UNIVERSITY OF OSLO
Department of Informatics

A Safety Layer for Foundation Fieldbus

Henrik Tobias Brodtkorb

Cand. scient. thesis

6th of May 2001

Summary

The focus of the work of this Cand. scient. thesis has been placed upon understanding and developing a concept for using digital communication in safety-critical applications. It is increasingly common to use programmable technology in a safety-critical control system. These new software-based components are in many cases replacing existing hard-wired and analogue components that have safety-critical functions. Implementing these software-based safety-critical systems require more in-depth methods and concepts than what traditionally has been used in software engineering.

I have concentrated my studies around problems concerned with using fieldbus technology between the subsystems in a software-based safety-critical system. During the work on this thesis I have had to acquire a substantial amount of knowledge about subjects not previously covered in my studies. I have gained knowledge about industrial process control systems, safety-critical systems, international IEC standards for safety systems, various fieldbus technologies, coding theory and hardware related programming. The basic knowledge required to appreciate the contents of this thesis is presented as background information in the introductory chapters. One of the main goals of this thesis has been to analyse and find out if it is possible to implement a safe communications protocol for Foundation Fieldbus fulfilling the stringent requirements of a SIL 3 application. My studies are based on a concept of a general communication protocol called a “Safety Layer.” A safety layer defines methods for increasing the probability of detecting errors that may occur between two communicating fieldbus devices. The safety layer’s objective is to make the transmission “safer” between two nodes in a fieldbus network. This involves enabling the communication parties to determine that messages have the right value, they are sent in the correct sequence and to the right time and have correct origin/destination. In my thesis I have further developed the basic idea behind the safety layer in a general sense so that it can in principle be used on top of any communication protocol to provide safe communication.

This study has furthermore been extended with an implementation of a safety layer prototype for Foundation Fieldbus. The purpose of this implementation is to demonstrate how Foundation Fieldbus can be made safer and that the concept and ideas of the general safety layer are feasible. The protocol incorporates a method for enabling two-way communication between two function blocks and new CRCs for improved error detection. The choice of generator polynomials for these CRCs has been based on probabilistic considerations. The safety layer also specifies a communication mode that sanctions transmissions of amounts of data larger that the maximum allowed transmission unit. The implementation of the safety layer is unique in its kind. Similar work has never before been done for Foundation Fieldbus.

Acknowledgements

I would like to thank my supervisor at the University of Oslo, Department of Informatics, Associate Professor Tor Skeie for his support and guidance throughout the whole process of writing this thesis. I would also like to thank Scientist Lars Lidström at ABB Corporate Research AS in Norway for giving me a flying start on my project and for his assistance during the work. Additionally I would like to thank Jan Endresen at ABB Corporate Research AS for his help on problems related to the CRCs. I must also thank my other friends and colleagues at ABB for many rich and helpful discussions and the encouraging words along the way. I am grateful to the support group at National Instruments and Tom Boyd at Fieldbus Inc. for their patience and for the help they offered me on technical problems with the development tools.

Contents

List of Tables

List of Figures

Chapter 1 Introduction

In process control industry there is a movement towards an increased use of fieldbuses to interconnect "intelligent" manufacturing equipment. There are several reasons why to use fieldbus, some of the benefits include lower cost of purchase and ownership (reduced cabling, less equipment) and access to more information available for operations and diagnosis. A fieldbus can most easily be described as a Local Area Network (LAN) specialised for use in industrial environments. A wide variety of networks are available in the industrial automation environment. Each bus is designed for a specific area within the fieldbus market. The most widespread bus technologies are based on international standards. Foundation Fieldbus and Profibus are examples of such buses designed for process industry. It is a coming trend in safety-critical systems, or just safety systems (systems that can be hazardous to man, environment or equipment in the presence of faults), to use fieldbuses with the same benefits as non-safety-critical applications. Traditional safety systems have been based on relatively simple sensors and actuators hardwired to the input/output port of the system controller. The transition from these traditional hardwired systems to the use of microprocessor based systems, represents a paradigm change within the technology of the safety industry. Several new challenges are introduced to the safety systems manufacturers related to the safe communication between the field devices and the system controller. Although software based systems provides high flexibility, their primary disadvantage is the complexity they involve.

1.1 Problem domain

In this thesis I will propose a way of modifying an implementation of Foundation Fieldbus H1 (FF) to make it usable for safety-critical communication. The work on this thesis is based on a technical report called "Using Foundation Fieldbus in Safety Applications," [1] written by Lars Lidström at ABB Corporate Research Centre in Norway. The report is a pre-study identifying possible weaknesses in the Foundation Fieldbus protocol stack. The outcome of this study was intended as foundation of a planned project aiming at using a Foundation Fieldbus H1 network in safety-critical surroundings. The investigation concluded that the standard Foundation Fieldbus H1 protocol stack was not well suited in high-risk applications. In the report the author pointed out weak elements within the FF communication stack related to communication. My assignment has been to further analyse the issues outlined in [1], propose concrete solutions to the problems concerning *safety-related communication* and investigate the possibilities of implementing them in a "safety layer." I have studied issues related to how to make the communication between fieldbus devices more "safe," or actually more reliable. The safety layer shall allow communicating applications in fieldbus devices to know, with a given certainty, that transmitted data has arrived correctly at its destination.

---

[1] Lars Lidström, "Using Foundation Fieldbus in Safety Applications." ABB NOCRC, June 1999

This thesis starts by having a closer look at safety systems and the evolution of process control technology. In the next two chapters I give an overview of the Foundation Fieldbus system architecture and IEC 61508, an international accepted framework for the use of programmable electronic devices in safety-related systems. Succeeding these introductions I address the design and implementation of “my safety layer,” which is the main focus in the remaining chapters.

1.1.1 Safety Systems

Process control is a general term used to describe the many methods of regulating industrial processes. It is a branch of technology that deals with automating, monitoring, and controlling complex processes. In an industrial process there are many factors and parameters that have influence on the process. Some of them can make the process deviate from its original objectives, thus are the operating conditions constantly subjected to change. The job of the process control system is to actively make small alterations to the process according to the process’ objectives. The process being controlled is monitored for changes by sensor devices that provides information about the state of the system. The gathered information is used by the control system to determine how to manipulate the process. The control system calculates feedback signals that are used to manipulate control devices (actuators). The actuators make small alterations to the process and “guides” it according to the strategy chosen to attain the goal for the system. We may, for instance, want to hold the process as close as possible to a given steady state. Often both internal and external conditions constantly affects the process and makes it deviate from the steady state. A control system must constantly generate feedback signals, based on sensor readings, to modify the process in order to reduce this deviation. One of the first installations using Foundation Fieldbus technology is sited in the northern part of Alaska. The fieldbus is used in Arcos’ oil production plant in the West Sak oilfield [2].

Some processes and systems involve risks. Risk is a combination of the frequency or probability of an accident and the severity of the potential consequences. Systems involving high risks obviously have the capability to do harm to personnel, the environment and to expensive manufacturing equipment. These systems are termed hazardous. It is often necessary to install a redundant system that reduces the entailed risk to an acceptable level in hazardous environments or applications. Such “risk reducing systems” falls into a category of *safety-critical systems*, or just *safety systems*. Examples of these systems can be found in aviation, in nuclear power plants, in the oil and gas industries, and in medical devices. The ANSI/ISA-S84 [3] standard defines a safety system for process industry applications as, “A system composed of sensors, logic

[2] “Fieldbus Foundation låter instrumenten styra själva.”Automation, tidningen för modern produktionsteknik, mars 1999 nummer 2.

[3] ANSI/ISA, “Application of Safety Instrumented Systems for the Process Industry,” ANSI/ISA-S84.01-1996

solvers, and final control elements for the purpose of taking the process to a safe state when predetermined conditions are violated." This definition can also be applied to other common systems used for plant safety, such as emergency shutdown systems and safety shutdown systems. Safety systems exist because conventional control systems cannot be depended upon in safety protection applications. Safety systems are critical in a sense that they are absolutely necessary to maintain a certain level of safety. If they fail, critical and hazardous situations are very likely to occur. The ANSI/ISA S91 [4] is a short, two paged, standard that includes a definition of *safety critical*, "a control whose failure to operate properly will directly result in a catastrophic release of toxic, reactive, flammable or explosive chemical." A safety system obviously has direct influence on the safety of its users and the surrounding environment, and failure of this system will cause the inherent risks of the application to escalate to an unacceptable level. This is illustrated in Figure 1-1 below.

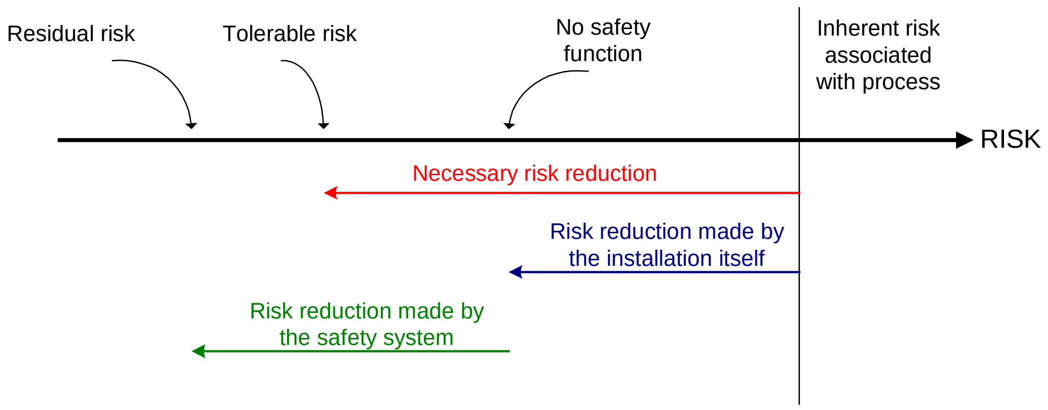

Residual risk
Tolerable risk
No safety function
Inherent risk associated with process
RISK
Necessary risk reduction
Risk reduction made by the installation itself
Risk reduction made by the safety system

Figure 1-1, The best way to minimise risks in plant operation is to design inherently safe processes, and in that way reduce the entailed risk to an tolerable level. Often this approach alone does not fulfil the necessary risk reduction. A safety system must be introduced to reduce the risk down to an acceptable level. If the safety system fails, the application returns to a level of unacceptable risk [5].

It is important to understand the fundamental difference between process control and safety control in industrial environments. A process control system is active and performs continuos computations on input from sensors and sends commands and outputs to actuators. Its task is to automate and regulate the process operation. Safety systems are more of the passive kind and are operating behind the scenes. Safety systems place more strict demands on the level of diagnostics and performance compared to process control systems. Outwards they remain inactive until an error is detected. The safety systems then perform a predefined action, like a shutdown procedure that brings the system to a safe state instead of letting the process continue and possibly lead to a dangerous

[4] ANSI/ISA, "Identification of Emergency Shutdown Systems That Are Critical to Maintaining Safety in Process Industries,"ANSI/ISA S91.01-1995, ISBN 1-55617-570-1.

[5] Based on a translated foil from Tor Onshus' presentation on Safety Critical Systems at ABB NOCRC, March 13th 2000

situation. If a system has a safe state it can adopt when errors are detected, the system is said to be failsafe. Even though a safety system seems idle under normal operational conditions, it is constantly monitoring selected variables in the process control system (See Figure 1-2).

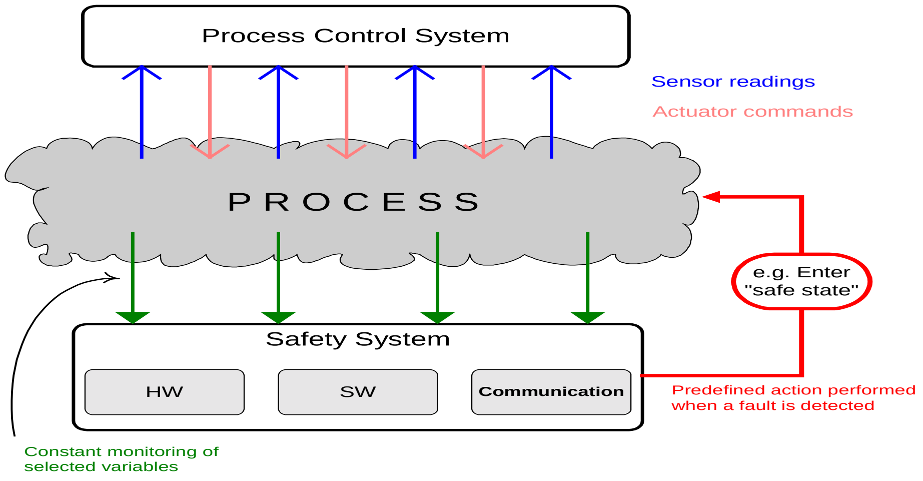


Figure 1-2, The control system regulates the process by collecting information from sensors and sending commands to actuators. The safety system monitors selected variables and its own integrity and is activated if an abnormal situation arises.

Furthermore must a safety system also monitor itself and detect faults in its own operation. The ability to detect faults within itself is very important for safety systems and is defined as the system’s integrity. There exists a number of national and international standards that defines different levels of safety, called safety integrity levels (SIL), that must be assigned to safety systems. The integrity levels are an indication of “how safe” a system is. The safety integrity levels express the probability of an error passes undetected through the safety system. When an error occurs in a safety system, it does not necessary have to pose a threat to the integrity of the system as long as the error is detected and flagged, and some sort of action is taken to handle the situation. It is when an error is not detected and passes unnoticed through the system, really hazardous situations with severe consequences may occur. If this happens the safety system has failed to do its task. This is where the SIL comes into the picture. The SIL provides an estimate for the likelihood of a failure passes unnoticed through the safety system. This issue will be discussed more thoroughly in chapter 3.

The two main fault-handling techniques used for making systems safe are *fault detection* and *fault tolerance*. Fault detection is used during service to detect faults within the operational system. If a fault is detected, an additional mechanism for forcing the system to fail to a safe state is needed. Fault tolerance can be used to prevent system faults from producing system failures. Fault tolerance techniques are designed to allow the system to operate correctly in the presence of faults.

1.1.2 Technological Trends of Safety Systems

There are several technologies available for use in safety systems. The choice of technology may be application dependent and driven by several factors like the level of risk associated with the process, size of the system, size of the budget, maintenance, flexibility, requirements put upon the communication and so on.

One of the earliest technologies used in safety systems was pneumatic, as in process control. Pneumatic systems are still in use today, especially in some offshore installations where systems are required to run without electrical power. Pneumatic systems are most suitable for small applications where simplicity is desired.

Another technology that has been commonly used in safety systems is electromagnetic relay systems based on logical signals. Relays are relatively simple devices with well-understood failure characteristics. They are safe and can meet SIL 3 (See Chapter 3) requirements assuming the right relays are used and that the system is designed properly [6]. Nowadays are relay systems chosen for small and simple applications. They are low-cost systems but can be expensive to maintain. Large relay systems are intricate and pose challenges to engineers whenever changes are needed. All changes made in the wiring must be documented and drawings and documentation must be updated.

Solid-state is yet another technology used within safety-critical systems. Solid-state systems are hardwired and are therefore often expensive to implement. They perform simple relay logic and include features for testing and performing bypasses. Solid-state systems are quite easy to verify because they do not incorporate any software. A single fault within a solid-state logic system usually disables one logic path, thus leaving all the others available.

The most used technology for safety-critical applications today are microprocessor and software-based systems (Programmable Logic Controllers, PLCs). The most obvious advantage of computer-based systems is their processing power [7]. These systems provide high flexibility. Just by modifying the system software its characteristics can be altered completely without requiring changes of the hardware. The primary disadvantage of such systems is the complexity involved. Both the hardware and software is associated with great complexity. PLCs are therefore difficult to verify and validate. However, the advantages of implementing a software-based system often outweigh the problems. Sometimes the costs of implementing a programmable electronic control system are less than that the cost of alternative methods and might also be the only feasible solution.

[6] Paul Gruhn and Harry L. Cheddie, ISA “Safety Shutdown Systems: Design, Analysis and Justification,”

[7] Neil Storey, “Safety-Critical Computer Systems.” Addison-Wesley 1996.

1.1.3 Commercial Dimensions of Safety Systems

Humans have always been concerned about their safety. Safety takes different perspectives depending on the viewpoint we are using. Aspects such as economy, tradition and geographical location of the plant facility are factors that strongly affect safety considerations and the amount of risk we are willing to accept. Risk is always perceived in a subjective manner from a personal point of view. For a machine-operator on the plant floor, safety may for instance involve some sort of an emergency stop procedure. At the corporate level safety involves other concerns. Some corporations may accept incidents others find unacceptable. Economic consideration of safety eventually boils down to put a value on a human life, and this varies between different cultures. Common tools used in the design of safety-critical systems are redundancy and formal methods. Both of these methods are considered to imply considerable expenses. The safer a plant is, the more expensive it will be. The expansion of the plant's safety precautions must be well balanced between the extra expense the increased safety would imply. "For an international supplier of factory automation equipment, safety is critical. Corporations can't afford the risk of large accidents. Neither can they afford the associated bad press, worldwide. It's far less expensive to ensure proper safety, than pay for an accident after the fact."[8] This statement is an example of safety-critical systems playing an increasingly crucial role in process plants. For the last few years the market for safety systems has expanded rapidly. There are several factors that have contributed to this growth. The two most important and recent contributors have been publications of several safety standards and the general public's increased awareness of safety. "Safety is without doubt, the most crucial investment we can make. And the question is not what it costs us, but what it saves." (Robert E McKee, Chairman and Managing Director, Conoco (UK) Ltd.)

In parallel with a worldwide increased density of automation, there has been a continued increase in industrial incidents. Other factors that have played a role in the growth of safety systems include increased use of programmable systems instead of relay or solid-state logic, increased use of safety systems in countries outside North America and Europe. The introduction of low cost safety systems has also contributed to the dissemination of safety systems [9].

Safety standards such as IEC 61508 [10] and IEC 61511 [11] and DIN V VDE 19250 [12], have gained worldwide recognition and are creating more strict safety requirements

---

[8] David J. Bak. "A factory floor 'Safety Net'" - Global Design News, April 1999

[9]ARC Advisory Group "Critical Control & Safety Shutdown System World Wide Outlook" – Market analysis and forecast through 2004.

[10] IEC 61508 – International Electrotechnical Commission (IEC) standard, IEC 61508 – "Functional safety of electrical / electronic / programmable electronic safety-related systems."

for process plants. Therefore, users in the process industry are looking for system suppliers that can offer tightly integrated control and safety systems. Therefore there is little or no doubt that safety is good business in the years to come. In 1999 the total of the worldwide safety market amounted to over $375 million. This amount is expected to grow to more than $500 million by the year 2004 [9]. North America, Europe, Middle East and Africa constitute the major part of the market for safety systems, but as the economy grows in Asia it is likely to think that this part of the world represents a long-term growth opportunity for safety systems.

1.1.4 The Evolution of Process Control Systems

Between 50 and 60 years ago, in the 1940's, most plants relied upon 3-15 psi (Pounds per Square Inch, a unit of stress or pressure) pneumatic signals to control their process. In the 1960's that standard was replaced by the 4-20 mA analogue signal for instrumentation. Both the pneumatic and the electrical standards were analog and unidirectional. Information could only flow in one direction. A more recent change in signal standards was a digital communications format called HART (Highway Addressable Remote Transducer). The HART protocol is an open system solution and provides simultaneous digital communications with the 4-20 mA output. HART has gained widespread acceptance within the process control industry. In 1970 the development of digital processors sped up and led to a more widespread use of centralised computers in process control automation. In the 1980's, so called "smart" sensors were developed. These "smart" or "intelligent" sensors were smart in a sense that they had built-in microprocessors, and became capable of running more complex control algorithms and performing self-diagnostics. Intelligent sensors were interconnected to form networks of field devices. The networks were called fieldbuses. A fieldbus can therefore be described as a network for factory floor instrumentation. Fieldbus is entirely digital, there is no analog signals. This changed the signalling standard in process control from analogous to digital. The introduction of fieldbus included more than a changing of the signal standard. Fieldbus technology provides two-way digital communication between smart field instruments. This allows an expansion of the amount of process data and non-process information flowing both from and to the field devices. Some fieldbus technologies also allow distribution of the control functions to the field devices. Each physical field device performs a small portion of the total process control. This approach eliminates the need for large, centralised and complex computers in control systems.

As the use of microprocessor technology got more common and widespread in control systems, it soon became obvious that it was necessary to formalise the control of smart

---

[11] IEC 61511 – IEC standard entitled "Functional Safety: Safety Instrumented Systems for the Process Industry Sector." This standard adheres to the main attributes established in IEC 61508.

