experienced provider of modular IEC 61131-3 solutions has set up a safety

PLC System software concept for an hardware independent safety system design capable to be ported to any hardware suitable for individual safety domain requirements.
This closed loop IEC 61131-3 safety concept fits the basic requirement to be consistent in a way of closing the virtual gap between the user, the target system and vice versa. This requirement basically relies on the assumption that the application it self can be made safe only at a certain limit by means of software and hardware design and implementation. Any intelligent safety mechanisms only can be efficient and valid as long as the user is part of it. And the user again has to rely on the system considering the fact that his input is processed and interpreted as intended and is not changed by any system error. This made us think about safety concepts always searching for a solution to assure the users input not being manipulated or misinterpreted by the programming or runtime system.

The following presentation of this concept first wants to provide an overview of these terms and definitions which are later used to introduce specific safety providing mechanisms used on several layers of the overall IEC-61131-3 environment.

The overall concept can be seen as a template for the implementation of any customer specific safety programming system. It is intended to provide the fundamental basis for a IEC 61508 compliant system development and is also intended to be subject for individual improvements and validation by according certification authorities.

Motivation

What is the purpose of this concept? The answer might sound like “because the IEC 61508 is requiring that!” This is definitely right, but infoteam is convinced no good jobs are done on an motivation like this!
Development of quality software needs a good design, and safety critical software has to be even higher quality software. You surely will believe, that a software suit like OpenPCS is developed on the basis of several well defined software design concepts. Proven in use reports together with extensively testing makes our OEM customers solutions fit for use in industrial IT environment. Nevertheless the design of a security system is underlying extended requirements regarding the safety strategy, concepts and the according safety mechanism solutions. Presenting an throughout consistent safety concept and explaining our overall safety intentions we want to create a common understanding of safety to all project stakeholders like the customer safety experts certification authorities and software development engineers. Everybody being part of an safety system development project has to agree to a common definition of the safety he or she is liable for.

Concept Topics

Designing a closed loop safety concept we came upon different topics to be handled separately. Starting with a general approach on a definition for safety in relation to functionality we identified appropriate software diversity variants regarding the detection of software errors.

With that background it is necessary to investigate the single system architecture layers searching for possible sources of errors and finding appropriate safety mechanisms.

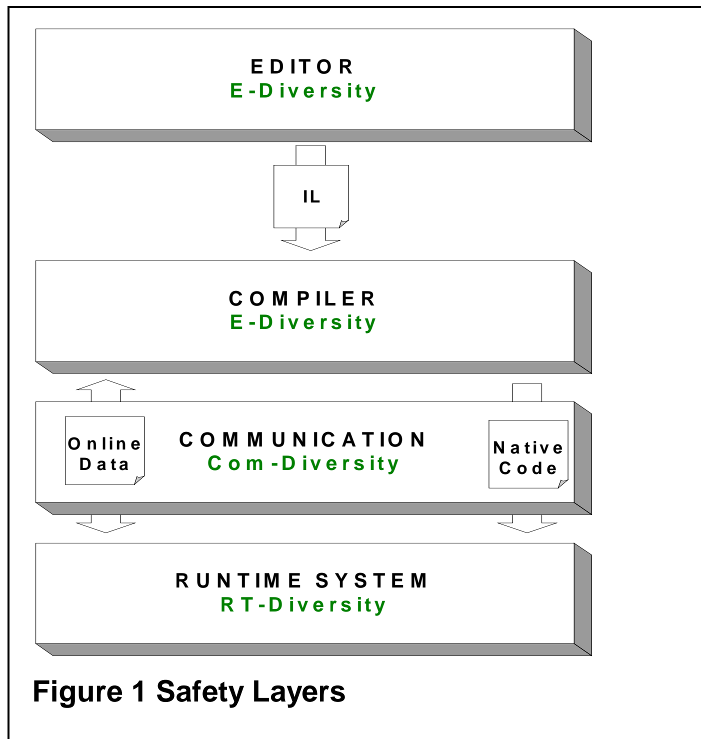


Figure 1 Safety Layers

Safety vs. Functionality

The number of source code lines of an software module as well as the amount of components on printed circuit board are in direct relationship to the probability of undetected errors. Safety and functionality are to be seen as concurrent issues.

