Kevin Toon
Open Automation and Control

IEC 61131-3 in Safety Applications

IEC 61131-3 in Safety Applications

Kevin Toon
January 2002

Open Automation and Control
Langford Hall Barn, Witham Road, Langford, Essex, CM9 4ST
Email: kevint@oacg.co.uk

The majority of safety related systems within process applications rely on plant or process specific software to define their functional safety operation. Previously, many of these systems have employed bespoke programming languages and environments. However, IEC 61131 is increasingly expected, and its absence may be a barrier to entry for some sectors. This paper presents an overview and introduction to elements of IEC 61508 as a summary of the safety requirements, how some of these requirements are addressed by an implementation of IEC 61131-3, and the practical experience of addressing these requirements whilst adopting a Commercial Off The Shelf (COTS) package.

Introduction to IEC 61508

Safety is the expectation that a system will not lead to risk to human life or health. Functional Safety is the ability of a system to perform the actions to achieve or maintain a safe state for the Equipment Under Control (EUC). IEC 61508 is an international standard covering Functional Safety of Electrical/Electronic/Programmable electronic (E/E/PE) safety-related systems.

Where EUC present hazard to humans, measures are required to reduce the risk to an acceptable level. The risk reduction is grouped into four Safety Integrity Levels; SIL1 providing the lowest reduction appropriate for safety-related applications, SIL4 providing the highest risk reduction.

The SIL is attributable to each safety function or Safety Instrumented Function (SIF) for E/E/PE Systems. A Safety-Instrumented System (SIS) provides one or more SIFs.

The approach to achieving functional safety is through an overall safety-lifecycle. This lifecycle defines the activities and technical measures required at each stage of a system's life, from initial concept through to decommissioning. These requirements apply to the overall system, including the EUC, sensors, actuators, control and protection systems, human factors, etc.

Is IEC61131-3 Acceptable for Safety-Related Application?

The basic program language requirements are summarized in 61508-3 table A.3; 61508-7 table C.1 provides language recommendations according to each SIL. Although a simplification of the assessment, this will suffice for this paper. For our purposes, entries 15-24 are applicable:

Programming Language		SIL1	SIL2	SIL3	SIL4
15	Ladder diagrams	R	R	R	R
16	Ladder diagram with defined subset of language	HR	HR	HR	HR
17	Functional block diagram	R	R	R	R
18	Function block diagram with defined subset of language	HR	HR	HR	HR
19	Structured text	R	R	R	R
20	Structured text with defined subset of language	HR	HR	HR	HR
21	Sequential function chart	R	R	R	R
22	Sequential function chart with defined subset of language	HR	HR	HR	HR
23	Instruction list	R	–	NR	NR
24	Instruction list with defined subset of language	HR	R	R	R

IEC 61131-3 In Safety Applications

Implementation of IEC61131-3 for Safety Applications

Traditionally, safety-systems have been developed in-house, specific development contracts, or re-use of existing designs. Market expectations for increased facilities and functionality, more stringent safety requirements and certification needs have led to higher initial development costs. Safety-systems are, however, subject to development cost and time to market issues in the same manner as other commercial developments. We therefore decided that existing best-in-class third party software components should we used where practical and where they could be enhanced to achieve the necessary safety requirements. The IEC61131 toolset was one obvious candidate.

The realization of an IEC61131-3 toolset and runtime for safety-related applications requires the application of IEC 61508, including where third party components are used. Third Party packages are pre-existing; the requirements therefore differ from those for specifically developed components. There is lack of conformance to the IEC 61508 safety lifecycle, however the proven-in-use evidence is available. To establish the safety requirements a hazard analysis is undertaken to identify where safety measures need to be applied, particular focus is placed on the elements where error may be introduced. Considering a typical application development process, possible sources of error include:

- Programming error
- Program storage and retrieval
- Translation/compilation
- Download
- Program execution
- Interaction with other software
- Version management

The original specification is another obvious source of error, but is largely outside the scope of this paper. The hazard identification is common to third party and specifically developed components, with both approaches requiring strategies to mitigate these hazards.

Programming error

IEC 61131-3, particularly FBD and LD, is a limited variability language, reducing the potential for programming error. Low-level programming (IL) should be avoided, as it does not readily promote the procedural measures identified later. SFC requires careful consideration to ensure all required functions are executed when required; its use may also require more sophisticated runtime monitoring than required where SFC is not used. The implementation of data-type checks when creating or translating programs and the adoption of a library of 'approved' function blocks help minimise potential errors.

A language subset should be defined, avoiding: execution control instructions that could generate unintentional looping and execution omission, functions having a potential for precision reduction, and functions not unambiguously defined. Complex function blocks with significant memory elements need careful consideration.