[12] DIN V 19250 (Deutche Industri Normen, DIN) Control technology – "Fundamental safety aspects to be considered for measurement and control equipment."

instruments in a standard. In 1985 the Instrument Society of America joined by the International Electrotechnical Commission, started to develop a standard for two-way, multidrop digital communications that could be used to integrate the smart field devices. This standard was originally outlined to comprise eight parts that each considered different aspects of the communication standard. In 1993, eight years later just one of these eight parts was completed and approved, the IEC61158-2. This part defines the physical layer. The extensive and time consuming work on the IEC standard was perceived by many to be moving too slowly. Some suppliers began using the currently available elements of the IEC standard, assuming that the missing parts would be defined in a certain way. The individual initiatives shown by the member companies lead to an emerging of several proprietary "standards" based on the same physical layer. The physical layer alone is of course not enough to ensure interoperability between different technologies.

The extensive development of a fieldbus standard got complicated and time consuming by the fact that company members of the IEC pushed to have their own product ideas standardised. Some of these products became de facto standards in different regions of the world and within different fields of process control. Finally in 1999 the IEC 61158 standard was complete and sent out for voting by the IEC-members. Company members that developed their own standard that differed from the proposed 61158 standard was negative to accept it as a standard and voted no whether to accept 61158 as an IEC standard or not. After a period of arguing and several re-votes, the disputed fieldbus standard finally got approved. 23 of the 26 member countries of the International Electrotechnical Commission voted in December 1999 in favour of a highly revisited draft of 61158 including eight different busses. 14 years of development and arguing ended in a compromise many characterised as a catastrophe. The fieldbus standard had become an "eight-headed-monster" of a standard that was quite different from the original intentions. The standardisation work, that was started to provide end users with device interoperability, ended up without a result according to its original intentions as vendors kept pushing to promote their proprietary differentiation.

1.1.4.1 The eight parts of IEC 61158

There are today many fieldbus technologies available for industrial environments. Each bus is designed to fulfil the specific needs of an area of applications, but many of the buses have overlapping capabilities. It is convenient to divide the busses into two categories, namely sensor-actuator networks and process level networks. The "sensor" networks constitutes the lowest level of buses. These buses are fast and effective, but with only limited applications beyond relatively simple machine-control. The next level of buses are the process level networks. They provide analog and digital support for more complex instruments and products.

The buses that is included in the IEC 61158 standard are some of the major industrial networks available today, and are as follows:

Type 1 is an all-digital, serial two-way communication system that resembles the “Foundation Fieldbus H1” from Fieldbus Foundation (The Foundation Fieldbus is described more closely in chapter 2). It serves as a LAN for process control and manufacturing automation instruments. The bus distributes the control application across the network. Foundation Fieldbus is an open standard describing digital, serial, two-way communication between fieldbus devices. Devices can be powered directly from the fieldbus and can also support intrinsically safe fieldbuses. There is support for both scheduled- and unscheduled- (user-initiated) communication. Foundation Fieldbus’ main application area is within process control. One thing that makes Foundation Fieldbus different from other technologies is that some parts of the application running in the field devices are standardised. This ensures interoperability between devices from different vendors. Foundation Fieldbus is the only fieldbus that has been implemented as a subset of the original IEC 61158 standard before it was determined that 61158 should include seven other fieldbuses.

Type 2 is ControlNet. ControlNet is a communications protocol that enables users to predict data transmission and guarantee its arrival. ControlNet has a capacity of 5 Mbit/sec and is based on a Producer/Consumer model. ControlNet is highly deterministic and repeatable. Repeatability ensures that transmit times are constant and unaffected by devices connecting to, or leaving, the network. ControlNet supports media redundancy and intrinsically safe options. ControlNet also provides a flexibility in topology options (bus, tree, star) and media types (coax, optical fibre, other) to meet various application needs.

Type 3 is Profibus, an open fieldbus standard and a part of a European Standard called EN 50170. Profibus can be used for a wide range of applications in manufacturing and process automation. Profibus comes in three variations. The most commonly used is DP for discrete applications, followed by PA, which defines the parameters and function blocks of process automation devices, and then FMS for critical, high-speed, complex applications. The physical layer in Profibus-DP is based on the RS485 standard, whilst PA has the same physical layer as Foundation Fieldbus H1. Profibus is a fieldbus network designed for deterministic communication between computers and PLCs. Profibus is a master/slave bus. Most of the master-slave communication is done in a cyclic manner. The functions required for these communications are specified by profiles, a basic set of rules and definitions that are valid within a group of field devices. Advantages: Profibus is one of, if not, the most widely accepted international fieldbus standard. Profibus can handle large amounts of data at high speed. The DP, FMS and PA versions collectively address the majority of automation applications. The disadvantages of Profibus include high overhead to message ratio for small amounts of data, no power on the bus for field devices and slightly higher cost than some other buses.

Type 4 is P-Net. P-Net has been a part of the European Standard - EN 50170, since July 1996. The electrical specification of P-NET is based on the RS485 standard using a shielded twisted pair cable. P-NET is a multi-master bus, which can accept up to 32 masters per bus segment. All communication is based on the principle, where a Master sends a request, and the addressed Slave returns an immediate response. Requests can be of a read or write type. Slaves handle the processing of data and the reception or transmission of frames, in parallel. The processing of a request by the slave is initiated as soon as the first data bytes arrive. The slave does not have to wait until the entire frame arrives before processing begins. P-Net can handle up to 300 confirmed data transactions per second, from 300 independent addresses. The right to access the bus, is transferred from one P-NET master to another, by means of a token. When a master has finished bus access, the token is automatically passed on to the next master, by a cyclic mechanism based on time. Any P-NET module, including a master, can be powered down or connected to or disconnected from the bus, without interfering with the rest of the bus system.

Type 5 is the Foundation Fieldbus high-speed Ethernet (HSE). HSE, High Speed Ethernet, is a 100Mbit Ethernet standard who uses the same protocol and objects as FF H1, on UDP/IP. H1 has many good features that HSE does not have and vice-versa. It is not believed that HSE will replace the H1 bus, rather that the two busses will complement each other. In control system architecture, H1 fieldbus operates at the field level to connect transmitters and the like. HSE operates at a higher level between linking devices and workstations and enables a very large network that is also open at the host-level. There are plenty of benefits to Ethernet, low cost, high speed, many media options, and ease of use. Drawbacks of Ethernet include, Ethernet is limited to 100 meters, which is too short for wiring instruments in the field. Ethernet requires multicore cable, which is too costly for long field cable runs. Ethernet needs a hub, which although cheap can be costly and outsized. A user would need one port for each field device. Ethernet provides no power so you would need additional wires. Ethernet is not intrinsically safe so it can't function in certain areas of the chemical and petrochemical industry.

Type 6 is SwiftNet. This is a producer/consumer technology created to satisfy the Boeing Co.'s need for a synchronous, high-speed (85000 samples/sec) flight-data bus. SwiftNet locks together the local clocks of all fieldbus nodes and therefore the common bus time is inherently available in all devices. The bus provides synchronous global, group, and individual triggers to stimulate device actions. SwiftNet is a very high efficiency / high scan rate, truly-synchronous fieldbus. It minimises and controls jitter. The bus has support for up to 896 bytes of user data. For large, bridged networks SwiftNet support up to 30,000,000 nodes. Timestamping works between any levels of the bridged network. SwiftNet is ideal for any control application that needs high-speed, synchronous data acquisition or control, a large number of nodes and wide-area coverage.

Type 7 is WorldFIP and is a single communication technology for both time-critical data and unscheduled messages. Applications include large distributed control systems down to low-end non-intelligent sensors/actuators. A single communication technology serves each level of control architecture. WorldFIP uses the producer/consumer model with a centralised bus scheduler. WorldFIP is a single consistent technology for all levels of the fieldbus system, combining the worlds of closed-loop control and information-technology on one bus. WorldFIP is designed to provide links between level zero (sensors/actuators) and level one (PLCs, controllers, etc.) in automation systems. WorldFIP provides a deterministic scheme for communicating process variables and messages at up to 1Mbit per second. WorldFIP uses a centralised scheduler that broadcasts a variable identifier to all nodes on the network, triggering the node producing that variable to place its value on the network. Once on the network, all modules which need that certain information "consume" it simultaneously. This concept results in a decentralised database of variables in the nodes and remarkable real-time characteristics. This feature eliminates the notion of node address and makes it possible to design truly distributed process control systems. WorldFIP can be used with all types of application architectures, centralised, decentralised and master-slave.

Type 8 is the Interbus-S and uses a ring topology. Due to that ring structure and because it has to carry the logic ground, Interbus-S requires a 5-wire cable between two devices. InterBus was one of the very first fieldbuses to achieve widespread popularity. It continues to be popular because of its versatility, speed, diagnostic and auto-addressing capabilities. InterBus differs from the other fieldbuses in that it uses a ring topology, not a bus. InterBus has two other advantages, because of its unusual network topology. First, a master can configure itself because of the ring topology. Second, precise information regarding network faults and where they have occurred can drastically simplify troubleshooting. The protocol has minimal overhead and few buses are faster than InterBus. InterBus is most commonly found in assembly, welding and material handling machines. Advantages: Auto-addressing capability makes start-ups very simple, extensive diagnostic capability, widespread acceptance (especially in Europe), low overhead, fast response time and efficient use of bandwidth, power (for input devices) available on the network. Disadvantages: One failed connection disables entire network, limited ability to transfer large amounts of data.

1.1.5 The Benefits of Adopting Fieldbus

Users want benefits when adopting fieldbus technology. They demand more from their control systems and industrial networks. The benefits of fieldbus in a factory are said to be many, and span the life cycle of a plant, from planning and installation through ongoing operations and maintenance. The main benefits include:

Lower installation costs

End users really expect this benefit. Compared to analog devices, fieldbus installations costs are greatly reduced. The use of fieldbus greatly reduces initial plant costs and installation labour. This can be attributed to the ability to pre-configure instrumentation and make adjustments to devices located remote from the control room. A fieldbus allows many devices to connect to one cable. This means that the installation costs are reduced because less wire and fewer intrinsic safety barriers are needed. One of the most spoken benefits of fieldbus is the reduced cost of cabling involved, but the most potential for wiring savings will be in new plants or major retrofits [13]. Connecting multiple field

[13] M. J. L. Ochsner, Ken Beatty. Technical Papers of ISA, Networking and Communications on the Plant Floor – Volume 392, 5-7 October, “Benefits and challenges experienced by Foundation Fieldbus Installations”

devices to a single bus also means reduced I/O and control equipment needed, including cabinets and power supplies.

Additional lower initial costs include

- Fewer, simplified drawings
- Easier control system engineering
- Self configuration with “Plug and Play” functionality
- Simplified commissioning and start-up.

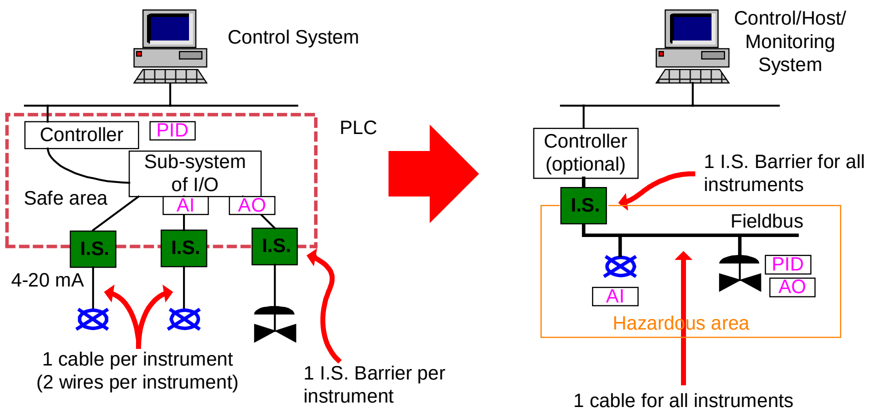

Control System
Controller
PID
PLC
Sub-system of I/O
Safe area
AI
AO
I.S.
I.S.
I.S.
4-20 mA
1 cable per instrument (2 wires per instrument)
1 I.S. Barrier per instrument
Control/Host/ Monitoring System
Controller (optional)
1 I.S. Barrier for all instruments
I.S.
Fieldbus
AI
PID
AO
Hazardous area
1 cable for all instruments

Figure 1-3, A fieldbus system needs only 1 I.S. barrier and 1 cable, compared to the traditional 4-20mA systems that needs 1 cable and 1 I.S. barrier per instrument.

Lower maintenance costs

Maintenance and process management is easier and more effective with fieldbus. Cost reductions are expected due to increased monitoring of process equipment. Lower maintenance cost is a result from increased diagnostics from the individual instruments. The operator's view of the entire process is expanded. This expanded view aids in maintenance because operators have access to more information and diagnosis. The need to send a maintenance person to the field to check a device that might have a problem is decreased. The fieldbus device self-diagnostics notify you when a problem occurs and more information allows preventive maintenance.

Improved performance

Digital communications from the field devices enable more detailed diagnostics directly from remote stations. Because the fieldbus provides better diagnostics, the down time is reduced and productivity and quality increased. Some fieldbus devices may perform multiple measurements, control and computations. That means both increased control over the entire process and the number of transmitters may be reduced

This is a very short and general indication of some of the many advantages fieldbus technology introduces in process control over conventional technology. The value added by implementing fieldbus technology span from a reduced cost of cabling, to making the control system more flexible with the introduction of processing power in the field. During the last couple of years there has been a movement in process-control industry towards an increased use of fieldbuses. The result of an extensive survey of over 700 end-users across 8 major industries throughout Europe points out two main reasons on why customers want to install a fieldbus [14]:

26 % said ‘cost savings’

- less wiring
- lower maintenance
- simplified plant structure
- lower staffing

22 % said ‘facilitation of intelligent field devices’

1.1.6 ProfiSafe

ProfiSafe is an example of a safety system based on microprocessors and software. ProfiSafe is a communication profile for safety-critical applications developed by the Profibus organisation. ProfiSafe is used together with Profibus-DP to achieve a failsafe safety function. If a system is failsafe, it has a safe state it can adopt when errors are detected. ProfiSafe is certified for use in Safety Integrity Level 3 (IEC 61508 SIL 3) applications. The concept of SIL is discussed in chapter 3.

The safe communication provided by ProfiSafe is achieved by using standard Profibus-DP as a transmission system and adding specific safety transmission functions collected in a profile above the protocol layers of Profibus-DP. The ProfiSafe safety profile is designed as a superstructure placed on top of the standard fieldbus. This design allows the use of standard devices and “failsafe devices” on the same bus. The ProfiSafe profile constitutes an entity, or a sort of an application, in a device that communicates with a similar entity in another device. The failsafe profile considers the basic DP-bus to be a “grey channel” it uses to convey messages to another profile. Only profiles of the same type can understand the contents of the safety-related data. Profibus is a Master/Slave bus and the safe profile defines a 1:1 communication between the failsafe devices. ProfiSafe supports only cyclic communication between master and slave.

[14] Datamonitor, “Developments and Customer Opinion on Fieldbus,” July 1999

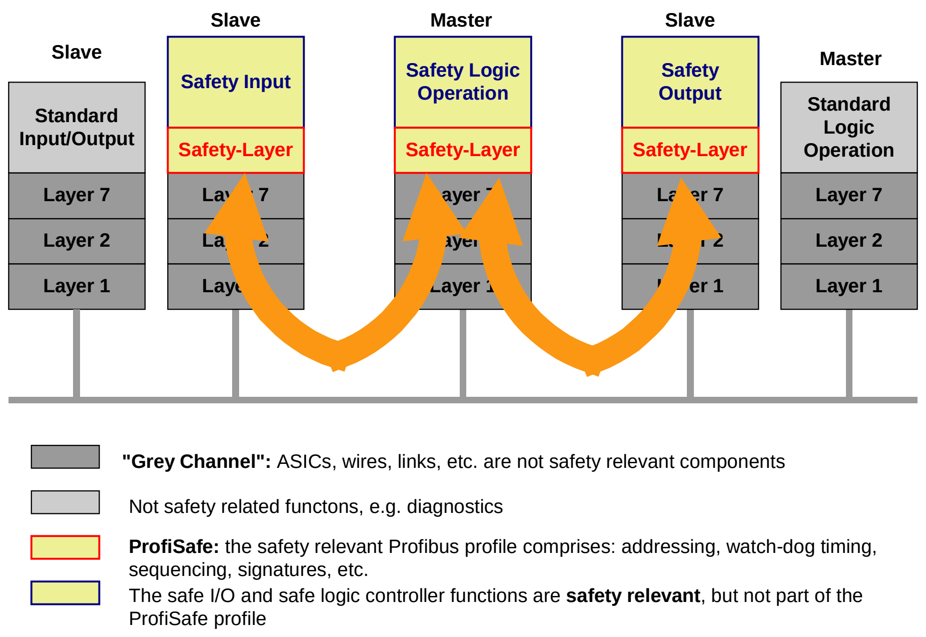

Slave
Standard Input/Output
Layer 7
Layer 2
Layer 1
Slave
Safety Input
Safety-Layer
Master
Safety Logic Operation
Safety-Layer
Slave
Safety Output
Safety-Layer
Master
Standard Logic Operation
Layer 7
Layer 2
Layer 1
"Grey Channel": ASICs, wires, links, etc. are not safety relevant components
Not safety related functons, e.g. diagnostics
ProfiSafe: the safety relevant Profibus profile comprises: addressing, watch-dog timing, sequencing, signatures, etc.
The safe I/O and safe logic controller functions are safety relevant, but not part of the ProfiSafe profile

Figure 1-4, ProfiSafe layer architecture [15]

Error detection is the mechanism used in the ProfiSafe profile to keep the desired level of safety. It is the profiles responsibility to detect communication errors like duplicated frames, loss of frames, incorrect sequence of frames, corrupted data within the frames, delay of frames and correct delivering of frames (addressing). The profile uses information redundancy to validate the communication between two devices. Redundant safety-relevant information is transmitted together with basic process data. The safety-relevant data are embedded in the data field of a basic Profibus-DP frame. A basic Profibus-DP-frame can hold maximum 244 bytes of process data. ProfiSafe reserves 128 of these bytes for safety relevant data. Out of these reserved bytes four or six bytes are set aside as status and control bytes depending on amount of transmitted safety data. Two control bytes are always sent in each frame, one byte for status and one byte for sequence numbering the frames. The remaining four bytes are reserved for a checksum that is generated to protect the redundant safety information. A small amount of transmitted safety relevant data (up to 12 bytes) implies a 16 bit CRC, and the control bytes amount to a total of four. For the transmission of more than 12 bytes of safety data (up to 122) a 32 bit CRC are used, thus six bytes are used for control bytes.

[15] PROFIBUS-DP/PA - ProfiSafe, Profile for Failsafe Technology, V1.0. Document No. 740257

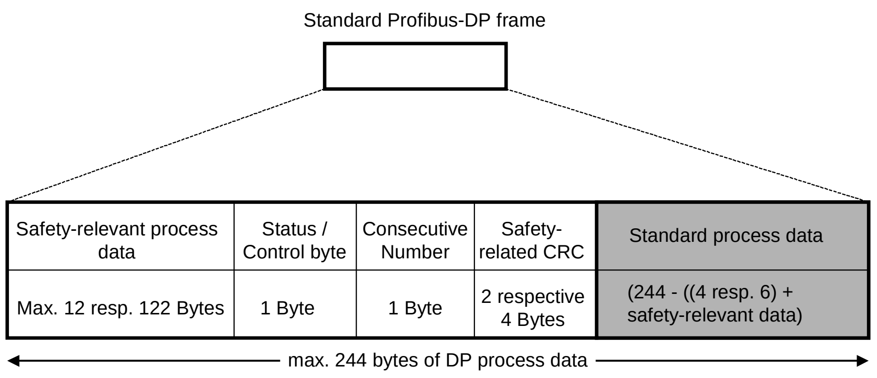


Figure 1-5, ProfiSafe frame structure. A maximum of 128 bytes can be used to hold safety data.

In addition to the redundant information mentioned above, there is implemented a time monitor to check that the safety communication between master and slave is intact.

There has played both technical and economical/political factors in the choice of using FF as a basis for safety-critical systems in this thesis. From a technical point of view is FF considered by many to be the basis of the next generation of process control. It is the most modern fieldbus and it is a subset of the original IEC 61158 fieldbus standard. Siemens is big actor in the process control market and they are using Profibus as “their” bus. There is also a desire from competitors of Siemens and the safety system industry to prevent Siemens to gain dominance and a monopoly in the safety-critical fieldbus market with “their” ProfiSafe profile. The industry would like to have an alternative to ProfiSafe and the competing vendors all want to get a share of the market. Competition among vendors is always an advantage for the end users.

1.2 Problem specification

The main goal of this thesis has been to analyse and propose an implementation of a safe communication protocol for Foundation Fieldbus (FF). This task can be formalised in a high level question to set the scene for this thesis:

“How can we make FF suitable as a communication system in a SIL 3 application?”

Recall from Figure 1-2 that a safety system can coarsely be divided into three parts, namely hardware, software and a communication profile. In each of these parts errors can occur and each part individually contributes to the total probability of failure of the safety system.

In this thesis I will consider aspects making the communication between devices in a FF network “safer,” frankly more reliable in a one-to-one communication.