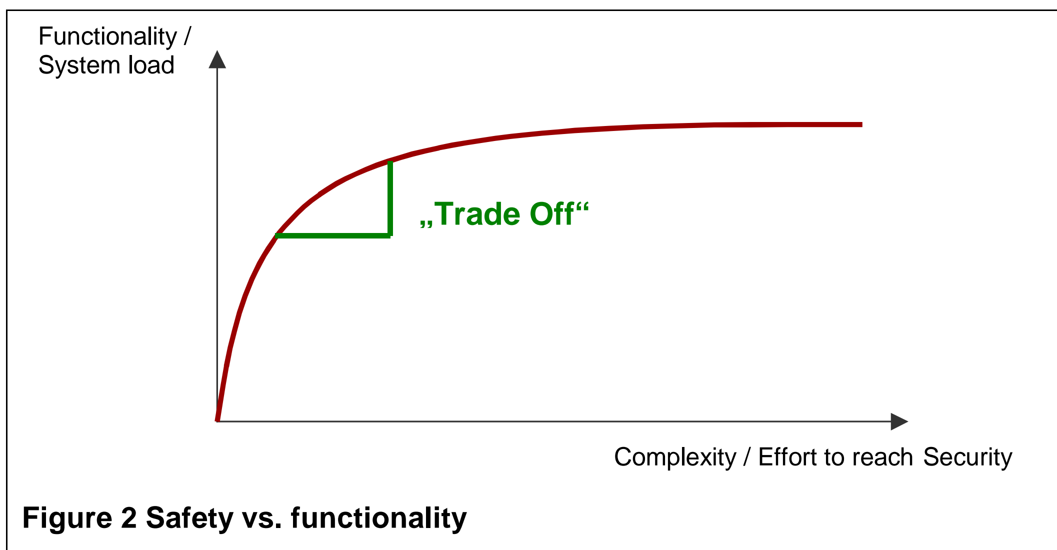


Figure 2 Safety vs. functionality

Because of this the most easiest way to increase the safety of a system is to reduce its functionality. Actually this is an issue to be serious considered raising up functional safety requirements. Nevertheless safety system vendors have to provide solutions for complex automation tasks competing on an open market!

This will lead to a trade off between individual necessary functionality and more or less efforts on safety mechanism implementation.
IEC 61508 as well as PLC-Open and EWICS accordingly defined subsets of languages, editors and data types giving recommendations whether to support or rather not support depending on the intended level of individual system safety.

Because this is obviously depending very much on the customers application domain and target specific requirements infoteam's IEC 61131-3 environment is individually scalable to support any subset of editors data types an IEC 61131-3 standard functions and function blocks.

Software Diversity

Software has errors! Since window based operating systems with high level API's this already applies to a simple "Hello World" application.
Regarding the fact that it might not be possible to write an application with no errors on a perfect operating system we have to search for different methods and techniques to detect the errors and to handle them in a safe way.
Diversity basically is based on the assumption, that many ways to the same aim will increase the probability to detect a wrong implementation depending on how much and how different ways are taken simultaneously.

In terms of software, sporadic errors are rather unlikely and there are appropriate methods to avoid them following appropriate coding conventions or languages. (Initialization of pointers …)
Software errors are more likely to be systematic errors that nevertheless may be rather hard to detect or reproduce. This has to be considered taking a choice on diversity mechanisms for software.
As the following table shows there are two different kinds of diversity applicable for software systems.

Kind of Diversity	**Recognizable errors**				
	Sporadic HW errors	Static HW errors (drop out)	Manufactu-ring defects	**Implemen-tation errors**	**Functiona-lity errors**
Environment (e.g. time)	yes	no	no	no	no
Physical	yes	yes	no	no	no
Hardware manufacturer	yes	yes	yes	no	no
Implementation	yes	yes	yes	yes	no
Functional	yes	yes	yes	yes	yes

Table 1 Diversity variants

1. **Implementation diversity to detect implementation errors**
   Two or more software units developed by independent software teams with different tools realizing the same functionality running in one system context. Comparing the results of the diverse software units processing identical inputs is decreasing the probability of implementation errors that are unlikely to be the same in diverse implemented pieces of software.