The function library requires management, with procedures and tools for the creation, adoption and maintenance of approved functions.

Further strategies for the avoidance of programming errors rely on procedural methods, these could include:

- Adoption of a suitable application development lifecycle with a staged approach,
- Definition and application of coding standards covering:
    - Modularisation and partitioning of programs,
    - Minimise coupling between programs and modules,
    - Ensuring functional blocks are readily testable,
    - Defensive programming measures,
    - Minimization of program complexity,
    - Ready identification and traceability of safety function(s).

If adequate security measures for program creation, modification, download, etc. are not provided by the toolset, procedural measures must be defined.

Detection of programming errors beyond its initial creation relies on V&V activities. Program review and simulation stages should be conducted. Off-line simulation should be considered a basic requirement for a third party toolset. Simulation should be test case based, verifying that the program provides the required functionality and not undesirable actions. Testing of all permutations of input sequences and internal (application) states is generally impractical; test cases need to consider possible EUC conditions, sequence of conditions and system fault scenarios. These cases are also candidates for later system integration and acceptance testing.

Program storage and retrieval

Program storage and retrieval has the potential to introduce errors. The development environment is a commercial platform (Windows); ensuring files are saved and retrieved correctly requires either the evaluation of the commercial platform itself or measures to ensure that potential errors are detected. Evaluation of Windows, or its filing system, without visibility of the source is at best difficult. For third party IEC 61131 toolsets, the measures to detect filing errors are limited unless already an integral part. Compile time checks already implemented will detect certain errors. The necessary program verification/review forms another diverse measure; alternative display methods should also be used to reduce the risk that visual library errors mask program errors. Further error checking will be provided by later test and verification activities.

Translation/compilation

The translation, or compilation process is another potential error source. The output is typically not in a format lending itself to human inspection. The translation process typically removes certain information that could allow (re-) creation of the source form. A diverse translator could be used, however this would rely on access to the internal source format and detailed knowledge of the translator operation. The most expedient risk mitigation is therefore often a combination of 'proven-in-use' evidence and testing of the resultant object code.

The choice of third party IEC61131-3 tool needs to consider this proven-in-use requirement. Ensuring that the tool has a considerable installed base, evidence of use, and traceability between versions, updates and bug fixes.

Download

It is necessary to ensure that the download process replicates the program within the target without error. Upload and comparison or checksum verification methods can generally be readily applied.

Alternatively, the communications with the target may be specifically generated for safety applications, providing reliable transport of the necessary program(s) and files.

Program execution

The program object may be compiled machine code or interpreter code. An interpreter may be specifically written for the application, provided the command set is published. Alternatively, a combination of testing, run-time monitoring and proven-in-use evidence may be used.

Run-time monitoring, should monitor execution timing, execution sequence, memory access protection, interpreter code validity, and interpreter process stack monitoring. Interpreter state machine monitoring should be considered to ensure independence between functions. The functions executed by the interpreter must be comprehensively tested. Methods outside the interpreter must be implemented that ensure that the EUC is placed in a safe state should an error occur.

With machine code a subset of these error checks are possible. Increased emphasis is placed on the compiler. The measures identified earlier would almost certainly be considered not adequate.

It is desirable to minimize the complexity of the interpreter (or machine code), with hardware handling, I/O access, communications, etc. handled independently.

Interaction with other software

The runtime environment inevitably includes other software. It is important to provide measures to ensure their independence. Well-defined interfaces must be used. Memory protection measures should be implemented, providing protected code space and independent data spaces preferably for each software process. Appropriate error handling and independent fail-safe action in the event of violation must be provided.

Version management

It is important to ensure that the correct version of the application is installed. Measures are required to be able to identify the executing version, and to be able to recreate the source form should the original be lost. Download of the source files may be appropriate, provided methods of ensuring the object file correspond to the source are implemented.

The development toolset and target environment may undergo change; measures ensuring compatibility between the toolset generated object code and the target environment (e.g. interpreter) are required.

Changes to the application program need to be carefully controlled, on-line changes in particular. Changes will require re-test of the modified elements; impact analysis identifying other programs and modules potentially affected by the change should be implemented. Similarly, methods to ensure that programs and modules that should not be changed because of the update are identical to their previous version are required.

Summary

A commercial IEC 61131-3 package enhanced with risk reduction measures, a subset described in this paper, has been successfully certified for use in safety applications for a number of systems. Many of the risk reduction measures have been automated, resulting in cost effective safety-related application development. The use of a COTS package substantially reduced time to market and development costs, and provides on-going benefits for up-to-date products with considerably reduced sustaining cost.