I will not consider other aspects of safety-critical systems concerned with system requirements, hazard analysis, implementation, verification and validation of system specification and design, configuration, maintenance, and so on. I will neither address issues directly related to security, even though safety and security are two qualities closely related. Both qualities deal with threats or risks. Safety deals with threats to life or property and security deals with threats to privacy, e.g. corrupt operators making unauthorised access and deliberately jamming the network. The most important difference is that security focuses on malicious activities, whereas safety is also concerned with well-intended actions.

A safety system requires that data can be validated in both a value- and a time-domain. This applies to all parts within the system. For every transmission of data, the safety layer must be able to check if the transmission was carried out at the right time and with the correct values. For the devices to determine the validity of the data they receive, they must ascertain whether the data are correct or not, this can be done by answering the following questions:

- ❑ Are this data I just received from the right sender?
- ❑ Does the data I just received have the right value?
- ❑ Is it correct that I received data at this point in time?
- ❑ Does the data have the right sequence number?

Systems for the safety-critical industry must not only be safe, they must be shown to be safe. Before any new safety systems can go into service and to achieve confidence among customers, they must be certified according to international safety standards by a certifying body (e.g. TÜV in Germany). IEC 61508 is an international standard that provides a framework for the use of programmable electronic devices in safety-related systems. The IEC 61508 standard is considered as one of the most useful contributions to industrial safety by suppliers and system integrators of safety systems today. The IEC 61508 points out among other things all known failure modes for communication systems, such as software failures in the communication protocols, hardware failures in transmitters, receivers, gateways and routers and sporadic disturbance of the transmission path.

The following failure modes are introduced in IEC 61508:
1. Data corruption
2. Corruption of sender and/or receiver addresses
3. Transmission of data packages at the wrong point in time
4. Wrong sequence of packages

In the study “Using Foundation Fieldbus in Safety Applications” [1], the author uses these four failure modes and investigates how they relate to the standard FF communication stack. He concludes that each mode applies to FF in a safety critical application in the following matter:

1. Data corruption: “The main problem of using “standard” FF protocols for safety related communication is the frame check sequence.” The error detection mechanism

incorporated in FF is not strong enough to be used in SIL 3 applications, and can not fulfil the validation of data in a value domain.

2. Corruption of sender and/or receiver addresses: Possible errors here can be

    - Wrong sender and/or receiver addresses
    - Multiple receiver addresses (identical addresses on different devices)
    - No address match

    There is only one of the three communication modes offered in standard FF that supports full addressing scheme, (e.g. both the sender and the destination address is included in the transmitted data frames). This mode is not time-deterministic and violates the demand to validate data in a time domain.

3. Transmission of data packages at the wrong point in time:
    - No transmission
    - Delayed transmission

    To be able to validate data in a time-domain, time-determinism is essential in safety systems. The only communication mode in FF that is synchronised with a clock, does not include sufficient addressing information.

4. Wrong sequence of packages: The only channel that supports sequence numbering of data is not truly timed.

"Using Foundation Fieldbus in Safety Applications" [1] concludes that there is none of the communication modes in FF that alone supports the required amount of redundant data to make FF applicable for safety critical communication. The concept of communication modes in FF is explained more closely in chapter 2. The pre-study does not design a safety layer, it only identifies problems related to safe communication and FF and points to possible means to handle the failure modes. The report leaves some issues open. Studying this report, questions, of more or less concrete manner related to design and implementation emerged. I structured the questions and tried to concretise the loose ends by forming a set of questions that formed the foundation for this thesis:

- How can I implement a "Safety Layer" without tampering with the protocol stack itself? Where in the FF system architecture does a "Safety Layer" fit in?
- How can we improve the CRC and the error detection mechanism to achieve a SIL 3 performance? Which CRC generator polynomial can provide the required error detection properties?
- How do we transmit the safety-related data between devices? Which of the three available communication channels do we use? What is the maximum amount of data allowed to be sent in one transmission?
- How can we implement the concept of acknowledgement in a unidirectional communication channel?

These questions served as a starting point for my work on proposing an implementation of a safe communication protocol for FF.

1.3 The structure of this thesis

The structure of the rest of this thesis is as follows:

Chapter 2 gives an overview of the Fieldbus Foundation and its technology. The Foundation Fieldbus communication stack is compared to the ISO OSI Reference Model before the system architecture is examined more closely. The content of this chapter serves as background information required for comprehending the technical discussion carried out in chapters 5 and 6.

Chapter 3 gives a presentation of the IEC 61508 standard and some important concepts related to safety are introduces and defined. IEC 61508 is an international accepted standard for use of electronic programmable equipment in safety critical applications.

Chapter 4 provides more necessary background information and discusses safety-critical communication and different failure modes the safety function must handle. Chapter four also looks at risk considerations for safety critical communication.

Chapter 5 gives a detailed description of the design of the safety layer I propose. This chapter describes how a safety layer can be inserted into the Foundation Fieldbus system architecture. Further are the safety layer’s countermeasures of the failure modes discussed. Within this part the concept of CRC is shortly introduced.

Chapter 6 discusses the safety layer prototype described in Chapter 5, pros and cons. There is also a section that deals with the work-process with this thesis, practical problems I have encountered on the way.

Chapter 7 gives a conclusion for the work and points out issues to be further investigated.

Chapter 2 Foundation Fieldbus

A large part of the work with this thesis has consisted of building up knowledge about the Foundation Fieldbus and understanding how the different components of the technology function and interacts. Chapter two summarises the Foundation Fieldbus System Architecture [16] and gives a brief introduction to the most important concepts of the Foundation Fieldbus technology. This chapter provides necessary background information for the understanding of the main part of my thesis.

2.1 FF - Fieldbus Foundation or FOUNDATION Fieldbus?

The Fieldbus Foundation is a not-for-profit corporation established in September 1994. The organisation is dedicated to a single international, *interoperable* fieldbus standard. This work involves increasing the industry acceptance of *interoperable* fieldbus technology, support the development of *interoperable* fieldbus standards and facilitate the development of *interoperable* fieldbus products.

Since the Fieldbus Foundation was created, it has grown to over 120 member companies. The members mainly consist of a mixture of the world's leading controls and instrumentation suppliers and end users. In March 1996, the Fieldbus Foundation released its own interoperable fieldbus technology called “FOUNDATION™ Fieldbus H1.” This technology was developed using ISA and IEC standards to ensure openness, and to move away from vendor specific and proprietary systems. The H1 fieldbus specifically targets the need for robust, distributed control in process control environments. The H1 fieldbus provides “intelligent” field- and control-devices with a digital, two-way communication link among.

Throughout the remainder of this document FF is used as an abbreviation for both the foundation and it’s technology, depending on the context it is mentioned.

2.2 Foundation Fieldbus vs. the OSI Reference Model

The Foundation Fieldbus technology is based on the general communication architecture for the interconnection of open systems defined by the OSI Reference Model [17]. The Foundation Fieldbus model has been optimised, compared to the OSI Reference Model, to meet the special requirements set by the needs of a fieldbus system. The FF model

[16] FOUNDATION™ Specification. System Architecture. Document FF-800, Rev. 1.3, May 8, 1998.

[17] ISO 7498 – International Standards Organization (ISO) Open Systems Interconnect (OSI) Reference Model (RM)

constitutes only a subset of the seven-layered OSI model and includes three of the layers defined in the reference model. The other four layers from the OSI model have been ignored primarily for performance reasons. The functionality of the omitted layers is not needed in a fieldbus system and has therefore been ignored. Starting from the bottom and working up, the FF architecture adopts the IEC/ISA-approved physical layer (layer 1). This physical layer is also used by other fieldbus standards. The physical layer handles the transmission of raw bits over the physical media. The second layer defined in the FF model is the data link layer. This layer collects a stream of bits from the physical layer into frames. The third layer defined in FF brings us to the seventh layer in the OSI model, the application layer. This layer provides communication services and a model for the applications to interact over the fieldbus. The intermediate layers (layers 3 - 6), the network-, transport-, session- and presentation- layer, which are normally associated with non-time critical and general purpose applications (such as routing and forwarding), are not needed in a LAN (such as a fieldbus) and has therefore been removed from the FF architecture (See Figure 2-1).

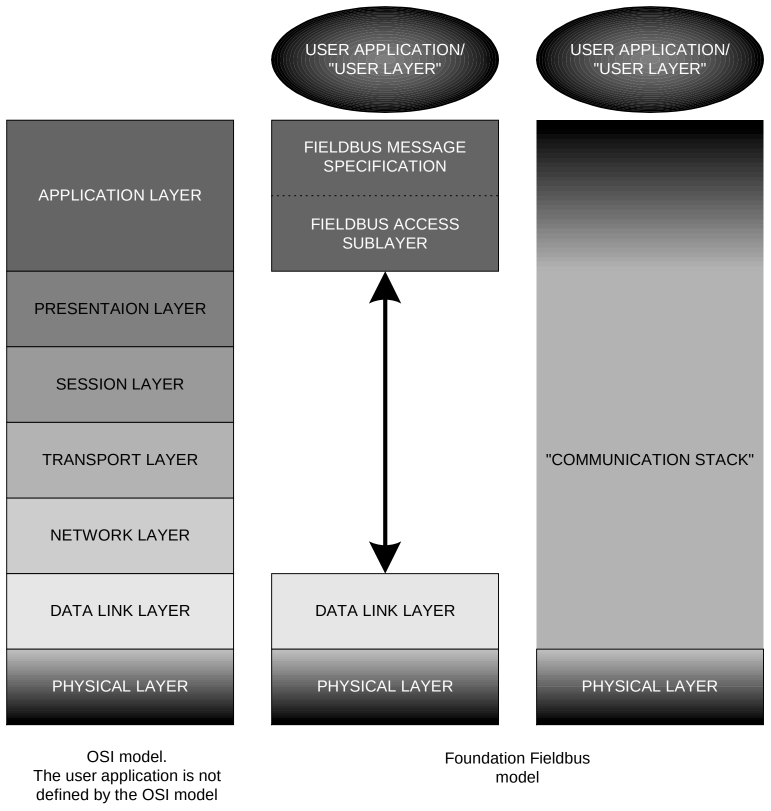


Figure 2-1, The layered OSI communication model compared to the Foundation Fieldbus model

The Foundation Fieldbus system architecture also defines an eighth layer, a “user layer.” This layer defines a standardised way for FF users to interact. The user layer provides guidelines for how the fieldbus device applications shall be designed and implemented to ensure interoperability. The user layer is described more closely in this chapter after the section handling the FF communication stack.

The fundamental purpose of this layered system architecture, is to provide *Virtual Communication Relationships* (VCRs). VCRs are communication channels between applications in the devices that are interconnected to form the fieldbus network. Their properties and functionality are discussed later in the section describing the application layer.

2.3 Foundation Fieldbus System Architecture

The Foundation Fieldbus application model is based on object-oriented programming concepts. This object-oriented approach is adopted to simplify the understanding of the different fieldbus components and their functionality. By using object orientation the architecture can be divided into naturally, logical and more manageable parts and concepts.

A Foundation Fieldbus network is a distributed system composed of field devices and control/monitoring equipment. In Foundation Fieldbus, the network is said to be the control system. The total system operation is distributed over the physical devices. FF allows the devices to perform control functions. Each device performs a small portion of the total process control operation by running small applications. These control functions are grouped in *function blocks*. Function blocks are an abstraction of software that models elementary process control functions. The function blocks represent basic automation functions, such as analog in- and output and PID (Proportional, Integral, Derivative). Function blocks can be combined together into Application Processes. These function block application processes represent actual automation functions to be performed by the process control systems. The automation functions may reside in a single fieldbus device, or may be distributed across several of the devices connected to the FF network (See Figure 2-2). The concepts of function blocks are discussed more closely in the "User Layer"-section of this chapter.

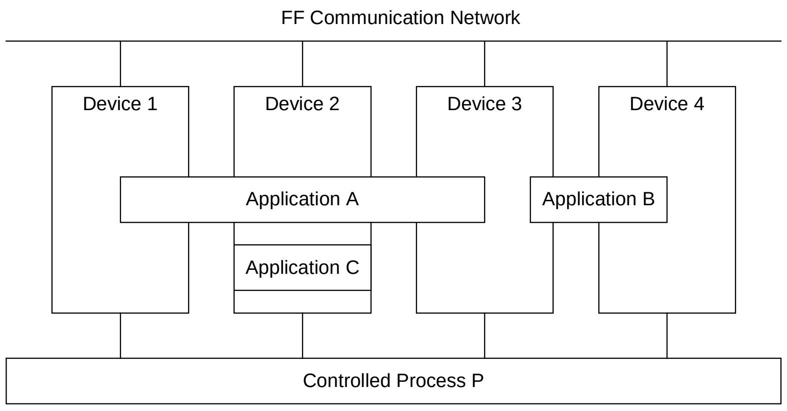


Figure 2-2, Distribution of the process control Application Processes among the field devices. Applications A, B and C control process P. Applications A and B are distributed over respectively devices 1, 2, 3 and 3, 4. Application C resides in device 2.

There exist three types of FF devices, master devices, basic devices and bridges. The master device controls the communication on the bus. This device cyclically issues a token to the other devices on the bus one by one and allows them to use the bus to communicate. All devices connected to the same fieldbus have a common sense of time. The execution of the device-applications is time-critical. The function blocks can be scheduled to execute their elementary process control functions in a determined sequence and publish the results on the bus at a determined time. Function blocks in other devices may use the published results.

The key components of the FF system architecture are shown in Figure 2-3, and each component is described in the following sections:

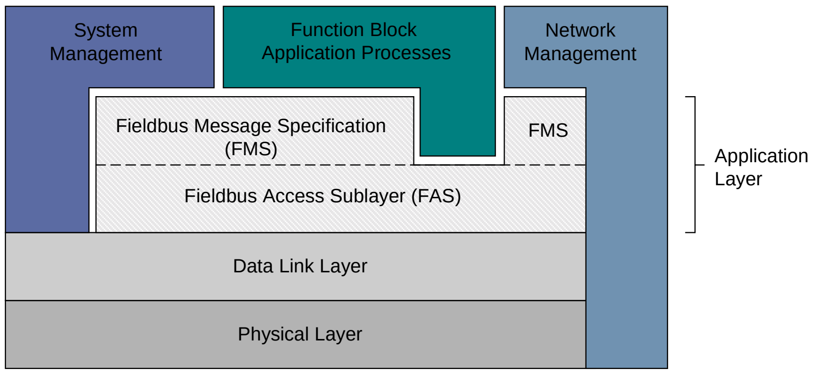

System Management
Function Block Application Processes
Network Management
Fieldbus Message Specification (FMS)
FMS
Fieldbus Access Sublayer (FAS)
Application Layer
Data Link Layer
Physical Layer

Figure 2-3, The key components of the Foundation System Architecture. The application layer is divided into two sublayers, Fieldbus Access Sublayer (FAS) and Fieldbus Message Specification (FMS) (ref. section 2.3.1.3 Application Layer).

The passing of data messages between entities connected to the fieldbus is described in the communication stack.

2.3.1 *FF Communication Stack*

2.3.1.1 Physical Layer

The Physical Layer is composed of the signalling protocol used to transmit the data at the physical medium. The physical layer protocol described in the Foundation Fieldbus H1 standard is defined by IEC 61158-2 [18]. This layer handles the transmission of raw bits over the communication cable and provides services to the data link layer. The transmission speed on the Foundation Fieldbus H1 network is 31,25kbit/s. This speed has been specified to support intrinsically safe environments. Devices that are intrinsic safe have been designed specifically for use in hazardous environments. The electrical power

[18] IEC 61158 – International standard for fieldbus for use in industrial control systems, IEC 61158-2/ISA-S50.02-1992 Part 2 Physical Layer Specification and Service Definition.

an intrinsically safe device use is below the level of power required to set off an explosion within a given hazardous area. In addition, "intrinsically safe"' products are incapable of storing large amounts of energy which might spark an explosion when discharged. An intrinsically safe barrier is placed between the power supply in the safe area and the intrinsically safe device in the hazardous area (See Figure 1-3).

The physical layer uses the Manchester (Biphase-L) technique to encode the physical signals that are to be sent. The Manchester technique combines the clock signal in the serial data stream to create the fieldbus signal. A positive transition in the middle of a bit time represents a logic “0” and a falling edge represents a logic “1” (Figure 2-4).

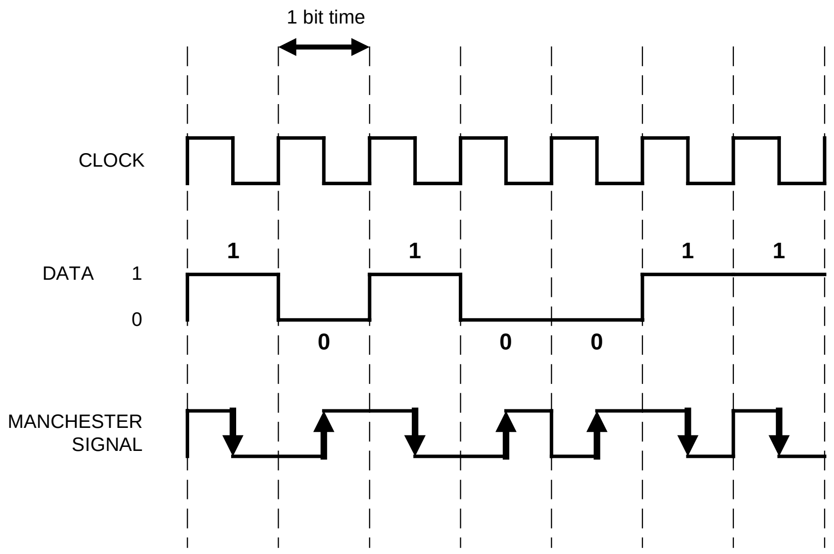


Figure 2-4, Manchester encoding. The signal is self-clocking, the clock is combined with the data stream to create the signal.

Special preamble characters are defined to make the receivers able to synchronise its internal clock with incoming signals. Start- and end-delimiters are used to mark the boundaries of messages. It is only the data part of the physical layer frame that is encoded using Manchester encoding. This makes the data part of the frame stand out from the preamble and delimiters.

The interchanging of data between devices is half-duplex, only one bit stream can be encoded on the fieldbus at a time; thus only one device can access the bus at a time. This means that a device can transmit and receive data on the same media, but not simultaneously. The devices must alternate using the bus by sending the messages synchronously.

There are two power options for the FF devices, self-powered and bus powered. The self-powered devices draw their power from an external source. The bus-powered devices draw their power directly from the bus. A power supply is connected to the network and the bus-powered devices require only the fieldbus cable for power supply and communication.

2.3.1.2 Data Link Layer (DLL)

This layer provides the means to establish and maintain basic communication services between field devices. About 90 % of the fieldbus communication functionality lies within the Data Link Layer (DLL). The DLL specification defines three types of fieldbus devices:

- **Basic Device**
  All devices on the fieldbus have basic device capabilities, this includes receiving the token and responding to it. Once a device has received the token it has the right to publish data on the bus. Only one device may hold the token at a time, thus only one device can access the bus at a time.

- **Bridge**
  A bridge device is used to interconnect fieldbuses to create larger networks.

- **Link Master Device**
  A link master device has the capability to become the Link Active Scheduler (LAS). The LAS is an entity in the DLL that controls access to the bus. There are two types of bus accesses offered, scheduled (cyclic) and unscheduled (acyclic). The scheduled bus access is used for transmission of data that is part of the control strategy. Under normal operation the devices cyclically access the fieldbus to exchange data needed to control the process. Unscheduled traffic on the bus consists of diagnostics from devices to operation interfaces and configuration information from a host system to the devices.

The LAS maintains a list of transmit times of all data buffers in all devices on the bus that needs to be sent on a cyclic or scheduled basis. Each time a device is scheduled to send data, the LAS sends a “Compel Data” message to the device. The device can now gain access to the bus and transmit the contents of its buffer. The “Compel Data” message forces a device to publish the data in its output parameters. When the bus is idle between scheduled transmissions, the LAS send a token to all the devices on the bus to grant devices permission to send unscheduled data. The device that holds the token is able to send data on the bus until it is finished or until the maximum token holding time has expired. The LAS keeps the addresses of the devices that respond correctly to the token between scheduled traffic in a “Live List.” The LAS is also responsible of detecting new devices that are connected to the bus. New devices can be connected to the bus at any time. The LAS periodically issues a “Probe Node” message to all of the addresses that are not in the “Live List.” If a node responds to the “Probe Node” message the LAS adds the node’s address to the “Live List.”

There may exist more than one link master devices in the fieldbus network, but only one maser device can function as the LAS at a time. During start-up of the network, there is a bidding procedure between all of the link master devices on the bus for becoming the LAS. The bidding procedure is also started if an error is recognised in the existing LAS by the other master devices on the bus. A new master device is selected to become the new LAS in the network.