2. **Functional diversity to detect software functionality errors**
   Approve the correctness of software units by checking the results with a different software unit. This can be any kind of CRC checking consistency checks or reverse transformations. Basically this diversity variation is reducing the probability of functional errors by monitoring and validating the correctness of an calculation with another controlling and protecting soft unit not necessary for the systems functionality but reproducible logically related to the systems software unit algorithms result.

Programming System

Knowing about the different meanings and effects of diversity it is possible to identify the necessities and possibilities to utilize different diversity mechanisms efficiently for the different parts of the overall programming system environment.

System and Application Data Safety

To assure that the programming system is working without errors it is first of all necessary, that the system's executables were not exchanged or altered by any unqualified means. I.e. only a validated and officially tested and released version of the system's binaries is allowed to be used to create valid application data.

Therefore all the systems safety mechanisms that keep implementations of functional diversity algorithms had to be encapsulated into one binary system security module (DLL) and is validated with key response authentification mechanism right at the very beginning of the system's framework start up in the context of an **P**ower **O**n **S**elf **T**est.

After this was successful the system is performing a **B**uild **I**n **S**elf **T**est where all binaries and initialization files integrity is CRC checked against an system global system security data configuration file that is also CRC protected. If only one of the systems components is detected as not compliant, the system will shut down with an user information message.

If the BIST was passed successfully the second part of the POST is performed compiling a reference test project that is delivered with a system installation version and is unpacked before every system start. This project is compiled and again checked against

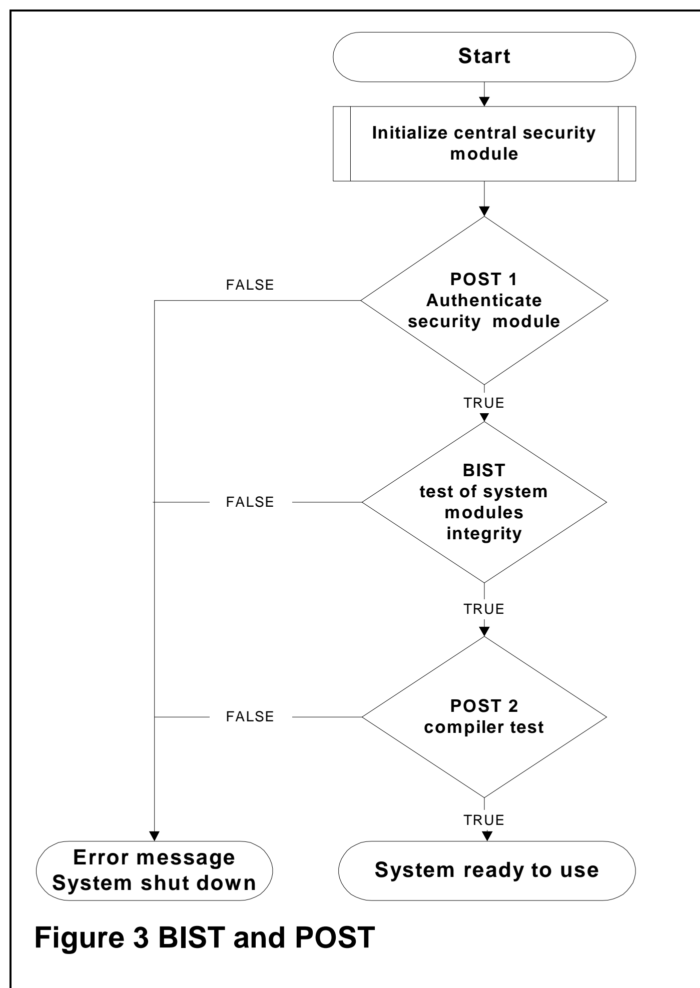


Figure 3 BIST and POST

related CRC data in the system security configuration file. Only if the compiler works correctly the reference project is deleted and the system will be ready to use.

Application Data Safety

Application files like POU's (programs, functions, function blocks) and all application project data are protected against any unqualified changes. The consistency and validity of all source files as well as the generated object files are CRC-safeguarded and every system access to those data is so, too. Besides this all sources and objects (downloadables) belonging to an IEC 61131-3 resource are packed and CRC-safeguarded into a data repository "safe". This repository can be moved only as one whether it is downloaded to the target or copied to a disk and can be extracted only as one again to provide a safe and consistent set of sources and binaries needed to rebuild the specific project resource.

The overall entry into separate areas of the system ore even of parts of the project is password protected. Different user levels can be defined for different levels of security.

Application solution libraries can be encrypted and with that hidden from the users view according to the current users safety level.

Every source file is versioned by a system internal versioning mechanism. Any kind of commercial version control system can be used for configuration management.

Files which are imported from other projects or data storage media into the current project are initially marked as not proofed. This should focus the users attention to the fact, that this file is not natively belonging to the project and should be checked.

This is just a short overview of some of the data safety mechanisms. Basically the system is open to be adapted according to any additional data safety requirements.

Tools Safety

Different programming systems tools safety issues have to be realized according to their complexity and criticality in the context of the overall PLC engineering environment.
To be able to choose from different safety mechanisms appropriately or to be able to participate from off the shelf safety related third party components it is necessary to provide a modular system architecture with standardized well documented interfaces. Nevertheless modularization of a safety related engineering environment is again a trade between enabling a maximum level of customization and preserving as much as possible common reusable system components to be subject of changing as less as possible.
The OpenPCS system architecture uses ActiveX controls as well as DLL libraries and COM servers. This provides the possibility to reuse most of the vital system functionalities as for example a common system safety module and on the other hand is open to integrate third party solutions as e.g. safe communication protocols. The ControlX framework allows to plug in any additional tools e.g. a C&E editor as an ActiveX control component.
This brings us to the issue of editors in safety related programming environments. This topic we want to address in detail as follows.

Editors

In an IEC 61131-3 programming an debugging environment editors are the first level user interface. To provide safety while editing, one way is to use only especially safety dedicated editors like C&E editors. This again leads to the functionality vs. safety trade off problem. As a choice of IEC 61131-3 compliant languages the Function Block Diagram language and the Ladder Diagram language are most suitable for safety related applications. Nevertheless in an open and scalable system it is also possible to restrict any other language on a safe syntax and data type subset. This might be reflected in forbidding the declaration of direct addressed variables in programs, forbidding while, and for loops in structured text and so on.

Additional means to reduce the probability of errors are incremental syntax checking mechanisms as well as assistants that guide the user during writing of code.
All those safety issues are concerning in any way the editor's user interface.

But this kind of "UI-safety" only makes sense if it can be guaranteed that the logic engine of the programming system will exactly generate code for what the users input describes. That means only if it can be assured that the displayed logic will really be processed down on the target.
This is what leads us to the necessity of an closed loop bringing the users input into the system and vice versa back again to the display.

On the systems editor layer this closed loop is realized on functional diversity levels.

There is the re-transformation of all user inputs during edit time:

This is done for all graphical editors by transforming any graphical input into an internal working set representation. This logical working set data is parsed, verified through plausibility checks and if valid transformed into the displayed graphical data. With that the graphical editors are rather an intelligent logic transformation machine than a drawing tool.