The LAS' five primary functions are:

- "Compel Data" message scheduling: The highest priority function of the LAS is to maintain a list of transmit times for all the buffers in all devices that are transmitted cyclic. On precisely timed intervals the LAS sends out a "Compel Data" message to a device, that forces the device to publish the contents of its buffer onto the bus.
- Token passing: Between the scheduled network traffic the LAS sends the token out to the devices in a round-robin fashion to allow unscheduled transmission of data.
- "Live List" maintenance: The LAS monitors the devices and check if they fail to use or return the token. Devices that do not respond to the LAS are removed from the list of active devices, the "Live List."
- Periodically distribution of data link time and link scheduling time. Every device in the network periodically receives a message from the LAS containing the correct time. This is important because every scheduled communication and executions of user applications are based on this time.
- Checking for new devices on the fieldbus. The LAS periodically probes the unused addresses on the fieldbus to check if any new devices have been connected to the bus. If a device responds from a previously unused address, the LAS add the device to the "Live List."

2.3.1.3 Application Layer

The FF application layer is composed of two sublayers, Fieldbus Access Sublayer (FAS) and Fieldbus Message Specification (FMS) (See Figure 2-1 and 2-3).

The distributed applications in the field devices need to communicate to be able to control the process. The **Fieldbus Access Sublayer (FAS)** uses the scheduled and unscheduled features of the Data Link Layer to provide Virtual Communication Relationships (VCR) to the **Fieldbus Message Specification (FMS)**. A VCR is a preconfigured communication channel. There are three different types of VCRs defined in the Foundation system architecture. Each of them offers various combinations of communication characteristics. These communication relationships or channels meet the various needs the fieldbus applications have, like messaging services and variable reading and writing.

- **Publisher/Subscriber VCR Type**

  The Publisher/Subscriber VCR is a connection-oriented channel that is *buffered* and *unconfirmed*. Buffered means that when new data are generated in a function block, it overwrites old data stored in the buffer. This results in that only the latest version of data is present and transmitted in the network. An unconfirmed service involves that no acknowledgement is given to the transmitter to indicate that the message is received at its destination(s).

The Publisher/Subscriber channel is the only *scheduled* VCR and can be used for *one-to-one* or *one-to-many* communication. The Publisher/Subscriber VCR is used by an application to cyclically publish process data. The publisher's role is to continuously produce data (e.g. output parameters generated by a value measured by a sensor). and broadcast it at cyclic intervals. The subscriber(s) in a communication relationship is (are) configured to listen for data from a specific publisher. The subscriber(s) in the relationship consumes the data produced by the publisher. The scheduled transmission of data from a publisher can be initiated either cyclically by the LAS or by a subscriber (a device that wants to receive the published message) on an unscheduled basis.

The Publisher/Subscriber model is designed to handle the cyclic transmission of data related to the control strategy of the plant operation. It is ideal for its use because it offers a deterministic and efficient transmission of accurate data. No efforts are wasted delivering data to uninterested receivers. The publisher broadcasts its data so that all devices that are interested receive the data simultaneously. No individual adjustments of data to each subscriber are needed since every subscriber receives the data at the same time. Length of time to deliver data is independent of the number of subscribers.

- **Report Distribution VCR Type**
  The Report Distribution VCR is a connectionless data link service that provides the application processes with *unconfirmed* services. This VCR is used for *unscheduled* or user-initiated *one-to-one* or *one-to-many* communication. The Report Distribution VCR is *queued*, which means that messages are sent and received in the same order submitted for transmission, without overwriting previous messages unlike the buffered Publisher/Subscriber VCR. The report distribution VCR type is used to multicast event notifications and trend reports (history information used for reviewing of behaviour of devices).

- **Client/Server VCR Type**
  The Client/Server VCR is a request-response VCR that is always established between two communicating applications. This VCR is *queued* and is used for *unscheduled* communication. The Client/Server VCR is only used in a *one-to-one* communication. A device can be both client and server. The Client/Server VCR provides a *confirmed* service. The transmitter receives an acknowledgement from its peer to indicate the arrival of a frame. The applications use the client/server VCR to upload/download configuration information between a host system and the devices.

The **Fieldbus Message Specification (FMS)** is the upper part of the application layer (See Figure 2-1 and 2-3) and defines a model for applications to interact over the fieldbus. FMS defines the application layer services and specifies the message formats that allow the applications to access remote devices through communication objects. The Object Dictionary (OD) and the Virtual Field Device (VFD) are important in this model. The OD is a structure in a FF device that describes the data that can be communicated on the bus. The OD can be thought of as a lookup table that provides information such as data type about a variable that can be read from or written to in a device. The VFD is a model for remotely viewing data described in the OD. The services provided by the FMS allow the applications to read and write information about the OD, read and write the variables described in the OD and perform other activities such as uploading/downloading of data and invoking programs inside a device.

In addition to the three layers described above, FF includes two management layers or entities called the Network Management and the System Management (See Figure 2-3). Network Management configures the communication stack, loads the LAS configuration, provides performance monitoring and provides fault detection monitoring. System Management assigns unique addresses to the devices on the fieldbus network and provides a function that locates objects in the network. The System Management entity in each device maintains an application clock it uses to independently administer when to execute its applications/functions. There is a device in the network called the Time Publisher that periodically sends a clock sync message to all fieldbus devices. Each device is responsible to maintain the clock between these synchronisation messages. System Management provides for the synchronisation of clocks across the network so that each device shares a common sense of time. The clock synchronisation mechanism may for instance allow consistent time stamping of data throughout the fieldbus network.

2.3.2 User Layer

FF defines a special User Layer that ensures interoperability by using function blocks and a standardised description of the blocks in a device called Device Description. The blocks defined in the User Layer represent the functions and the data available in a device. Rather than interfacing a device through a set of commands, like most communication protocols, a FF user interacts with devices through a set of objects and blocks. Device Descriptions are explained in the next section.

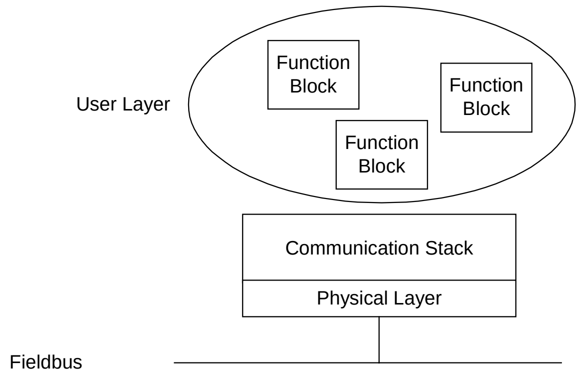


Figure 2-5, The FF system architecture specifies a User Layer. The User Layer defines function blocks that model the user configurable part of a fieldbus system.

2.3.2.1 Function Blocks

We have already heard that application processes can be decomposed into a set of smaller objects called function blocks. Function blocks are the core components in a control system. They can be thought of as an abstraction of software that models repetitive and time-critical functions. The function blocks describes a common way to define basic and elementary automation functions, such as input and output, and other algorithms that is necessary in process control systems. The Fieldbus Foundation has defined an initial set of 10 function blocks. These definitions meet basic measurement and control requirements of a broad group of control systems and instrumentation suppliers rather than a single entity. It is expected that these function blocks form the basis for at least 80% of the function blocks implemented in FF H1 networks [19]. An additional set of function blocks are also defined for more advanced applications.

Function blocks model repetitive and time-critical functions as parametric algorithms. When a function block executes it takes its input parameters as argument, run the control algorithm to completion and produces a set of output parameters. Function blocks are invoked repeatedly, based either on a schedule or on the occurrence of an event. A function block is defined by its I/O- and contained-parameters, and by the algorithms that operate on these parameters. Input and output parameters may be used to exchange data through links between function blocks. The contained parameters are used to define the private data of a function block. Although visible over the network, they may not participate in function block links. A loop or a control loop is a group of function blocks linked together executing at a specific rate (See Figure 2-6). Each block is executed at the configured rate in a specific order. Data is transmitted between the blocks along the linkages. It is possible to have multiple loops running at different rates on a fieldbus segment. For instance may one control loop have a loop time of one second and another can have a loop time of two seconds. Even though loops can run at different rates they can send data through linkages to each other. A macro cycle is defined to be the least common multiple of all loop times on a given fieldbus segment.

When a function block is scheduled to execute, its input buffers are snapped. This ensures that they are not changed during execution by some external sources. Once the inputs are snapped, the function block algorithm operates on them and generates output. When the algorithm has finished its execution, the contained parameters are saved for the next execution and the output data is snapped and released for use by other function blocks.

[19] FOUNDATION^{TM} Specification. Function Block Application Process. Part 2. Document FF-891, Rev. 1.3, May 8, 1998.

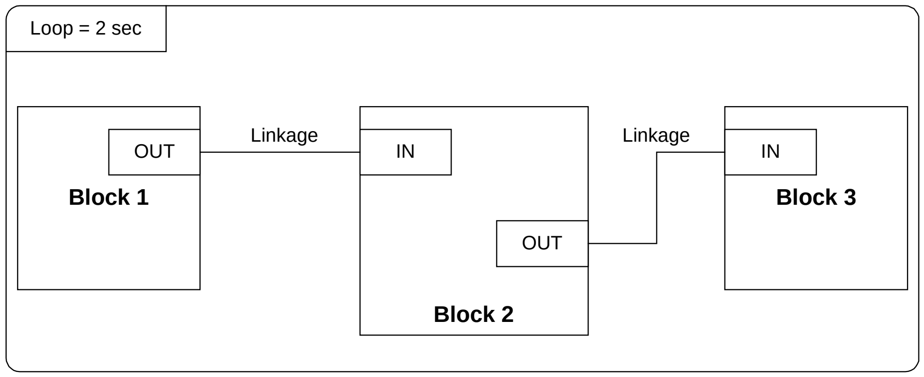


Figure 2-6, Schematic illustration of a control loop. A FF network is made up of devices connected together by a serial bus. The set of functions a device can perform is represented by the function blocks within the device. The function blocks are configured and linked together to control a process.

The object-oriented approach adopted in the FF system architecture allows us to decompose function blocks into smaller objects. The lowest level of decomposition and most suitable level of abstraction are simple variables. Function block parameters are small data structures of simple variables. Simple variables can for example be of the types integer, float and string.

Function blocks represent the user configurable part of a FF system. Blocks are incorporated into fieldbus devices to achieve the desired device functionality, as well as to define a wide range of features and behaviours that must work in a standard way for devices to achieve interoperability. To allow the function block application to be as independent as possible of the real I/O and physical device hardware, the function block application architecture also provides transducer and resource blocks. Transducer blocks are an interface between function blocks and the actual hardware of the device. Transducer blocks disconnect the function blocks from the hardware specific details of a device. Manufacturers of devices must define their own transducer blocks. There is only one resource block per fieldbus device. The resource block describes the device’s general characteristics, such as manufacturer, device name and tag.

Based on a function block’s processing algorithm, a desired monitoring, calculation or control function may be provided by the block. The results from a function block’s execution can be used in a number of ways depending on the type of function block and the configuration of the application. Function blocks can be classified into four categories based on their parameters and behaviour:

1. *Input Function Block* - accesses physical measurements through channel reference to an input transducer block. After processing the value, the results will be provided as an output for linking to other function blocks.
2. *Output Function Block* - acts upon input from other function blocks and passes its results to an output transducer block through channel reference.
3. *Control Function Block* - acts upon inputs from other function blocks to produce values that are passed to other control or output function blocks through output parameters.

4. *Calculation Function Block* - acts upon inputs from other function blocks to produce values that are passed to other function blocks through output parameters.

The Foundation Fieldbus system architecture allows device manufacturers to differentiate their product from competitors by modifying the standard function blocks or defining completely new function blocks. If interoperability is to be achieved between devices from different manufacturers in a setting like this, where different suppliers can build unique features into their devices, then some features and implementations must be consistently implemented. The User Layer gives some general guidelines that should be followed to ensure maximum consistency with other device implementations. If a manufacturer has identified that there is a need to define a custom function block, it should compare its needed variables and functionality with the 10 standardised function blocks. If the variables and functionality of the needed function block closely match that of a standardised function block, then this defined block may serve as the basis for the new block. In this case, it will only be necessary to add the variables that are not included in the defined block.

Device Description (DD) is the key element along with function blocks in the User Layer that enables interoperability. Device Descriptions are used to describe a device and its blocks and their parameters. A DD contains information needed for a control system or a host to understand the meaning of the data in a device, such as name, engineering units and how to display data and information about the relation of a parameter to others. DDs are written in a special programming language called Device Description Language (DDL). The DDL was developed to define a common way to describe FF devices. It is used to describe the standard set of block and parameter definitions as well as user group and vendor specific definitions. Device Descriptions allow the host system to operate the device without custom programming. The DDL describes the semantics of user data. Each device function, parameter or special feature can be described. DDL is a C-like language compiled into a binary form. The DD provides all the information necessary for a control system or host to understand the meaning of device data, including the human interface for functions such as calibration and diagnostics. The Fieldbus Foundation provides DDs for all the standardised blocks. What a device manufacturer typically will do is to prepare an "incremental" DD, a DD which adds additional functionality to an already existing standardised DD.

For further information on the Foundation Fieldbus system architecture refer to the FF specification [20].

2.3.3 Function Block Shell

The Function Block Shell (FB shell) is the interface between the communication stack and the application process. Function blocks residing in different devices use the services

[20] FOUNDATION^{TM} Fieldbus Technical Specifications. Fieldbus Foundation, www.fieldbus.org

of the FB shell to communicate. The interface between function blocks in the same physical device is locally defined for that device. The FB shell is an application-programming interface (API) designed to allow easy implementation of function block application processes. The FB shell is not defined in detail in the FF standard. It is the vendors of FF stacks that are responsible for making the FB shell for their own stack. I have seen FB shells from two vendors (National Instruments and Softing), and as far as I am concerned they look very similar. Since the shell is squeezed in between the function block applications and the FF stack, and both of them are quite rigidly specified in the FF standard, the FB shells cannot be very unlike from the different vendors.

The FB shell has three major purposes. The first one is to provide services and functions that are common to all function blocks (trend processing and notification, alert processing - notification and reception of acknowledgement, establishment and maintenance of function block links, publishing of outputs and subscription of inputs). The second purpose is to insulate the function blocks from the FF stack. The FB shell must pass only those instructions to the function blocks that are specific to the application. The FB shell services all FMS indications and it only passes those read- and write-indications, which require function block application-specific checks. The FB shell passes the function block start indication to the FB application. The third purpose of the FB shell is to configure the FF stack with the parameters and objects specific to the FB application and specific to the manufacturer of the fieldbus device.

I used a FB shell from National Instruments in the implementation part of this thesis. This shell is described more closely in the next section (2.4).

2.4 Development of a FF Device

In this section I will describe the fieldbus equipment I used during the practical part of the thesis. The development environment I used during the implementation of the safety layer was a “Fieldbus Device Starter Kit” from National Instruments (NI). This Starter Kit includes all the components (both hardware and software) related to fieldbus necessary for developing FF devices. The kit consists of two developmental fieldbus round cards, a PC interface-card, a desktop power supply and cabling, a serial programming daughter card and a software tool used for configuring the fieldbus. “A fieldbus round card is a stand-alone card that allows you to interface to a network that complies with the Foundation Fieldbus H1 specification… The fieldbus round card uses the Motorola MC 68331 embedded processor and a programmable 128 KB X 16 Flash to run the stack, FB shell and user applications” [21]. The PC interface-card connects the fieldbus to a host running the fieldbus configuration tool called the Configurator. The Configurator is a graphical environment for creating linkages, loops, setting device addresses, creating and editing schedules and other things related to configuration of the fieldbus communication. The Configurator tool can be seen in Figure 5-10.

[21] National Instruments, “MC 68331-Based Fieldbus Round Card User Manual.”

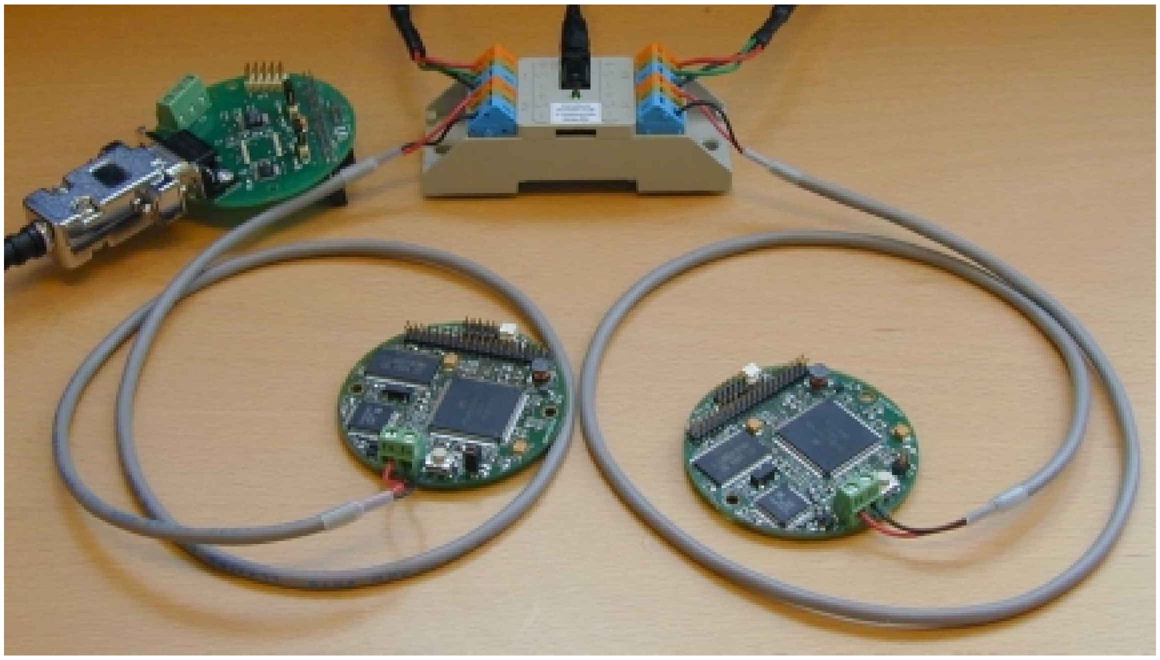


Figure 2-7, The picture shows the Fieldbus “Power hub,” two development round cards in the foreground and the serial programming daughter card is up in the left corner.

The major part of the code running on the round cards has already been written. This code comes along with the starter kit in the form as a linkable library and includes the FF stack and the FB shell. The code that remains to be written for a new field device is parts of the device application. The FB shell from NI is an interface between the stack and the device application. It is this shell a device developer must deal with during implementation. The FB shell provides a set of callback functions. These functions are responsible for handling services such as parameter access, function block execution and alert notification. For example is a callback function called ”`cbExec(`function block`)`“ invoked by the FB shell each time the specified function block is scheduled to run. This callback performs whatever algorithm the developer has coded the function blocks to perform, thus are the majority of the code required for a new device written in this callback. The process of developing a new FF device with the Starter Kit is described in [21] and can be coarsely described as following:

1. Write a device template and a device configuration for your device. A device template contains information about the function blocks and their parameters that constitutes the device application. The device configuration contains the initial configuration of the device, such as device name (identification) and address. (See Appendix 8.4 for both a template and a configuration). The template and the device configuration are converted into C-code so that they can later be compiled and linked with the other code.

2. Write the algorithms in the function block callbacks.

3. Compile, link and download the code to the round cards. After all C-files have successfully been compiled they must be linked with the library containing the FF stack and the FB shell. After the linking process the object bytes must be extracted and converted into a binary image ready to be downloaded into the memory of the target processor. To burn the Flash on the round card the daughter card is connected to the round card and a downloading utility prompts for the binary file to be downloaded.

For more information about this process refer to the round card user manual [21].

Chapter 3 IEC 61508

To day it is almost, if not impossible, for people involved in design and implementation of safety systems to evade an international standard called IEC 61508 – "Functional safety of electrical/electronic/programmable electronic safety-related systems." This standard set out a generic approach for all of the activities involved in the whole lifecycle of electrical/electronic/programmable electronic systems related to safety. The general awareness of safety is getting stronger and the acceptance of accidents to be a part of every day life is decreasing as technology is making progress. This has large implications for companies making safety systems. This chapter gives a short introduction to IEC 61508 and its content is essential for the further understanding of the rest of the thesis.

3.1 Risks in Industrial Processes

There is a vast array of hazards related to industrial processes, such as explosives, toxins, large amounts of energy just to mention a few. Only the imagination sets a limit on the amounts of hazards we can list. The importance of a hazard is expressed by risk. We recall from section 1.1.1 that risk is a combination of the likelihood of an accident and its consequences. The risk related to a system increases if either the likelihood of the system to fail or the severity of the consequences of a system failure increases. Ever since the beginning of the 20th century it has been an increasing concern about industrial safety. More or less continuously efforts have been made to control and minimise risk based on historical expertise. Despite the great advances that have been made in technology over the last years, we are unable to eliminate risk altogether [22].