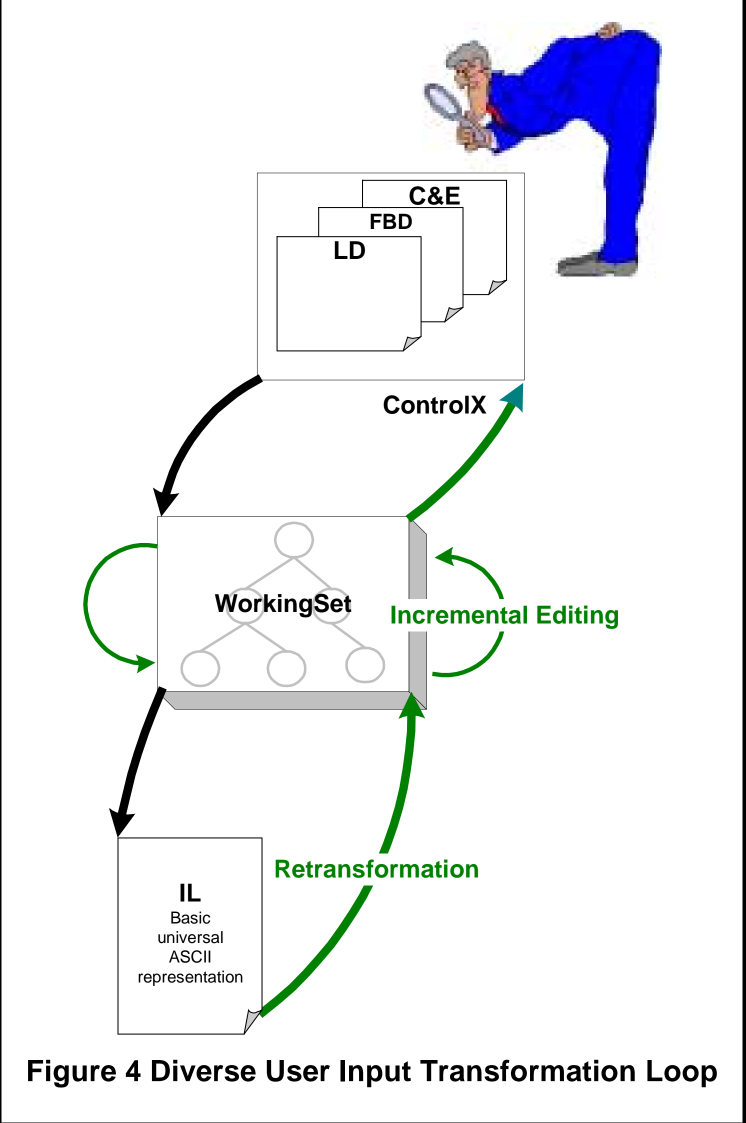


Figure 4 Diverse User Input Transformation Loop

The safety benefit that comes along with this mechanism results in the following facts:

a) No invalid logics in terms of the specific language syntax and semantics can be edited
b) Due to an internal safety backup even in case of a software exception a valid data set is guaranteed

So in this first level loop functional diversity is implemented in the way of back translation from working set representation into graphical information and in the way of intelligent fail safe plausibility checks and monitoring of software exceptions.

The second “level” description of the editor transformation loop is based on the fact, that all editors of the OpenPCS environment create an instruction list output. IEC 61131-3 Ladder and Function Block diagram as well as C&E semantics can be re-transformed from ASCII IL into the according graphical presentation. With this it is possible to have an additional functional diverse retranslating “path”.

Compiler

Due to the fact mentioned above, there is only one safe compiler necessary for any customized safety editor environment independent from an individual choice on language editors. This universal editor is documented by long term approval reports and detailed bug tracking history reports. The editor’s logic engine (scanner parser) was created with the help of an compiler compiler CASE tools that is also established and verified in lots of independent compiler implementations. A second command line toolbox edition of this compiler is available and can be seamlessly integrated into a safety system to be used in parallel to perform an additional diverse compile run.

Corresponding to the individual editors the compiler is also limiting available syntax and data types to a safe subset prompting according error messages to user during the compilation process.

Because we called our concept a “Closed Loop Safety” you will expect a special realization for the compilers, too. The compiling process in OpenPCS is a two fold approach. The first compile run is always creating a universal interpreter code representation called UCODE. This UCODE than can be directly interpreted by an virtual target independent IEC 61131-3 runtime machine. Optional there is the possibility to plug in a native code generator that transforms UCODE into target and processor optimized code that is directly executed on an individual target.

Reverse translation is possible on the basis of UCODE and a symbolic table mapping IEC identifiers and data types to UCODE data offsets.