New hazards evolve as the complexity of installations increases in step with the technological development. New techniques for minimising the risks related to these new hazards are needed. Suppliers of safety related systems for the industry have to engineer their equipment according to a number of local and national user-regulations applying to the area where the facility is to be installed. This has to be done in order to attain the necessary round of approvals a product needs before commissioning. Safety regulations may vary greatly between projects in different parts of the world. For large suppliers operating internationally on a global market, it is extensive work to cope with all the different local regulations. In 1998 a new international standard called IEC 61508 was published. The introduction of this standard should make it easier to obtain worldwide approvals for safety-related equipment and systems. IEC 61508 can be used the world over and might contribute to the reduction of the large quantity of local safety guidelines. The IEC 61508 standard is considered by suppliers and system integrators of safety systems to be one of the most useful contributions to industrial safety.

[22] Nancy G. Leveson. "Safeware. System safety and computers. A guide to preventing accidents and losses caused by technology."

3.2 What is the IEC 61508 standard?

The International Electrochemical Commission (IEC) standard, IEC 61508 – "Functional safety of electrical/electronic/programmable electronic safety-related systems," was developed to provide an internationally accepted basis and a framework for the use of programmable electronic devices in safety-related systems. The IEC 61508 is not a reference book providing a recipe-like description to system designers on how to make safety systems, "First you take a little bit of this, then you add some of that and stir well. If the system is not safe enough, add some more of this and cook for 15 minutes." Instead of using this approach and exactly pinpointing the means to achieve a certain level of safety, IEC 61508 describes general requirements and guidelines on safety-related systems. The standard describes a generic approach for all safety relevant activities for electrical/electronic/programmable electronic systems (E/E/PES). The IEC 61508 has a wide scope, but it places great emphasis on process industry and focuses on the use of computers and programmable electronic equipment and technology in safety related systems. By taking this non-specific approach, IEC transfers much of the actual responsibility of making safe and reliable critical systems over to the system designers and implementers.

IEC 61508 is the first standard that addresses the entire lifecycle of a safety system. The lifecycle describes all activities associated with safety during the entire lifetime of the equipment. The first part of the standard describes the overall safety lifecycle requirements that must be in order to act in a safe manner during the entire lifetime of the equipment. The second and third part concern requirements with respect to the design and development of hardware and software. Part 4 contains definitions and abbreviations. The other three parts are informative.

3.3 Safety Integrity Levels

Considerations of safety have implications for an entire system. All the phases of the system's lifecycle are implicated by safety considerations, from inception to decommissioning. Safety can only be assured by considering all the aspects of a system, including both software and hardware. IEC 61508 states that a plant's safety requirements should be based on analysis of the risks posed by the equipment comprising the plant's control system, the equipment under control (EUC). Unmitigated analysis of all individual components must be carried out to identify all possible faults. Such a thorough analysis may be defined as consisting of three stages, hazard identification, hazard analysis and risk assessment. Risk is usually defined as a combination of two factors, the severity and the frequency of occurrence of an event. In other words, how often can it happen and how bad is it when it does? There are many elements in industrial processes that can be considered potential sources of harm and may pose hazards. Each factor is carrying its own risk and contributes to increase the total risk related to the application. IEC 61508 defines three generic means to reduce the risks involved in an industrial process to an acceptable level (See Figure 3-1):

1. Use an E/E/PE safety-related system.
2. Use mechanical safety devices like relief valves and rupture disks.
3. Use external risk reduction facilities like mechanical changes such as stronger pipes, firewalls and drain systems.

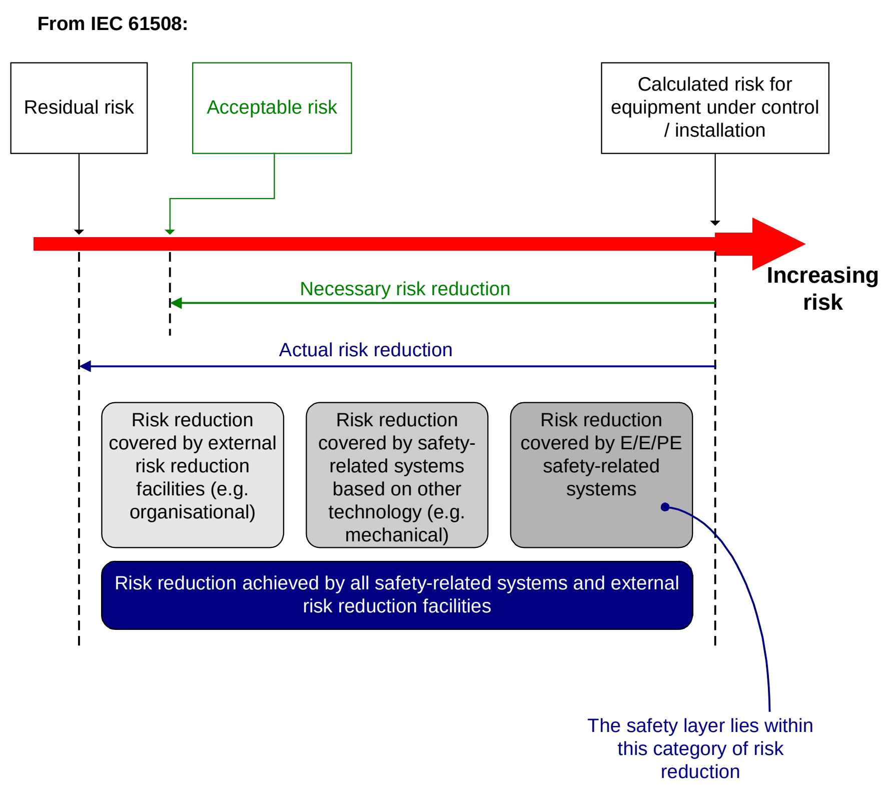

From IEC 61508:
Residual risk
Acceptable risk
Calculated risk for equipment under control / installation
Increasing risk
Necessary risk reduction
Actual risk reduction
Risk reduction covered by external risk reduction facilities (e.g. organisational)
Risk reduction covered by safety-related systems based on other technology (e.g. mechanical)
Risk reduction covered by E/E/PE safety-related systems
Risk reduction achieved by all safety-related systems and external risk reduction facilities
The safety layer lies within this category of risk reduction

Figure 3-1, Risk considerations according to IEC 61508.

The outcome of a risk assessment analysis is a quantitative figure being one of four safety integrity levels (SIL). These safety integrity levels represent a very important concept in the IEC 61508. They reflect the importance of correct operation of a system. The implications of failures differ widely between applications. Some system failures can cause direct and serious harm to people and equipment, others may not. The integrity levels are used to determine the development methods to be adopted when building a safety system. Systems assigned with a high SIL can justify the use of more rigorous design and testing compared to systems with a lower assigned level of integrity. There is a difference between risk and safety integrity, although they are two concepts that are closely related and almost always mentioned in association with each other. Risk is related to a hazard and measures the likelihood of that specific hazard. The safety integrity of a system is a measure of the likelihood of that the safety system correctly performs its task. The SIL indicates the average probability that the safety-related system will not perform its safety function on demand. The SIL can also be said to express the likelihood of the event that an error passes undetected through a safety system.

The IEC standard distinguishes between two types of systems that are used in different ways:

Low demand:
Systems in this class operate in a demand mode, they are called upon when needed. Failure rates in the low demand class are expressed in the probability that the system will fail to function correctly when called upon. Typical systems that belong to this class are shut down systems. These systems sit idle and perform nothing until they are needed.

Safety Integrity Level (SIL)	**Low demand mode of operation** (Average probability of failure to perform its design function on demand)
4	$\geq 10^{-5}$ to $< 10^{-4}$
3	$\geq 10^{-4}$ to $< 10^{-3}$
2	$\geq 10^{-3}$ to $< 10^{-2}$
1	$\geq 10^{-2}$ to $< 10^{-1}$

Table 3-1, This class of systems operates in a demand mode. The demanded function is performed less than once per year. The Safety Integrity Levels expresses the average probability of a system's failure to perform its design function on demand.

High demand:
Systems in the high demand class operate in a continuous mode, or uninterrupted operation mode.

Safety Integrity Level (SIL)	**High demand or continuous mode of operation** (Probability of a dangerous failure per hour)
4	$\geq 10^{-9}$ to $< 10^{-8}$
3	$\geq 10^{-8}$ to $< 10^{-7}$
2	$\geq 10^{-7}$ to $< 10^{-6}$
1	$\geq 10^{-6}$ to $< 10^{-5}$

Table 3-2, Safety Integrity Levels for systems with high demand or continuous mode of operation. Shows SIL related the probability of a dangerous failure per hour.

The requirements for SIL 1 and 2 are met by many of the distributed control systems (DCS) and programmable logic controller (PLC)-based control systems available today [6]. Depending on the SIL assigned to a system, different amounts of functions must be added to achieve the appropriate level of safety. According to "Using Foundation Fieldbus in Safety Applications" [1], FF is in its standard form capable of fulfilling SIL 2 requirements. Dedicated safety systems are generally required for applications at or above SIL 3. So, if FF is going to be used as a communication system in a SIL 3 application, some additions have to be made. These additions are described in chapter five, where I also will return to SIL-issues and probabilistic considerations of the safety layer.

3.4 Other standards and regulations

The IEC 61508 standard covers all safety systems based on E/E/PE equipment. Technologies used in various safety systems for industrial applications are mentioned in section 1.1.2 except pneumatic systems. The standard also provides requirements for sub-systems that are part of a larger safety system. As mentioned introductorily in this chapter, a safety system must comply with other international and national standards applicable for the specific application the system are to be used in. Table 3-3 shows a list of the standards most commonly used to supplement IEC 61508 compliance for safety related sub-systems.

Applicable Standards for Safety Related Sub-Systems	
Standard	**Specification**
IEC 61508, Parts 1 to 7 (inclusive)	Functional Safety for Safety Related Systems
ANSI / ISA S84.01	Application of Safety Instrument Systems for the Process Industries
IEC 68 Parts 1, 3,2,14, 26, 30	Environmental Testing
IEC 801Parts 3,4,.5,6	Electromagnetic Compatibility for Industrial Process Measurement and Control
IEC 1000 Parts 4-4 and 4-6	Electromagnetic Compatibility (EMC)
IEC 1131	Programmable Controllers
EN 50081	Electromagnetic Compatibility -Emission Standard
EN 55011	Electromagnetic Compatibility -Emission Power Lines
ANSI / IEEE C62.41	Immunity, Power Line Surge
ANSI / IEEE C37.90	Immunity, Electrical Fast Transients
EMC Directive	EMC European standard

Table 3-3, List of some of the most common standards used to supplement IEC 61508 [23]

[23] Paris Stavrianidis. “What Regulations and Standards Apply to Safety Instrumented Systems?” Control Engineering Online. http://www.controleng.com/archives/2000/ctl0301.00/0003we1.htm

Chapter 4 Safety Critical Communication

IEC 61508 generally divides a safety system into smaller *logical* units, or subsystems. On the input side there is the sensors, sampling data from the process and feeding the system with input. On the basis of this input the logic-solving unit determines actions according to the control strategy the actuators performs. In traditional safety systems are these subsystems directly connected together by cables. Each single actuator and sensor is coupled to the centralised computer with a pair of wires. This is illustrated in Figure 4-1. Most early safety systems used duplicated hardware modules to achieve a certain levels of reliability and safety. The use of redundancy prevents that a failure of one module would cause the whole system to fail. An arrangement of duplicated hardware allows the logic solver to make use of majority voting to remove the effects of single failures. In case of disagreements between the signals collected from the sensors, the centralised computer assumes that the majority view is correct. The computer in the safety system used loop supervision to monitor all the wires to the peripheral devices. Small electrical currents were sent through the devices and back to the logic solver to check if the connections and cables were intact.

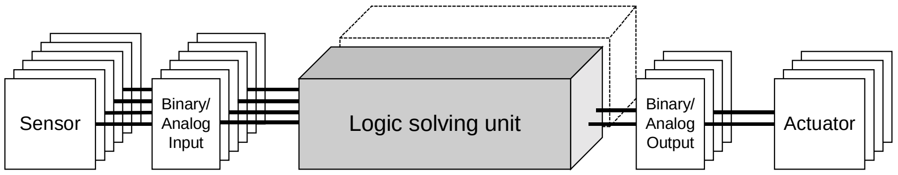


Figure 4-1, In a traditional safety system the sensors and actuators were directly connected to a logic solver by cables. Each peripheral device had at least one pair of wires coupled to the centralized computer. Redundancy have been used on all levels in a safety system, from software redundancy with n-version programming to redundancy of hardware (wires, sensors & actuators and logic solving units).

By replacing the direct cabling with a fieldbus communication system, the signalling between the devices and the logic solver is changed drastically, and implementing these software-based safety-critical systems require more in-depth methods and concepts than what traditionally has been used in software engineering. In systems with direct cabling, there is no actual communication between the sensor and the logic solver. The centralised computer collects sensor readings and sends actuator commands through a pair of wires. The signals and commands are sent as voltage levels or currents. Fieldbus supports real data communication. Each device has communication abilities and data is transmitted in digital messages. The fieldbus enables a two-way communication between the devices or network nodes. Every device in the network shares the same communication media, the bus (See Figure 4-2). It is obvious critical for a safety system that data can be reliably exchanged. Fieldbus technology is more advanced and complex than direct cabling, thus new challenges arise in terms of the communication between the devices in a safety system. Failure modes related communication and fieldbus that may affect the system safety are discussed more closely in section 4.2.

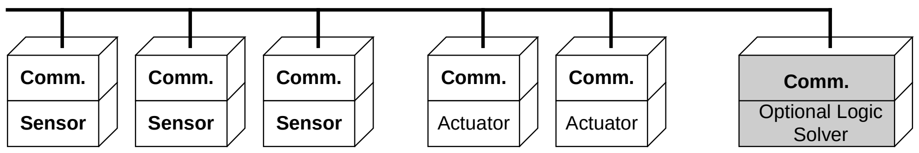


Figure 4-2, In fieldbus based systems is there a two-way communication between the devices and/or the logic solving unit. All communication is done on through the same cable and this introduces more possible sources of errors.

We are now touching the essence of this thesis; how can a standard fieldbus be modified to provide the same level of safety and reliability as a direct-cabled system? This can be done with modular redundancy and voting similar to what is done in traditional safety systems, but this issue is outside the scope of this thesis. I have concentrated on issues related to the communication between the “intelligent” fieldbus devices. The safety layer shall ensure that messages are transmitted correctly from one device to another at the right place in time.

Each single subsystem has an inherent probability of failure. A failure is an event that occurs when a system or component is unable to perform its intended function for a specified time under specified environmental conditions [22]. A failure can also be described as the mechanism that makes faults within a system or component apparent. A fault is a defect within a system. There are many types of faults and therefore it is convenient to categorise them into two distinct classes, namely *random faults* and *systematic faults*. Random faults are associated with transient hardware component failures. All physical components are subjected to failure, due to e.g. wear-out and degradation over time or sporadic environmental disturbance. This implies that individual components fail randomly once in a while. This in turn implies that all systems are subject to random faults. These failures are difficult to predict, but it is possible to build up statistics to estimate failure rates and in that way predict the overall performance of the system. Experience allows us to model the effects random hardware faults may have on a system. Systematic faults cover many forms, design faults and mistakes within the specification of the design of the system. All software faults are systematic. Systematic faults are not random in their nature and are difficult to analyse. It is almost impossible to predict their effects on system performance.

None of the subsystems in a safety system can be considered to be free of faults. Every fault in each individual subsystem contributes to increase the probability of failure for the whole safety system. Communication systems consist of both hardware and software, and are therefore subject to both random and systematic faults. It is impossible to design a system without faults and therefore also impossible to make an absolutely safe system. A system must be adequately safe for its given role, and IEC 61508 can be used as a reference to assist developers how to determine what “adequately safe” means.

4.1 Risk Considerations For Safety Critical Communication

In IEC 61508 a safety system is given a probability of failure as a whole by determining a SIL for the system. The SIL is an expression of the probability of an event occurring where the system is unable to perform its function correctly. This event may result in a dangerous situation. Each subsystem is critical for the system safety and is allocated a fraction of the total probability of failure. How much of a safety system's probability of failure that is allocated to each subsystems varies and must be determined for each single system.

Subsystems are also built up of smaller systems. If we decompose a complete system down in to smaller and smaller parts, we start to se that there are many single parts that can fail and can directly or indirectly lead to dangerous situations. Only a small fraction of the total probability of failure for a safety system is related to the communication system. The rate at which errors do occur in a communication system are strongly application dependent and must be considered individually. It is quite evident that data transmitted via a wireless communication link in a noisy environment are much more subject to errors, than data sent in an optical fibre under optimal conditions. Nevertheless, a communication system is a part of the total system and the safety function depends upon it to function correctly. It is extremely important that the inter-subsystem communication is reliable. A chain can only be as strong as its weakest point.

Foundation Fieldbus is a communication system that sends and receives data several times per second, this implies that a safety function based on FF has a high demand of operation (Table 3-2). The relationship between the total probability of failure for the safety system as a whole, and the communication system can be expressed in the following manner:

$P_{safetyFunction}$:	Probability of a hazardous error per hour in an uninterrupted operation mode *for the whole safety function*.
P_{ComSys}:	Probability of a hazardous error per hour in the communication system (In this thesis the communication system is FF).

$P_{ComSys} = k * P_{safetyFunction}$

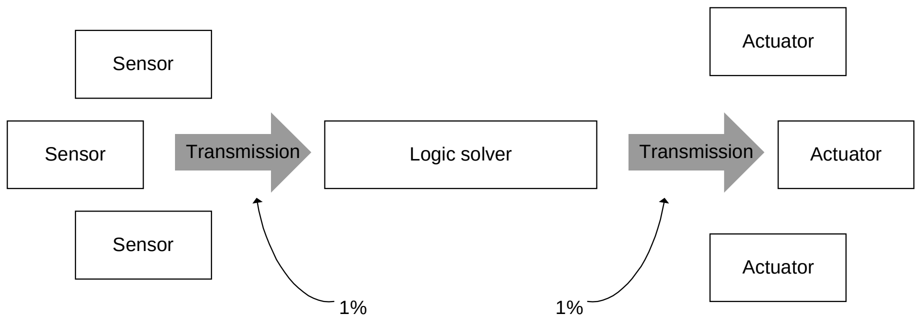


Figure 4-3, The whole system should be considered for safety. The entire path from input, through logic solving to output is relevant for the safety, but the communication subsystem is allocated 1% of the total probability of failure.

For communication systems the k is typically set to 1% [1, 15] (See Figure 4-3). This means that 1% of the total probability of failure of the safety system are allocated to the communication system. This means that 1% of the failures that can happen in the safety system is caused by an error during transmission of data.

Table 3-2 shows that for a high demand system with SIL 3 requirements we can see that the maximum probability of a hazardous error per hour (in interrupted operation mode) is less than 10^{-7}. The following applies to the safety layer:

$$P_{safetyFunction} < 10^{-7}/h$$
$$P_{ComSys} < (0.01 * 10^{-7}) = 10^{-9}/h$$

4.2 Communication failure modes outlined in IEC 61508

IEC 61508 address all known failure modes for communication systems, such as software failures in the communication protocols, hardware failures in transmitters, receivers, gateways and routers and sporadic disturbance of the transmission path.

The IEC 61508 introduces the following failure modes:
1. Data corruption
2. Corruption of sender and/or receiver addresses
   - ❑ Wrong sender and/or receiver addresses
   - ❑ Multiple receiver addresses (identical addresses on different devices)
   - ❑ No address match
3. Transmission of data packages at the wrong point in time
   - ❑ No transmission
   - ❑ Delayed transmission
4. Wrong sequence of packages

A safety-critical system requires that data can be validated in both a value- and a time-domain. To determine the validity of safety relevant data, we must have access to enough information related to the data to be able to correctly answer four questions. Under each of these questions I have listed alternative ways to look at the same questions to clearly state what lays beneath the motivation for theses inquiries:

- Are the data I just received from the right sender?
  Does this arrived data have the right sender-address? Is it right that this device is sending data to me? Am I the right recipient?
- Does the data I just received have the right value?
  Have the data been corrupted along the way to me? Can I trust the values I just received from my communicating partner?
- Is it correct that I received data at this point in time?
  Is it too late or too early to receive data now? I expect to receive data now, where is it?

- Does the data have the right sequence number?
  Have I received this data before? Have I missed some data?

4.2.1 The IEC 61508 communication failure modes related to the standard FF communication protocol

In this section I summarise the findings presented in "Using Foundation Fieldbus in Safety Applications" [1]. This report started the primary work on identifying weaknesses in the standard FF stack that can be associated with safety. These "flaws" was related to the failure modes in IEC 61508. The material presented in [1] served as basic considerations for my approach to the safety layer problem.

4.2.1.1 Data corruption

The first weakness that renders it impossible for plain FF to be as the communication system in highly safety-critical applications is its error detection mechanism. The data link layer incorporates a Frame Check Sequence (FCS), also known in the literature as a cyclic redundancy check or cyclic redundancy code (CRC). CRCs and the concept behind them are discussed more closely in section 5.3.1 "Data Corruption." The FCS is used to detect small changes in the data frames. It is claimed in [1] that the FCS implemented in the data link layer is the most uncertain issue related to using FF for safety relevant communication. Standard FF uses a 16 – bit CRC generator polynomial. According to IEC 61158-2 [18] the Hamming distance for telegrams shorter than 15 bytes using the FCS is 5, and for telegrams longer than 15 bytes and shorter than 345 bytes the Hamming distance is 4. In [1] there is put a question to whether FF's error detection mechanism is powerful enough to even meet SIL 2 requirements, therefore is it very likely that a new FCS/CRC with a longer Hamming distance must be provided.

4.2.1.2 Corruption of sender and/or receiver addresses

The second problem with FF is the VCRs (Virtual Communication Relationship). The three VCRs available in Foundation Fieldbus each offer a different Quality of Service (QoS). Depending on which type of communication (VCR) channel used, a different amount of addressing information is added to the header of the frames. Two of the communication channels (Publisher/Subscriber and Report Distribution) include only the address of the sender in the header of the transmitted data packages. In safety critical communication it is necessary to include both the sender- and destination address. With both addresses included in message headers, it is possible to check whether a package is delivered according to its intentions. Wrong deliverance of packages may cause unpredictable behaviour and problems in the system. Because some messages do not include the destination address, the receiver does not have the ability to check if received packages are really intended for them. The Client/Server VCR includes both the sender and the destination address, despite this feature it is not suitable to convey safety critical data. The Client/Server channel is used for user initiated communication, such as up and downloading of data, but it does not offer time-deterministic capabilities. Even though this VCR includes both sender and receiver addresses, it cannot be used for safety critical communication because of its lack of time determinism. Thus none of the three standard channels provided by FF incorporate all the QoS-attributes required in safety critical communication.

4.2.1.3 Inaccurate timing of transmission of data packages

The third "weakness" of FF related to safety was introduced in the previous section, namely time-determinism. To be able to answer whether an event, like the arrival of a data frame, has occurred at the correct point in time, it is essential for the system to be time deterministic. The Publisher/Subscriber VCR is the only communication channel offered in FF that is scheduled and synchronised with time, and thus providing time-determinism. The Publisher/Subscriber VCR is used during normal, operational, cyclic communication on the bus to periodically distribute control loop data. The scheduled traffic is controlled by the LAS, which ensure that fieldbus devices cyclically transmit data at the right time. But as we remember from the previous section, the only address in the frames sent while using the Publisher/Subscriber VCR is the transmitter's address. The subscribers are configured to listen for messages from their corresponding publishers, instead of listening for messages addressed to themselves. A subscriber has no way to verify the origin of received messages under these circumstances.

4.2.1.4 Wrong sequence of packages

If a frame sent from a device to another is lost because of some reason or another, the consequences may vary from application to application. During configuration of a FF system, the operator can configure the number of consecutive duplicate values a function block should accept before it labels the input data as old. What happens when the data is stamped old, is application dependent. In some non-safety-critical applications the process can tolerate a relatively high number of losses of frames with fresh data, depending on the inertia of the process. In a safety-critical communication system no loss of data and frames can be tolerated. During normal operation of a FF system the Publisher/Subscriber VCR is used. This VCR does not have any form of sequence numbering of its frames, thus the receiver does not have any concrete means to detect whether it have missed any transmitted frames, or whether a faulty publisher is locked in one state and distributes legal but old data. The Client/Server VCR is the only channel that supports package sequencing and retransmission of lost frames, but again, this channel has no support for time determinism..

4.3 Principle Solution For Safety Critical Communication

The essence of the four previous sections is that none of the preconfigured communication channels incorporates all of the attributes necessary in safety critical communication. All of the three VCRs offered from FF have identical error detection mechanism incorporated by the same FCS. This error detection mechanism is probably not powerful enough to be used in an application that is assigned SIL 2 or higher [1]. In this thesis I have had as a goal to design and implement a safe communication function with a target of SIL 3. Hence, due to insufficient error detection none of the VCRs is suited for use in the safety function. The Client/Server is the only VCR providing sufficient amounts of addressing and sequence numbering, but this VCR lacks time-determinism. The Publisher/Subscriber VCR distinguishes itself from the two others at this point as being the only time-deterministic channel supporting scheduled transmissions. But this VCR cannot be used in a safety function either without some modifications, because this VCR does not incorporate sufficient addressing and do not have any form of sequence numbering. We have now seen that when applying the failure modes to the standard FF protocol, the result is that none of the standard VCRs are suitable for safety critical communication.

4.3.1 The Safety Layer

The safety layer is a concept introduced in [1]. I chose to use the same name for my safety function. One of the main design criteria posed for the safety layer was that it should be done in a way that left the FF protocol layers untouched. Any solutions involving making changes in the FF communication stack would make the system an incompatible FF-hybrid. This was not an acceptable solution, as safety applications and standard applications shall share the standard communication system at the same time.

This constraint limits the possible ways to approach a safety layer solution. The safe communication protocol I describe in the next chapters is based on a model that uses standard FF as the transmission system, with an additional safety function on top. This approach provides the fieldbus applications safety on an application to application level. By adopting this approach and aiming for a peer-to-peer safety among applications, I ignore the built-in functionality of FF that relates to reliable transfer of data. I use FF as a means (a channel) for conveying messages between the safety functions. The channel is considered to be unreliable by the safety layer. The FF stack provides just availability of communication abilities, and does not have anything to do with the safety. The protocol stack just conveys messages for the applications independent of the content of the messages. The standard FF stack has no means for understanding the content of the transported data, and therefore knowing whether the messages are logical correct for the applications. The added safety function provides for validation of safety relevant data sent through the FF-channel. This design gives a fundamental and generic solution that does not pose any demands for making changes of the fieldbus. This type of approach makes the concept of the safety layer independent of the underlying technology. It is therefore possible to use the safety layer design in conjunction with other fieldbuses than just FF by allowing for small implementation specific differences. The safety layer shall detect faults that are by some reason or another presented by the standard FF communication network. By introducing a safety layer the probability of an error/fault passing undetected through the system are kept under a certain limit. The safety integrity level assigned to the safety system determines this error-limit.

Chapter 5 The "Safety Layer" for Fieldbus Foundation

In chapter 1.2 it is stated that a major purpose of this thesis is to propose a safety layer making Foundation Fieldbus suitable as a communication system in SIL 3 applications. I have up to this point in the paper introduced FF and the concept of safety and safety systems. I have also discussed how the communication failure modes described in IEC 61508 relate to FF and outlined counter measures for the failure modes. In this chapter I will describe my own proposal for a way to implement the safety layer in a FF environment. Sections 5.1 and 5.2 describe how the safety layer concept must be specified to fit into the FF architecture. Section 5.3 deals with how the layer addresses the individual failure modes. Then the design of the safety layer is outlined in 5.4 and I explain more in detail how the safety layer works. At the end of the chapter I comment on a demonstration implementation of the safety layer. The safety layer in the demo system was implemented as described in the consecutive sections in this chapter.

The safety layer outlined in this report is strictly concerned with aspects related to communication between FF devices. I have proposed a safety protocol that makes the communication between two FF devices more reliable and able to meet the stringent SIL 3 requirements. The safety protocol gives the user-level applications in a field device an increased probability to determine whether transmitted messages are valid or not. The applications can with an increased amount of certainty claim that the messages are free from errors. This will contribute to make the system safer by diminishing the possibility of errors occurring and also the probability of not detecting present errors.

There is more than one possible approach to make the communication "safer" between two FF devices. One possible solution is to change the protocol stack itself. This solution involves making changes and additions in the different layers to improve the communication reliability. There are both technical and practical reasons why I chose not to adopt this solution. First of all I wanted to make a generic layer or mechanism that was independent of the underlying technology. I wanted to make a safety layer that could be used together with other technologies than FF. A completely generic approach like this is of course not possible, because different technologies have distinct properties and interfaces. By changing the protocol layers we produce a hybrid of the original FF stack. This is not a practical solution concerning device interoperability, and may lead to that "safe devices" and standard devices may not be used on the same bus. Another factor keeping me from choosing an approach involving tampering with the protocols is simply because I did not have access to the source code of the stack. I only had a binary file of the FF stack. This also applied to the FB-Shell, the interface between the stack and the user applications. First I thought that I could base the safety layer on communication primitives, like send() and read(), provided by the FB shell. I soon realised this was not possible since the shell only provides callback functions (Ref. paragraph 2.3.3 and 2.4). The FF device starter kit from National Instruments that was employed for implementing the safety layer did not provide the source code of the FB-Shell, therefore the only possible way remaining of accessing the bus was through the user application.

The safety layer described in this report is conceptually based on a standard transmission system (in this case FF). In order to achieve “safe communication” (strictly speaking, more reliable communication) some additional safety transmission functions are added on top of this standard transmission system. The communication stack is left untouched. This approach is similar to how the safe profile is designed in ProfiSafe (Ref. section 1.1.6). With the term standard transmission systems I mean, in this context, the whole standard FF communication stack including all the hardware and the belonging software protocols.
At the top of the FF stack we have the user layer. The user layer defines function blocks (See Figure 2-1 and 2-5). The FF standard opens for development of function blocks with new functionality. Device vendors can therefore develop their own function blocks to differentiate their devices and instruments from the products of their competitors. The safety layer takes advantage of this feature. In a simple way the safety layer can be described as a built-in part of a function block. The safety layer design is based on a standard, regular function block that gets an addition of safety-related communication parameters. The function block reside in the application on top of the stack, hence the communication stack is left untouched. The majority of the safety-related add-ons are placed inside the standard function blocks. This approach incorporates an application to application safety-related communication. The safety layer or function disregards the reasons for the errors and faults occurring in the transmission system. From the safety function’s point of view, FF provides the availability of communication and not the safety. It is the safety layer that is concerned with issues relevant for the safety-related communication. The layer considers FF to be unreliable, and therefore it adds its own redundant information it uses to validate transmitted data. The concept of the safety layer presented in this thesis can be applied to other transmission systems. Some adjustments have to be done in order to adapt the layer to the underlying technology. The principles can be transformed to suit other underlying technologies. The underlying technology does not have to be FF. But of course there are certain implementation specific details that must be changed when applying the safety layer on other communication technologies. The idea and principles can be used. With this approach we will not get a safety layer in the layered sense. We will get application to application safety. This approach means that the safety layer is not really a protocol layer, it is more like a part of the application, than a separate layer like the other protocols in the OSI reference model. The safety protocol runs as a user application in the CPU of the device. The safety layer is therefore also referred to as the safety function in this document.

5.1 The Safe Function Block Concept

The general idea behind the “safe function block based safety layer” is based on the idea of a library of “safe function blocks” that incorporates some means to communicate safely, e.g. more reliably. The library shall contain similar blocks to the collection of standardised function blocks. The difference between the regular function blocks and safe blocks, is that the blocks from the “safe library” are customised and gets an overhead that makes the communication between these blocks more reliable. From a process-control point of view, the safe function blocks have the same functionality (i.e. the same control algorithm) as the standard function blocks. The difference between standard and safe blocks lies in the extra efforts that have been made to make the communication between the safe function blocks more reliable. Building a library of safe function blocks will practically speaking involve taking existing, regular blocks and modify them with the necessary additions to make them communicate safer. The safe blocks will get redundant information that is used to improve the reliability by strengthening their abilities to detect communication failures.

To illustrate the concept of the safe function blocks and the main difference between standard function blocks and the safe blocks, let us first consider a part of an ordinary process control scenario with two regular FF function blocks FB1 and FB2. FB1 and FB2 are located in two separate physical devices and are configured to communicate via the bus in the following manner; FB1’s OUT-parameter is linked to FB2’s IN-parameter (See Figure 5-1). FB1 is constantly sampling data from the process environment. This data is cyclically published by FB1 when the LAS issues a “Compel Data” message for FB1. FB2 consumes the data produced by FB1 and is therefore configured to listen for messages on the bus published by FB1. The transmission of data between FB1 and FB2 is scheduled, thus the linkage between the two function blocks is based on a Publisher/Subscriber VCR. We recall from section 2.3.1.3 that data transmissions using the Publisher/Subscriber VCR are unidirectional broadcasts, thus data is flowing in only one direction, from FB1 to FB2. The Publisher is told by the LAS to publish the data in its output parameters on the bus. The Publisher do not care which function blocks that might be interested in the data it just publishes it. The Subscribers are the destination of the published data, and they are configured to listen on the link for messages and pick up the messages they want. In safety-critical communication it is essential for a function block receiving a frame to be able to notify the sender whether the frame arrived correctly or not. In a scenario using regular function blocks, the Publisher do not have any chance to know whether all of the destinations have correctly received the data they subscribe to. The linkage between the Publisher and the Subscriber only supports a monologue communication pattern.

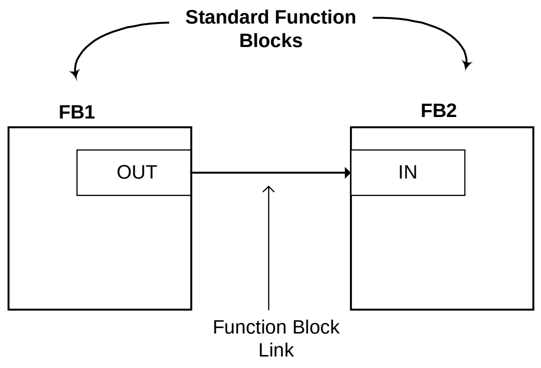


Figure 5-1, Two standard function blocks liked together. Data cyclically flows only in one direction, from FB1 to FB2.

Let us now consider an installation with the similar configuration as shown in Figure 5-1, but this time in a safety system. In this safety-critical setting the importance of correct transmission of data is evident. In a safety critical setting a system engineer would configure the system using blocks from the safety library. By adding a safety function to a regular function block the probability of undetected errors between two communicating applications will diminish, supposing both blocks have a synchronised safety function. In that way a safety function will contribute to increase the safety and reduce the risk related to the installation. When enhancing a standard block with a safety layer, a set of data and some related software routines are added to its definition. Figure 5-2 shows the same application as in Figure 5-1, but this time the application is built up of blocks from the library of safe function blocks. There is no difference between operational functionality of the safe configuration and the non-safety critical configuration. The difference between the two configurations lies within the reliability of the transmission of data between the function blocks. The function blocks in Figure 5-2 communicate safely through their safety layers.

A safety layer performs its task when the function block executes. This implies that the safety layer executes cyclically and is totally dependent on the LAS schedule. This in turn means that a Publisher/Subscriber VCR lies beneath the connection between the two safe blocks. So far there is no fundamental difference in the communication model between the "safe-" and the regular configuration. The safety layer takes advantage of the scheduled properties of the Publisher/Subscriber VCR and uses it as a basis for the safety critical communication. The safety layer enables a function block to monitor the communication it performs with another block. The layer attaches redundant information to the process data broadcasted by a Publisher. The safety layer on the receiving end of a linkage use this information to validate both the time of arrival and value of each frame it receives. The safety layer also implements a feature making the linkage between two function blocks a bi-directional channel. This opens for a two-way communication between safe function blocks and a Subscriber to acknowledge messages from the Publisher. I have called the messages sent from a safety function-enhanced Publisher for "*safety frames*." A safety frame contains all the data needed for the Subscriber to validate its communication with the Publisher. To confirm correct receptions of safety frames the Subscriber sends short messages back to the Publisher. This short message is not a full safety frame and is just referred to as an acknowledgement or an ack. The function blocks in Figure 5-2 communicate safely through their safety layers.

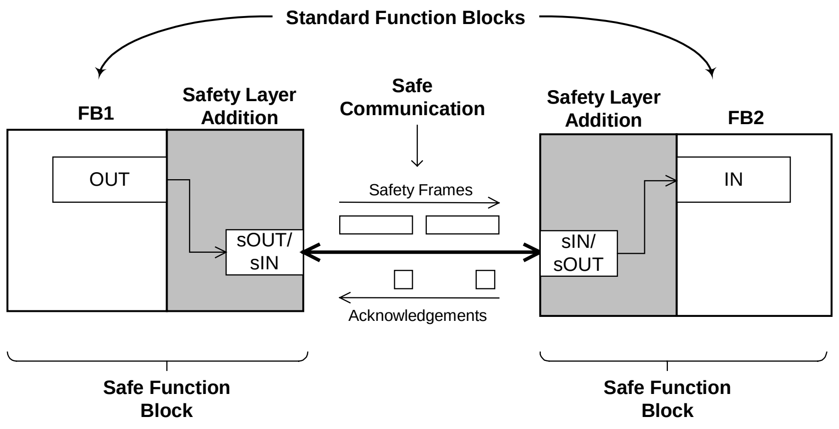


Figure 5-2, Schematic illustration of the concept of the “safety layer.” The grey shaded area symbolises the overhead added to ensure a safer communication between the function blocks. The safety layer has both safe input- and output-parameters that makes two-way communication possible between the function blocks.

The concept of safe function blocks results in that the safety layer only supports the cyclic transmissions. User initiated communication, like downloading of configuration data, will not be protected by the safety function. For system engineers using this function block based solution, they can configure the system using the same tools and techniques as they are used to with non-safety critical fieldbus systems. The use of already known tools and techniques is advantageous and does not introduce the need for any comprehensive training of personnel. The safety frame, the acknowledgement and their contents are discussed more thoroughly in section 5.2 and 5.3.

5.2 Safe Function Block Specification

In the previous section the safety layer was defined to be an integrated part of a function block. The safety layer manages a set of redundant variables used to hold information about the communication. These variables are discussed more in detail in section 5.4. When a safe function block is scheduled to run and publish data, its safety layer includes updated versions of the safety-related variables in the message before the frame is published. The redundant data consists of a new addressing scheme for the safe function blocks, a frame counter that handles sequence numbering of transmitted frames, timestamping of frames and an enhanced error detection mechanism. The receiver uses the values of these variables to validate safety frames. Each published safety frame incorporates the same amount of redundant data, only the content is updated between each transmission. A safety frame has therefore a fixed size. The same applies to the acknowledgements, except they are smaller in size than a complete safety frame.

For each scheduled execution of a function block there is a limited amount of data that can be sent and received at a time. The data type of the I/O parameter in a function block determines the amount of data the block can send/receive in one bulk. It is a FF requirement that input and output parameters of function blocks have to be of one of three data types, either a VsFloat, a VsDiscrete, or a VsBitstring. These I/O parameters contain both value and status attributes.

FF I/O-Parameter Data Type	Element Name	Data Type	Size
Value & Status – Floating Point (VsFloat)	Status	Unsigned Char	1 byte
	Value	Float	4 bytes
Value & Status – Discrete (VsDiscrete)	Status	Unsigned Char	1 byte
	Value	Unsigned Char	1 byte
Value & Status Bitstring (VsBitstring)	Status	Unsigned Char	1 byte
	Value	Unsigned Short	2 bytes

Table 5-1, The FF specification defines that a function block's I/O parameters can be one of these data structures.

Table 5-1 shows that the maximum amount of data a function block can send or receive through one I/O parameter per execution are five bytes (4 + 1 in VsFloat). All of this space is already occupied in a regular function block by the process value. The amount of redundant data the safety layer appends exceeds the capacity of a regular function block's output. How can the safety layer append any redundant information if there is no vacant space in the messages? What can we do to send more than five bytes at a time?

5.2.1 Safety Frame

To overcome the problem related to sending and receiving of a safety frame bigger than the I/O parameter with the largest capacity, I have chosen to divide a safety frame into smaller units called fields (See Figure 5-3). A field has a fixed size of five bytes so it can fit into a function block's I/O parameter of type VsFloat. When a safety frame is going to be sent, the safety function on the publishing side of the communication splits the frame into fields. Each field fit into the block's output parameter and can be sent sequentially one by one. The safety frame is therefore not transmitted as a single unit, but as separate fields. The receiving safety layer builds the safety frame piece by piece as the fields arrive and the frame is gradually assembled. After the arrival of the last field, the safety layer validates the whole frame. If the safety frame is valid, an acknowledgement is issued to the sender of the safety frame. If not, no acknowledgement is sent. If the publishing safety layer do not receive an acknowledgement within a limited period of time, a timer times out, and the layer flags the error.

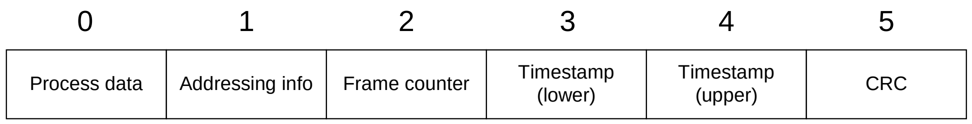


Figure 5-3, One safety frame contain all the safety relevant data necessary to validate a transmission. The frame is too big to be sent in one bulk and therefore must be split into fields that are individually transmitted. A safety frame consist of 6 fields, numbered from 0 to 5.