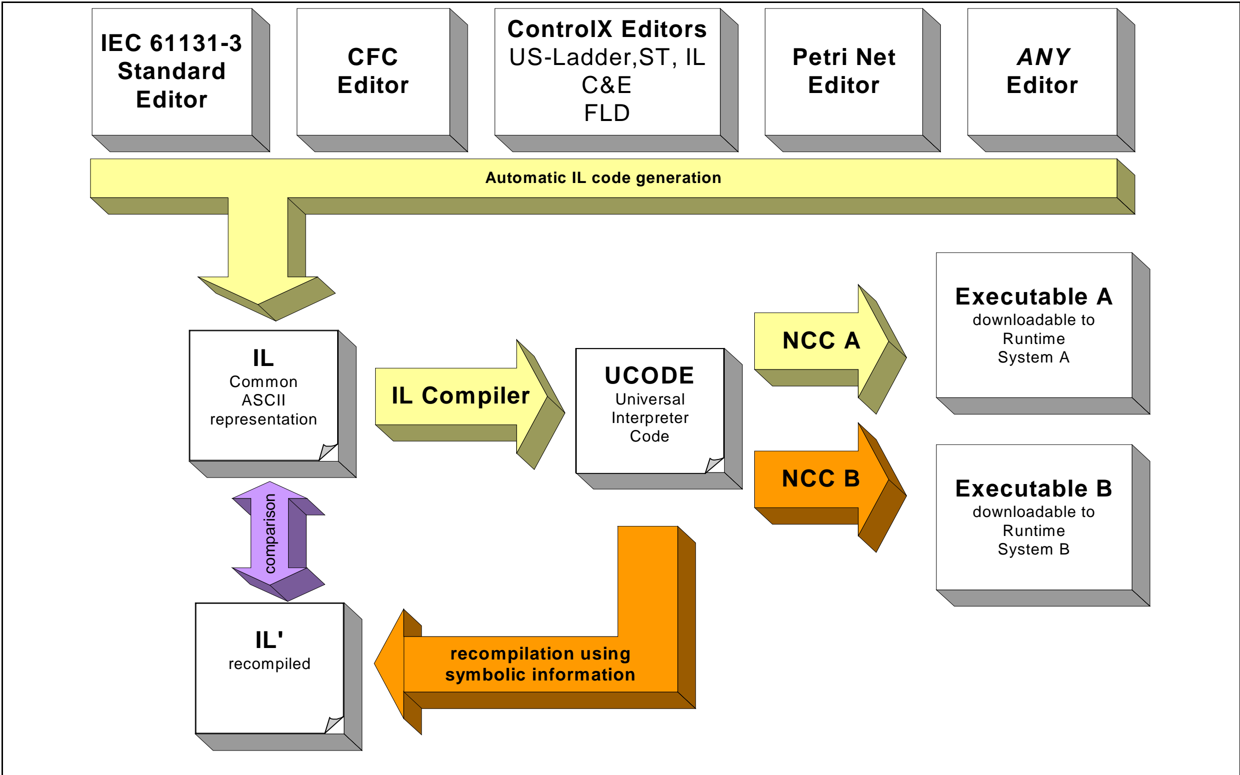


Figure 5 UCODE Recompilation

Recompilation results in in an IL' representation which can be compared to the according original IL source to validate the compiling process.

Two diverse implemented NCC native code compilers could additionally assure safety for native code executing targets. Of course it would be also an option to use a diverse IL-to-UCODE-compiler as we mentioned before.

Runtime Environment

Redundant systems bring along a bunch of requirements on the runtime system side concerning availability and consistency as well as synchronicity of program execution. The OpenPCS runtime environment architecture is capable to be individually adapted to redundant target system environmental needs in different ways.

Redundant IEC Runtime System Architectures

Following we describe two mayor redundancy system architectures that we want to refer to in this concept. Both of them running one system as master and the other one as slave. A loss of the master activates a slave to continue the program execution with a minimum loss of data and execution time.

Cold Stand-By

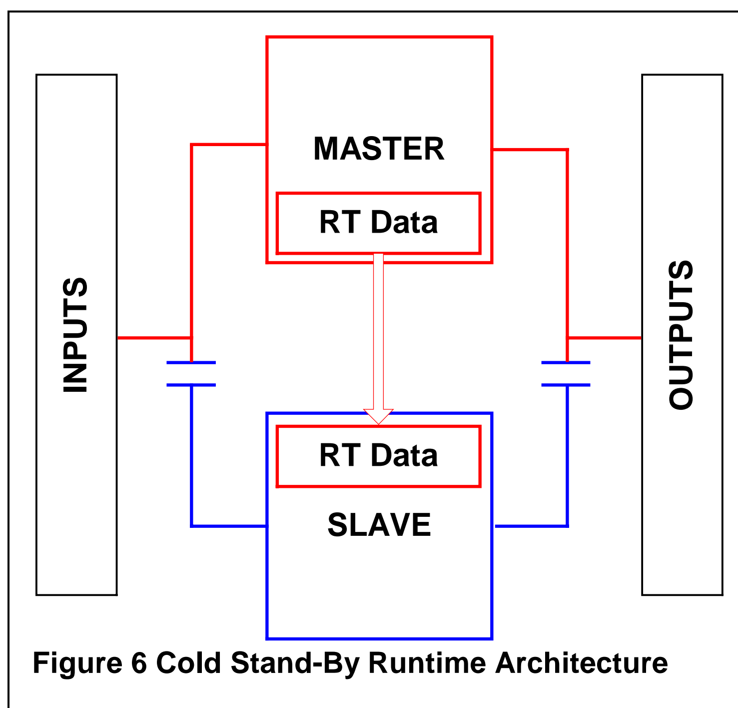


Figure 6 Cold Stand-By Runtime Architecture

In a cold stand-by system the slave is not executing the program. Only the master is reading and writing the inputs according to the processed runtime data (RT data). The RT Data have to be synchronized with the inactive slave so that in case of a loss of the master the slave can continue the program execution without any loss of RT data.
As this architecture just prepares a slave continuously to start execution it dos not have to take care of synchronizing any parallel executions that may cause delays on the masters side.

On the other hand copying the RT data can be critical for the master system performance depending on the frequency and data volume of the copying process.

Hot Stand-By

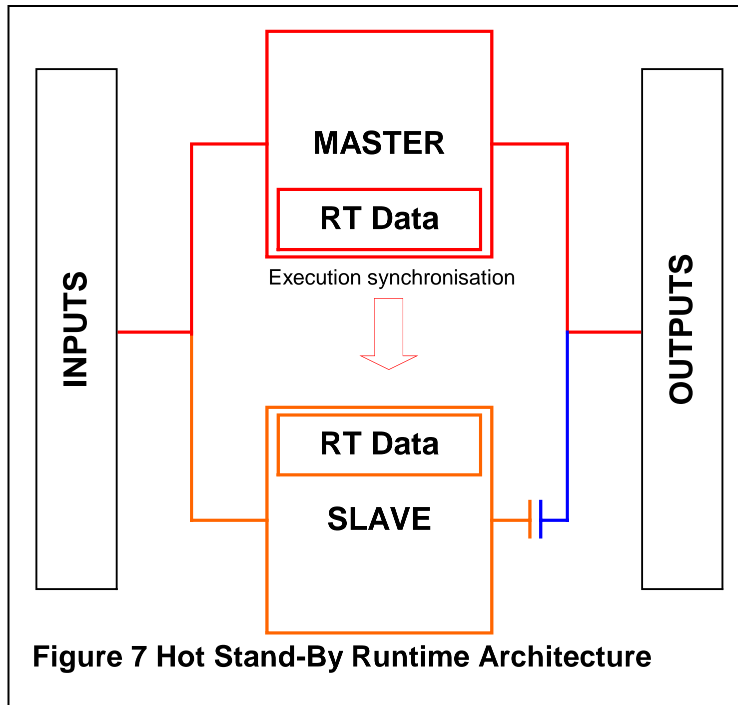


Figure 7 Hot Stand-By Runtime Architecture

In a hot stand-by system the master and the slave are executing the IEC program. Master and slave are reading the same inputs and processing separate identical data images.
To ensure that master and slave are always working with the identical input image the master will have to copy it to the slave input image.
Only the master writes to the outputs.
In case of a loss of the master the slave is immediately continuing the execution of the program with its own set of identical runtime data without any further initialization. The outputs now also are written by the slave.
To ensure that the RT Data of the master and the slave are unambiguous it is necessary to synchronize the program execution in an appropriate way. We will come back in detail to an description of different approaches to that later in this document.
Compared to the cold stand-by architecture this solution dos not have to copy RT data except in case of substituting a system component during runtime (Hot Plug-In)
On the other hand it might be difficult and time critical to synchronize parallel execution.