The whole concept with splitting a frame into fields and serially transmitting frames through a single I/O parameter seriously damages the systems throughput. This can be compensated for, up to a certain point, by increasing the frequency of function block executions. Let's say a control loop (Ref. section 2.3.2.1 for definition of control loop) of regular function blocks is configured to have a loop time of one second. This means that each function block executes once in intervals of one second. If a loop consisting of safe function blocks were configured to run at the same rate, process data would only be transmitted each sixth second. The rest of the "bandwidth" would be occupied with safety overhead. But if the safe function blocks were configured to run with an interval of 166 ms (on sixth of a second), approximately the same performance would be achieved as the loop with regular blocks.

5.2.2 Fields

The content of a safety frame is not divided randomly into fields. Each safety-related variable has a corresponding field in a frame, this means that a specific field holds the same type of content during each transmission. The first field (field 0) in a frame always contains process data. This means that the content of field 0 always corresponds to the data regular function blocks would exchange under normal operation. Field 1 contains addressing information used to identify the origin and destination of the safety frames. Field 2 contains the actual frame's sequence number. This is primarily used to monitor the continuity of frame transmission. The frame counter is used to detect loss of frames and avoid function blocks to interpret a frame more than once. This last issue is rather unlikely to happen because the safety layer does not support retransmissions of frames. The fourth and fifth fields contain a timestamp. The timestamp is too large to fit into one field, therefore must this parameter be divided into two separate fields. The sixth and last field carries a checksum that is calculated to protect the five preceding fields (See Figure 5-3).

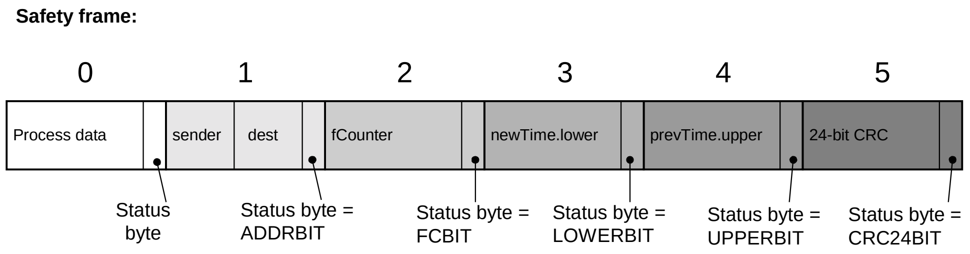


Figure 5-4, The structure of a safety frame. The status byte is used to hold a bit that indicates which field that are sent or received.

Each field has an identifier. This identifier is a byte containing a bit-pattern unique to the field it represents, and acts as a field header that indicates what type of data the field contains. This byte is always sent in the status byte of the I/O buffer (e.g. the status byte in the VsFloat I/O parameter, see Table 5-1). As fields are arriving at the receiving end of a safe communication link, the recipient checks if the arriving fields has the correct identifier according to what is expected. When all six fields have arrived and the safety frame is complete, the safety function computes a checksum for the arrived data. If the calculated checksum in the destination is equal to the checksum it received from the sending function block, the probability of error during transmission is so low that the block presumes the frame is not corrupted. When a frame has arrived successfully, the receiver issues an acknowledgement message to the sender.

5.2.3 Acknowledgements of Safety Frames

To enable the safety layer to send and receive acknowledgements, I have defined a new I/O-parameter. This I/O parameter specifies two versions of the safety layer, one for a Publisher and one for a Subscriber. The Publisher's safety layer got an additional input parameter while the Subscriber's layer got an output parameter. These parameters were included exclusively for one reason, namely to handle the sending of acknowledgements. This means that the connection between two safe function blocks is composed of two Publisher/Subscriber VCRs. One VCR is used to convey safety frames from one block to another and the other VCR is used to transport acknowledgements from the recipient of safety frames to the original Publisher. Both function blocks can therefore instantly be considered to be a Publisher and a Subscriber. One safe block publishes safety frames and subscribes to their related acknowledgements. The other function block subscribes to safety frames and publishes their acknowledgements if the whole frame is received correctly (See Figure 5-5).

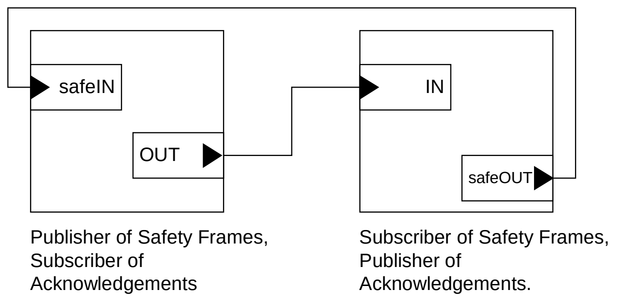

safeIN
OUT
IN
safeOUT
Publisher of Safety Frames, Subscriber of Acknowledgements
Subscriber of Safety Frames, Publisher of Acknowledgements.

Figure 5-5, The safe bi-directional channel between to safe function blocks is constructed from two linkages based on the Publisher/Subscriber VCR.

The acknowledgement is defined to be one field. This size allows an ack to be sent during a single transmission. By limiting the size of the ack to one field the throughput is practically not affected. This leaves us with five bytes for disposal in an acknowledgement message. The receiver of an acknowledgement, the ack-Subscriber, needs to recognise which safety frame the ack is intended to confirm. The sequence number serves this purpose. An ack include the sequence number of the safety frame to indicate which frame it acknowledges. The safety layer is designed as a stop-and-wait protocol, therefore a new safety frame can not be sent before the last of the previous frame has been correctly acknowledged. It is not possible for a pair of communicating safe blocks to have more than one outstanding unacknowledged frame at a time.

As already mentioned, the frame counter or sequence number is an important part of the ack. An ack is invalid if it gets corrupted during transmission, therefore an error detection mechanism is also required for the ack. To be able to fit both the frame counter, an ack-bit that identifies that the message/field is an ack, and redundant data required by the error detection mechanism, the available bytes in the ack must be managed carefully. I have chosen to just send the two least significant bytes (LSB) of the frameCounter in the acknowledgement. As the frame counter is incremented at intervals of six loop times and the safety layer uses a stop-and-wait communication pattern, the two LSB of the sequence number contains plenty of information to acknowledge a frame and keep track of the frames in a stop-and-wait protocol. To protect the acknowledgement I implemented a 16-bit CRC. A 16-bit CRC provides good error detection when it is used to protect just 24 bits. CRC-performance is addressed in section 5.3.1.1. Figure 5-6 shows the structure of an acknowledgement.

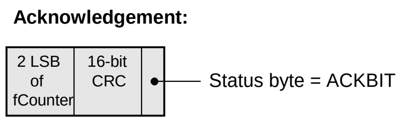


Figure 5-6, The acknowledgement frame is similar in size as a field. It is five bytes and contains four bytes of data. The status byte contains an acknowledgement identifier. The four bytes of data is used to hold two bytes of CRC and the two LSB of the frameCounter to identify the safety frame the function block is acknowledging.

5.3 How the IEC 61508 failure modes are handled in detail in the Safety Layer

There are several error cases the safety layer must handle and they are listed in the table below together with counter measures for the respective failure modes.

Counter measure: Failure:	Frame counter - Sequence number	Time expectation	Sender/Receiver addresses	Data protection
Corruption of data				X
Erroneous addressing				
Wrong address of sender/receiver			X	
Multiple receiver addressing			X	
No address match			X	
Wrong point in time				
No transmission	X	X		
Delayed transmission		X		
Wrong sequence				
Repetition	X			

Table 5-2, Measures for mastering errors and failure modes [1, 15].

In the rest of this section (5.3) I address the issues summarised in the table above (Table 5-2) and discuss in detail how the failure modes outlined in IEC 61508 (Ref. section 4.2) are handled in the safety layer.

5.3.1 Data Corruption

The safety layer offers an enhanced protection of transmitted data so that a function block can, with a sufficient certainty, determine if exchanged data are free of errors. The error detection mechanism incorporated in the safety layer shall have a targeted performance satisfying the requirements of SIL 3. These requirements are listed in Table 3-2. The error detection mechanism is implemented by means of CRC (Cyclic Redundancy Check) routines. CRC has been chosen to detect errors in the safety layer because it is one of the most effective and popular mechanisms for detecting errors used today [24]. The CRC is widely used because it offers extremely good error detection performance in proportion to the overhead it involves. In “Implementing CRCs” [24] it is given an example on CRC performance. J. W. Crenshaw writes that if we have given the proper choice of algorithm, a 16-bit CRC can detect:

- 100% of all single-bit-errors
- 100% of all two-bit errors
- 100% of all odd number of errors
- 100% of all burst error less than 17 bits wide
- 99,9969% of all bursts 17 bits wide
- 99,9985% of all bursts wider than 17 bits

The cyclic redundancy check is used as a way to detect small changes in blocks of data. The basic idea behind CRCs is that a sender computes a number that is mathematically related to message going to be transmitted. The number is appended to the message prior to transmission. The receiver calculates the number using the same formula based on the data it just received. The idea is that any errors in the data will affect the number computed, so that the errors can be detected.

In practice will a CRC-algorithm treat a message as a large binary number and divide this number by another binary number, the generator polynomial. The remainder of this division is the checksum. The checksum is sent along with the message. When the message reaches its final destination, the receiver performs the same division. If the two remainders are identical, it is very likely that the transmission was successful and the message has not been corrupted by any bit errors. Nevertheless, there is still a small possibility that some errors will not be detected. This happens when the pattern of a bit error affects the original message in a way that results in a new value which, when divided, produces exactly the same remainder as a correct message.

---

[24] Jack W. Crenshaw, “Implementing CRCs.” Embedded Systems Programming, January 1992.

5.3.1.1 CRC Performance

The performance of a CRC routine is measured by the probability of undetected-errors, P_{ue}. P_{ue} denotes the probability of the event that channel noise transforms a sent codeword into another codeword so that parity checking at the receiving end fails to detect the errors in the transmitted codeword. A central parameter affecting the P_{ue} is the *minimum Hamming distance*. The Hamming distance (HD) between two equally long code words is the number of bit positions two code words differ.

Given two code words,
N1: 10001001
N2: 10110001

To determine the Hamming distance between two code words, we can XOR the code words and count the 1-bits in the result. Hence we can determine the HD between N1 and N2:

	10001001 : N1
XOR	10110001 : N2
=	00111000

In this example we have three 1-bits, thus the HD between N1 and N2 is 3.

When two code words have a Hamming distance of *d*, it takes d single flipping of bits to convert one code word into the other. If a data frame consist of m data bits, called the message or the payload, and r redundant bits called the checksum, the whole code word has a length of $n = m + r$ bits. In most situations of transmitting m data bits in frames of *n* bits, all the 2^m possible message alternatives are legal values in the data field of the frame. Due to the way the redundant error detection bits (the checksum) for the message is calculated, not all of the 2^n alternatives of the *n*-bit long frame represent valid messages. Given the algorithm that calculates the checksum, it is possible to calculate the complete list of all legal *n*-bits code words. Legal code words mean frames that are not corrupted. From this list can we find the two code words with the smallest HD. To do this we have to compare all the code words in the list and locate the pair of code words that are the most alike (with the lowest number of differing bits between them). This Hamming distance is called the *minimum Hamming distance* (HD_{min}) of the CRC-code. The HD_{min} expresses among other things how many single bit errors the CRC-code can detect. This is a good indication of the polynomial's error detection properties. To guarantee the *detection* of *d* bit errors in a code word, we need a HD_{min} of $d + 1$. It is guaranteed that a CRC-polynomial with $HD_{min} = 4$ can detect all 3-bit errors, because it is not possible that three single bit errors in a code word, from the list of legal code words, can transform it into an other legal code word. CRC-codes can also be used to correct errors in data frames. To *correct d* bit errors in a frame, we need a $HD_{min} = 2d + 1$, because in this case the legal code words differs so much that with d bit flips (errors), the original code word is closer than any other code word, thus it can be determined. The greater the Hamming distance between two code words, the more difficult it is to change one into the other, thus we look for codes with the largest possible minimum distance.

When first confronted with the problem of choosing the CRC generator polynomials for the safety layer, I assumed that the best thing to do was to go for polynomials that are well known and widely used. After some more studying of the topic I quickly understood that it was not as easy as just picking a polynomial, implement it and then expect that it would do the job in the safety layer. Using generator polynomials just because they have been proven in use is not a good enough argument by itself to justify the use of the polynomial in this particular application. In the paper "Optimum Cyclic Redundancy-Check Codes with 16-Bit Redundancy "[25] we can read "CRC-16 codes with maximal minimum Hamming distance at short block lengths do not have maximal minimum Hamming at large block lengths, and vice versa." The same applies not only to CRC-16 codes, but also to CRC codes of other lengths. CRC generator polynomials are constructed to optimise the HD in a certain range of block lengths. If we use a CRC-code on a block with length outside the range the generator polynomial was designed for, the code's error detection properties may be completely unpredictable and show poor performance compared to the performance expected.

5.3.1.2 Probabilistic Considerations

To be able to find an appropriate generator polynomial, it is first of all necessary to be able to specify the performance we demand from the safety layer. This performance demand is indirectly reflected in the SIL assigned to the entire safety system. The safety layer is a high demand system operating in a continuous mode (Ref. section 4.1), and in this thesis I am targeting SIL 3 performance. From Table 3-2 we have for the whole safety system $P_{ue} < 10^{-7}$ per hour. The safety layer is a subsystem responsible for the communication between the other subsystems in the safety system. In section 4.1 we read that a communication subsystem is usually allocated 1% of the probability of failure for the whole safety system. Thus the probability of a hazardous error per hour in the safety layer during a continuous operating mode is:

$$P_{safetyLayer} < 10^{-9}$$

Further have I assumed that the number of messages sent per hour from one safe function block to another is 360.000 (assumes 100 messages pr second, thus a loop time of 10 ms.)

We have the following:
mr (message rate) = 360.000 ms/h (messages sent per hour)
$P_{safetyLayer} < 10^{-9}$/h

[25] G. Castagnoli, J. Ganz, P. Graber. "Optimum Cyclic Redundancy-Check Codes with 16-Bit Redundancy." IEEE Transactions on Communication, vol. 38, No. 1, January 1990

The residual error rate R for decoding of *one* telegram is:

$$R = \frac{P_{safetyLayer}}{mr} = \frac{10^{-9}}{3{,}6 \cdot 10^{5}} = 3{,}6 \cdot 10^{-9-5} = 3{,}6 \cdot 10^{-14} \approx 10^{-14}$$

We must therefore choose a CRC generator polynomial that can guarantee the safety layer that the probability of not detecting any errors when it decodes a messages is less than 10^{-14} for representative bit error rates on the FF bus. This is discussed in more detail in section 5.3.1.3.

The safety layer has two types of codewords with different lengths. First, there are the safety frames, which are six fields long including the CRC field. Each field is five bytes. A full safety frame is 30 bytes of which 25 bytes are the message string. The acknowledgement message consists of 3 bytes of data and 2 bytes of CRC. These codewords are relatively short compared to the codewords in i.a. Ethernet, were each frame can contain up to 1500 bytes.

The safety layer implements two CRC routines, a 24-bit generator polynomial used for the safety frames and a 16-bit polynomial protecting the acknowledgements. The CRC algorithms and implementation specific details are discussed in paragraph 5.4.4.

5.3.1.3 16-bit CRC

The safety layer uses the 16-bit CRC polynomial to detect any bit-errors occurring during the transmission of acknowledgements. An ack is quite short, only five bytes or 40 bits (4 bytes float and one byte status, Vs_Float). It is therefore necessary to choose a polynomial with a high HD_{min} for short block lengths.

The 16 bit generator polynomial used in the safety layer is based on a 12-bit binary BCH code (63, 51, 5), with generator polynomial $g_{12}(x)$ = 1001110010101 ($x^{12} + x^9 + x^8 + x^7 + x^4 + x^2 + 1$). BCH code is an abbreviation for Bose-Chaudhuri-Hochquenghem code, which is a multilevel, cyclic, error-correcting, variable-length digital code used to correct errors up to approximately 25% of the total number of digits [26]. The first number listed in the parenthesis (63, 51, 5), denotes the length of the block inclusive the 12-bit checksum. 51 is the length of the message or payload (63 bits – 12 bit checksum = 51 message bits). The last number is the HD_{min} of the polynomial for this block size. By multiplying the 12-bit BCH code with the following terms, ($x + 1$) and ($x^3 + x +1$), we acquire a 16-bit polynomial. Each factor expanding the 12-bit BCH code affects the new 16-bit polynomial in the following manner:

[26] Federal Standard 1037C, "Telecommunications: Glossary of Telecommunication Terms."

$(x + 1)$: As long as the generator polynomial contains this factor, any odd number of errors are detected [27].

$(x^3 + x + 1)$: As the polynomial has a factor of at least three terms, all double-bit errors are detected [27].

The polynomial that constitutes the expanded BCH code is the actual generator polynomial used by the safety layer to protect the acks;

$g_{16}(x) = 11111100000111001\ (x^{16} + x^{15} + x^{14} + x^{13} + x^{12} + x^{11} + x^5 + x^4 + x^3 + 1)$.

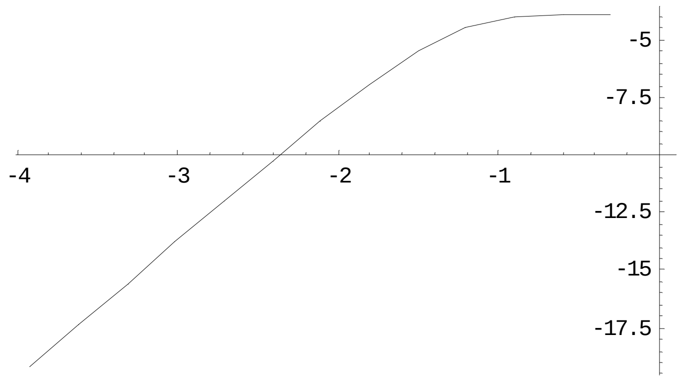


Figure 5-7, X-axis: FF bit error rate, Y-axis: Residual error rate. This graph was generated with Mathematica and the units on the axes are powers of 10.

The graph in Figure 5-7 shows the calculated residual error rate for the ack transmission as a function of the bit error rate on the FF signal. The graph shows that the requirement of $R < 10^{-14}$ is satisfied for bit error rates less than 10^{-3}. In order to understand the significance of this, we have to consider the signal to noise ratio on the FF signal bus needed to provide this bit error rate performance and consider whether or not this signal to noise ratio is provided by the FF standard under normal operating conditions.

The signal on the FF bus is Manchester encoded with the following bit signal representation [28]:

[27] Larry L. Peterson & Bruce S. Davie. "Computer Networks – A Systems Approach" Morgan Kaufmann Publishers Inc.

[28] Ferrel G. Stremler. "Introduction to Communication Systems," section 9.9.3. Addison Wesley.

$$f_1(t) = \begin{cases} -A & 0 < t \le \frac{T_b}{2} \\ A & \frac{T_b}{2} < t \le T_b \end{cases}$$

$f_2(t) = - f_1(t)$

where A is the signal amplitude, T_b is the signal duration (bit duration) and $f_1(t)$ and $f_2(t)$ represent the bit values 1 and 0 respectively.

This representation is equivalent to an antipodal binary signalling-scheme. The only difference is that the Manchester signal has one phase shift during the bit duration while the binary antipodal signal has no phase shift during the bit duration. This means that the Manchester signal requires the double bandwidth of the antipodal signal for the same bit duration or data rate. This means that for the same signal strength the signal to noise ratio for the Manchester signal will be 3 dB below the signal to noise
ration of the antipodal signal for the same additive noise situation.

The theoretical bit error curve for binary signal is copied from [29] and shown in Figure 5-8. The corresponding curve for Manchester signalling is included as a curve shifted 3 dB to the right.

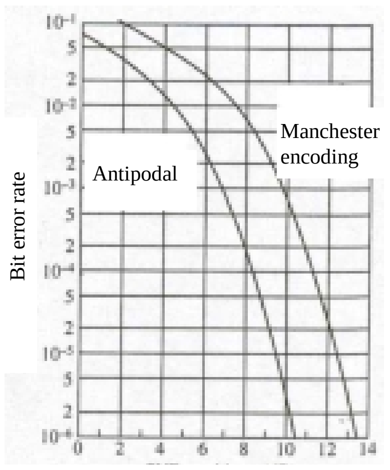


Figure 5-8, Signal to noise ratio per bit (dB).

From Figure 5-8 it is then possible to read out the correspondence between bit error rate and signal to noise ratio per bit for a Manchester signal.

[29] John G Proakis. “Digital Communications,” Figure 5-2-4. McGraw-Hill Book Co.

Earlier in this section it was shown that the proposed 16 bit CRC will satisfy the residual error requirement of $R < 10^{-14}$ for bit error rates less than 10^{-3}. From Figure 5-8 we can see that a bit error rate less than 10^{-3} requires a minimum signal to noise ratio per bit of 10 dB. This seems to be a very reasonable requirement that will be satisfied for the FF communication bus under normal operating conditions. Hence, the proposed 16-bit CRC is a suitable choice for securing $R< 10^{-14}$ for the transmission of acknowledgements.