Switching Master and Slave Operation Modes

Due to the fact that a master has to become a slave in case of a master system loss, the Runtime System has to be extended in a way to support both operating modes master and slave mode as well. Usually we assume there is a hardware "alive" interconnection between the master and the slave that indicates whether one of them is out of order.
If there is the requirement to substitute a defect system component during runtime (Hot Plug-In) basically each of them has to be able to run as master, as slave and stand alone.

Data Synchronicity

Regarding the architectures above mentioned an important requirement for redundant IEC runtime environments is to ensure that a slave system can be activated as fast as possible working on the last valid dataset created by the master that has dropped out.

Using shared memory for master and Slave will initially provide an all-time consistent data image for master and slave. On the other hand it has to be considered that also a shared memory hardware architecture will have to fit the overall system availability requirements.

In hardware architectures with separate memory images data will have to be copied from the masters IEC memory image to the slave systems one to ensure the consistency of data between redundant master and slave system components.

This might be necessary, if

- A cold stand-by redundancy architecture has been chosen and with that the slave's data has to be continuously synchronized in advance of an master system loss.
- A hot stand-by redundancy system part has to be substituted and initialised by the master with a current valid set of RT data after a "Hot Plug-in". In fact this will cause a dead time in IEC program execution for at least the time the system needs for copying the data to the plugged in system memory. Stopping is necessary to avoid data "aliasing" of the master system meanwhile the copying process.

The process of copying RT data can be basically done at two different times

1. after the execution of every single cycle.
2. after the execution of every single task.

Copying RT data can be very time consuming depending on the data volume and according hardware features.

Execution Synchronicity

Synchronizing the cyclic execution of an IEC Task can generally be done following two different strategies.

- Clock Synchronized CPU Instruction execution;
  Needs a connection between the redundancy system's CPUs that allows to control the delaying of command execution controlled by the master's CPU.

- Synchronized IEC Runtime execution
  Need to have a common runtime Clock and a communication channel to synchronize execution on cycle level. (Avoid aliasing of timers)

There may be additional hard and software implementations to add redundancy to I/O system to ensure, that the right inputs are read and the correct outputs are written. This again is more an item to be considered in real diverse systems that also may want to compare the results of master and slave before writing output values.

OpenPCS Redundant Runtime System extensions

The OpenPCS runtime environment is available as ANSI C Source Buy Out. It is possible to port this runtime system to any target environment and to adopt and extend it according to any redundant target system architecture requirements as existing OEM redundant system solution are able to approve!

Individual hooks are available in the runtime system source code implementation to address all of the above mentioned adaptations necessary to realize any of the described redundancy items. Please directly refer to infoteam Software for any detailed information on that.

Conclusion

This concept will definitely not save the effort of setting up precise safety requirement specifications that will have to be verified and validated by according certification authorities.

But with this safety concept and the introduced OpenPCS toolbox elements it is possible to set up an individual safety system architecture using existing implementations on the one hand and integrating any additional domain specific safety related tools and mechanisms on the other hand.

The open and thorough concept of dedicated safety layers with individual diverse safety mechanisms is best prepared to realize individual overall Closed Safety Loops made of any kind of specialized and customized “ropes”.

Referring to the beginning of this document we were pointing out the benefits of IEC 61131-3 in terms of re-using proven in use software components. Component Based Automation in an overall engineering lifecycle context is another issue which is addressed by a separate concept of infoteam. In there we defined an engineering process model that integrates all engineering disciplines flexibly with any related tools into an intelligent data base driven assistant supported framework. There the user is guided with a structured and controlled process work flow that is scalable to any application domain requirements.

Tailoring this framework for safety engineering equipped with a safety related programming and debugging environment is closing another remaining gap between safe programming and the realization of an IEC 61508 compliant engineering process model.