5.3.1.4 24-bit CRC

The 24-bit generator polynomial implemented in the safety layer is used to generate a checksum for the safety frames. The block length of a safety frame, exclusive the checksum it self, is five fields and one byte long (the extra byte is the status-byte of the sixth field). This is a total of 208 bits. The 24-bit CRC polynomial is a binary BCH code (255, 231, 7). The generator polynomial is

g24(x) = 1101110111010000110110101
$(x^{24} + x^{23} + x^{21} + x^{20} + x^{19} + x^{17} + x^{16} + x^{15} + x^{13} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + 1)$

Block length inclusive 24-bit checksum is 255, block length exclusive the 24-bit checksum is 231, Hamming distance is 7 for the block length.

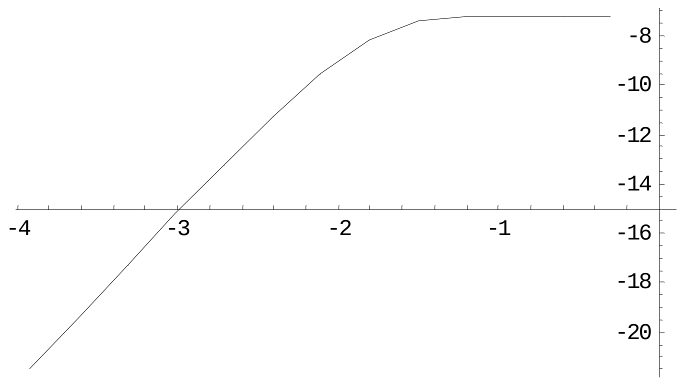


Figure 5-9, X-axis: FF bit error rate, Y-axis: Residual error rate. The units on the axes are powers of 10.

The graph in Figure 5-9 shows the calculated residual error rate for the safety frame transmission as a function of the bit error rate on the FF signal. The graph shows that the requirement of $R < 10^{-14}$ is satisfied for bit error rates less than 10^{-3}. Hence, the proposed 24-bit CRC is considered a suitable choice for securing $R< 10^{-14}$ during the transmission of safety frames.

5.3.2 Corruption of Sender and Receiver Addresses

The safety layer addresses the problem related to the lack of sufficient addressing information in the Publisher/Subscriber VCR by providing an arrangement for addressing the safe function blocks. Each safe function block is assigned a unique 16-bit address. These addresses are independent of the conventional FF naming scheme. The addresses provided by the safety layer are used to identify the parties in a communication relationship between two safe blocks. Each safety frame includes both the sender's and the receiver's address. This enables the receiving function block to validate the destination and the origin of the frame. A regular function block would not understand this addressing. The safety-related addresses are static and do not change during operation, but they can be changed during configuration.

5.3.3 Inaccurate Timing of Transmission

The communication between two safe function blocks can at a high level of abstraction be described as a scheduled exchange of safety frames and acks. This high level safety-critical communication is controlled by two separate events, namely; regularity of function block execution and reception of "Compel Data" messages from the LAS.

Firstly, the safety layer depends on that the function block, it is a part of, executes regularly. If the function block do not execute, the safety layer can not perform its functions like updating the CRC, the safety-related addressing and other information used to validate and monitor the safety-critical communication. In section 2.3.1 we read that System Management (SM) in each device administers the execution of function blocks according to an application clock it maintains (See Figure 5-10). If SM fails to trigger a safe function block to execute, the block's safety function is not executed.

Secondly, the high level transmission of safety frames and acks is dependent on "Compel Data" messages from the LAS. This is quite obvious, because, if a function block do not receive a "Compel Data" message from the LAS, the content of the function block's buffer is not published on the bus. For example, the Publisher that misses out on a "Compel Data," may never get to send a field, because the next time the function block executes, the buffers are over-written with new data. The previous field is lost forever and the complete safety frame the field is a part of is destroyed.

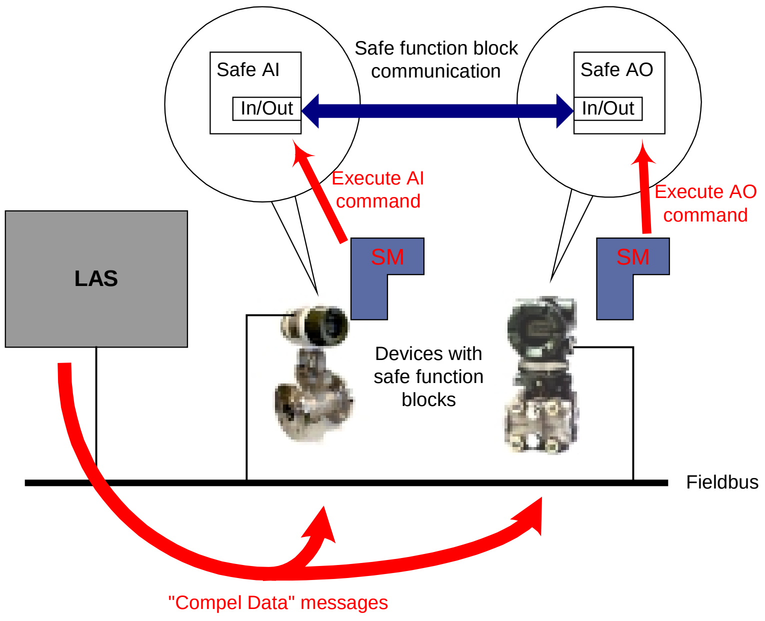


Figure 5-10, The communication between two safe function blocks consists of two events (marked as red arrows): 1. The commands from System Management (SM) in each device telling the safety layer to update its safety variables related to communication. 2. The arrival of “Compel Data” messages from the LAS forcing the blocks to publish the safety variables.

In order for a safety layer to be able to accurately time transmissions of safety frames, it must monitor both the frequency at which the function block executes and the arrivals of “Compel Data” messages.

The safety layer maintains two *timestamps* that enables its safe function block to determine whether “Compel Data” messages are arriving correctly according to the schedule. One timestamp is used to hold the time of the previous sent or arrived frame, depending on if the function block is on the sending or receiving end of a “safe communication relationship.” The second timestamp is used to hold the time of the newly sent or arrived frame. These two timestamps are used to check that the time lag between the sent (or arrived) safety frames are within the correct interval (not too late neither too early). These timestamps are sampled and updated inside the function block algorithm. Section 5.4 continues the discussion of the timestamps.

The safety layer includes a watchdog-function to monitor that SM is triggering function block executions correctly. The watchdog-function is not a part of the function block, and is therefore *not* invoked by SM. It runs in loop and executes periodically. The interval at which the watchdog is executed is equal to the loop time. This means that the watchdog-function is invoked once for each time the safe function block executes. If a function block is configured to run every second, the watchdog must also be configured to run at this interval. The watchdog works similarly to the timestamping of safety frames. Each time the algorithm in the safe function block is started by SM, the time is sampled and stored in a variable. The watchdog’s assignment is to sit in the background and at regular intervals compare the current time with the time stored in the variable holding the time of the function block's last execution. Function block execution is deterministic, thus if the difference between the two times is longer than a preconfigured value, the watchdog flags an error.

It is especially relevant to ensure that transmissions are accurately timed for output function blocks, like an AO-block. An output block controls an actuator that can physically affect the process. It is therefore crucial that an output block by itself is able to detect loss or interruption of communication. The consequences of a communication failure are highly application dependent. Some applications have a high degree of inertia, and can tolerate loss of communication for some time. The system can wait to see if the communication is restored. If not, a predetermined action is performed to shut the process safety down. Other systems may need to take action more rapidly, sometimes there is no time to wait to see if the communication can be re-established. In that case must the output block just initiate some sort of safety procedure on its own.

5.3.4 Wrong sequence of data packages

The safety function incorporates a frame counter chosen to be 32 bit long. Practical implementation considerations of the length of the counter are discussed in section 6.1.2. The counter operates as a sequence numbering of safety frames transmitted between two safe function blocks. The frame counter sequence numbers just whole *logical* frames, not each field separately transmitted. This counter is therefore incremented once every sixth macro cycle.

5.4 Safe Function Block Design

This section describes specific details directly related to the implementation of the safety layer. The safety layer mainly consists of a set of variables and some related software routines. Each variable and their intentions are discussed in detail.

There are six variables that together form the spine in the safety frames: `fieldNo`, `sender`, `dest`, `frameCounter`, `newTime` and `prevTime`. Additionally there exist four help-variables used for storage of temporary data during computations and conversions. These variables are collected together to form a C-structure (struct). I have called this structure `safetyParams`, an abbreviation for "safety parameters." This structure contains the majority of the parameters necessary to make a function block safe. The structure is displayed below in Listing 1.

```
typedef struct safetyParams_t
{
      uint16           fieldNo;          /* 1 */
      union{
            uint16 num;
            unsigned char c[2];
      }sender;                           /* 2 */
      union{
            uint16 num;
            unsigned char c[2];
      }dest;                              /* 3 */
      union{
            uint32 num;
            unsigned char c[4];
      }frameCounter;                      /* 4 */
      union{
            float f;
            unsigned char c[4];
      }floatTemp;                         /* 5 */
      union{
            uint16 num;
            unsigned char c[2];
      }shortTemp;                         /* 6 */
      float             errorFlag;        /* 7 */
      FF_Time           newTime;          /* 8 */
      FF_Time           prevTime;         /* 9 */
} safetyParams_t;
```

Listing 1, The type definition of safetyParams_t.

5.4.1 struct safetyParams_t

`safetyParams_t` is a data type defied as a "C" structure that holds safety relevant data. Each of the variables shown in Listing 1 is described individually below. Many of the struct-variables are defined as a `union`. These unions always include a character array, of two or four unsigned characters depending on the data type of the other variable declared in the union. I have done this to simplify the insertion of the variables in the unsigned character array used to hold the argument for the computation of the CRC checksum.

uint16 fieldNo:

An unsigned short variable that keeps track of the fields in a safety frame. This is a local counter managed by the function block to keep track of the fields in the safety frames. `fieldNo` is used by a safe function block to indicate what type of data (actually, which field) the block shall send or expect to be receive. When a function block is told by System Management to execute, the block reads its `fieldNo` variable. If for instance the block is a Publisher and its fieldNo = 1, then the block knows that it are supposed to send the address field (See Figure 5-3 and 5-4 for the contents of the different fields). The safe block prepares to send the field by copying both its own address ("`sender`" – see next paragraph) and the address of the receiving block ("`dest`" – see two paragraphs below) into its output parameter. The status byte of the output parameter is assigned with the value of an address field identifier bit. This bit is used by the frame destination (the Subscriber) to recognise the content of the arriving field as addressing information. Just before the function block is finished executing, it increments `fieldNo` with one. The safe function block is now prepared to transmit field number 2 of the frame next time the block algorithm is invoked by SM. The "safe-Subscriber" is using its local `fieldNo` counter to tell which field is arriving. When the “safe-Publisher” has sent the sixth and final field, a whole safety frame has been conveyed from the sender to the receiver. The two function blocks prepare themselves for the next safety frame by setting their local `fieldNo` counters back to 0. It is therefore absolutely essential for two communicating safe function blocks to be synchronised and have their `fieldNo`-variables loop “in phase” from 0 to 5. A “safe-Publisher” and a “safe-Subscriber” must have identical values on their `fieldNo` variables during the same macro cycle. If a Subscriber’s `fieldNo` is 3 when the Publisher’s `fieldNo` is 2, the Subscriber is not able to interpret the field correctly when it arrives. When the sender has `fieldNo` = 2, it is sending the frame counter. The receiving block must also have `fieldNo` = 2 to interpret the information in the arriving field as the frame counter. It is vital for the safety function that this counter is synchronised and “in phase” at all times between the communicating safe function blocks.

uint16 sender:
A 16-bit variable used to hold the address assigned to the safe function block considered being the sender (Publisher) in a safe communication relation. The value of this variable must not be changed during operation. It is not declared as a constant since it must be possible to change it during configuration of the system. This address does not have anything to do with the name of the device the function block resides.

uint16 dest:
`dest` is an abbreviation for “destination.” `dest` is similar to `sender`, but this variable holds the safety-related address of the Subscriber of safety frames.

uint32 frameCounter:
This 32-bit variable serves as the sequence numbering of the logical safe data frames. The `frameCounter` is incremented with 1 for each complete safety frame being sent or received. The `frameCounter` is used to give a sequence number to the frames and allows the parties in a safe communication relationship to monitor the progress of their dialog.

float errorFlag:
This variable is initialised to 0 and is set t another value when an error is detected. This flag can be set during the execution of the block algorithm or by the watchdog-function. When an error is detected and the error flag is set, a predefined action must be taken. This action is completely application dependent, therefore have I chosen to just set this flag when an error is detected. Depending on the type of detected error, the `errorFlag` can be assigned a certain value. This value can later be used to determine what caused the error and if any repairs must be done.

float floatTemp and **uint16 shortTemp:**
Variables used to store temporary values during computations. These variables are also used to convert values between different data types, especially during insertion of data into the array used as argument for CRC calculations.

FF_Time newTime and **FF_Time prevTime:**
`newTime` and `prevTime` are two timestamp variables used to validate the safety frames in a time domain. The timestamping of the frames has nothing to do with the data read from the process control. It is not an indicator on when the data in the first field of a frame was sampled from the environment. The timestamps are primarily used to record a point in time when the safety frames are sent. The timestamps allow the safety layer to monitor the frequency of frame transmissions/receptions and to check whether frames have arrived too late or too early.

`newTime` holds the time of day when the first field in a frame is sent. This time can be local time or an absolute time, depending on the time the Time Master distributes. The Time Master is a device on the bus responsible of distributing the time. If the time in the Time Master is correctly set, the application time is the number if 1/32 ms periods that have passed since January 1, 1972. `newTime` is sampled just before the function block is finished executing for the first field. FF_Time is a 64-bits time type and can hold the application time. The application time in a FF device comes from the Time Master (Ref. section 2.3.1.3).

`prevTime` this timestamp declares the time when the first field of the *previous* frame arrived. This timestamp is used to check the validity of the `newTime`.

5.4.2 Safety related variables not in safetyParams
In addition to the `safetyParams`-struct, three (four) other variables must be added to a function block in order to make it able to communicate more reliably with other safe function blocks. These are shown in Listing 2:

```
FF_VsFloat        fVal;       /* NOT safety related! */
FF_VsFloat        sIn / sOut;
FF_VsFloat        ack;
char              CRCarg[25];
```

Listing 2, Additional variables to the struct of “safe parameters” required in safe function blocks.

`FF_VsFloat` is one of three possible data types a function block’s I/O parameters may have. Recalling form section 5.2 and Table 5-1 that the abbreviated name stands for “Value and Status Float.” The FF_VsFloat is a type defined as a struct containing a float variable called f, and a char status, hence the name “Value and status.”

FF_VsFloat sIn / sOut (“Safe Input” and “Safe Output”):
These two I/O-parameters are respectively used to receive and send acknowledgements between the safe function blocks. They are the basis for

enabling a two-way communication between the safe function blocks. These variables are the only visible difference in a configuration tool between a standard and a safe function block. In other respects should the name of a block also reflect whether it has a safety layer, e.g. can a regular AI-block be just called “AI-1”, whilst a safe AI block should be called something like “safe AI-1” to emphasise that it is not a regular block.

FF_VsFloat ack:
The `ack` variable is used to construct and store acknowledgement messages prior to their transmission. `ack` is of date type `FF_VsFloat` so that it can be easily placed in an output parameter for transmission. Like most of the other “fields” that are sent, do the status byte in the `ack`-field contain a field identifier, the acknowledgement identifier ACKBIT. The remaining four bytes of the ack contains the two LSB of the frameCounter, and two bytes of checksum. The function block at the publishing side of a communication relationship (“the ack-Subscriber”) always check that the previous frame it transmitted has been ack’ed correctly before it begins the process of sending a new frame.

char CRCarg[25]:
`CRCarg[]` is the argument array for the calculation of the CRC. This array is used for both the CRC-24 and the CRC-16. The array has a size of 26 bytes, capable of storing five fields and the ACKBIT. Each time a safe function block executes, it copies the field into the argument array. Every sixth execution, the `CRCarg` is filled up and the block is ready to calculate a checksum for the whole frame. This calculation is identical in both the sending and the receiving function block.

The receiving block is also responsible to make an acknowledgement message when a frame has been successfully received. The argument array is cleared and used again to hold the acknowledgement message. The original sending device also uses the `CRCarg` array to calculate the checksum for the acknowledgement message. This is done before the work on building a new frame is begun.

FF_VsFloat fVal:
This variable do not have anything to do with the safety function itself, it is included just for simulation purposes only. The function block uses this variable to store simulated input values between block executions. Each time a process value is simulated, the function block just reads the value of this variable and increments it with 1,2. When `fVal` reaches a value above 22,2, it is set to 0 (See the `simulateInput()`-function in Appendix 8.2.1).

5.4.3 The Watchdog function

Execution of the watchdog-function is based around a data type called `wDog_t`. This data type have I defined in the following way:

```
typedef struct wDog_t
{
      FF_Time     now;
      FF_Time     FBtime; /* sampled at each FB execution */
} wDog_t;
```

Listing 3, Type definition of wDog_t.

Each time a safe function block executes, the time is sampled and placed in a variable of type `wDog_t`, for example called `wDog`, in the `FBtime` element. The time of the function blocks last execution is therefore always stored in `wDog.FBtime`. The watchdog-function executes once every macro cycle. The watchdog first samples the current time and places it in the `wDog.now`-variable. `wDog.now` and `wDog.FBtime` is compared by subtracting `FBtime` from `now`. If the absolute value of the difference is larger than a preconfigured value, one of three things can be wrong. Either has SM "forgotten" to start the function block, or the block's algorithm was invoked too early or too late. If the watchdog detects a delay or absents of block execution it notifies the surroundings by setting the `errorFlag` to a specific value.

5.4.4 CRC-Implementation

CRC algorithms can be implemented both in software and hardware. The CRC routines in the safety layer are operating at the application level and must therefore be implemented in software. Implementations of CRC in software can be done in a variety of ways. I have chosen to implement the CRC routines in the safety layer with high speed a table driven algorithm. Table-oriented algorithms usually produce higher speeds, but at the expense of memory usage. Before I describe how a table driven algorithm works, I will outline a straightforward and slow software implementation using a division register.

The division register must be able to contain the same number of bits as the length of the chosen generator polynomial. If a 16-bit polynomial is used, the register must be able to hold exactly 2 bytes. The simple CRC algorithm described below is from "A painless guide to CRC error detection algorithms" [30]. To perform a polynomial division the following must be carried out:

[30] Ross N. Williams. "A painless guide to CRC error detection algorithms" http://www.repairfaq.org/filipg/LINK/F_crc_v3.html

```
Load a W bit register with ‘0’ bits, where W is the length of the
generator polynomial.
Augment the binary message string by appending W ‘0’ bits to the end of
the message.
While (more message bits)
      Begin
      Shift the register left by one bit, reading the next bit of the
      augmented message into registers rightmost bit position.
      If (a 1 bit popped out of the register during step 3)
      Register = Register XOR Generator polynomial.
      End
```

The first thing to be done is to initialise the register by loading it ‘0’ bits. After the register is cleared, the binary message string is shifted into the register from the right, one bit at a time. During the first W shift operations, only ‘0’ bits are popping out on the left-hand side of the register. These ‘0’ bits are the zeros initialising the register. If a ‘1’ bit pops out of the register in the while-loop, we have to XOR the current content of the register with the generator polynomial. The algorithm says that the polynomial shall be XOR’ed into the register each time a ‘1’ bit is shifted out. As long as there are ‘0’ bits popping out, do noting (except keep shifting message bits into the register). When the least significant bit (the rightmost bit) of the message bit has been shifted out of the register the polynomial division register holds the remainder of the division. This remainder is the checksum. The checksum is appended to the message before it is transmitted.

A table driven algorithm shows a much greater performance in terms of speed compared to the straightforward algorithm. A table driven algorithm processes the binary message in units larger than one bit. The most common unit used is a byte. A table driven algorithm works somewhat different than the simple algorithm, but the two algorithms produces the same result. A table driven algorithm shifts the register left by one byte, reading in a new message byte from the right. The byte just rotated out of the register is used as an index of a table. The value looked up in the table is XOR’ed into the register. This is done until all message bytes have been processed through the register. The register holds the checksum when the algorithm has run to completion.

The table holding the values XOR’ed into the register is related to the generator polynomial. To generate the data for the lookup table, we need to generate the CRC result for all 256 possible combinations of the message-byte popping out of the register. The table is, of course, constant so it can be pre-computed at initialisation time or loaded as a constant into memory. For a table driven algorithm to process the CRC, all it has to do is a table lookup and a couple of XOR’s. The implementation of the table driven algorithms can be seen in appendix 8.1.1. For more details on CRCs and how the table driven algorithm works, I recommend [30] and [24].

5.4.5 Demo Implementation of the Safety Layer

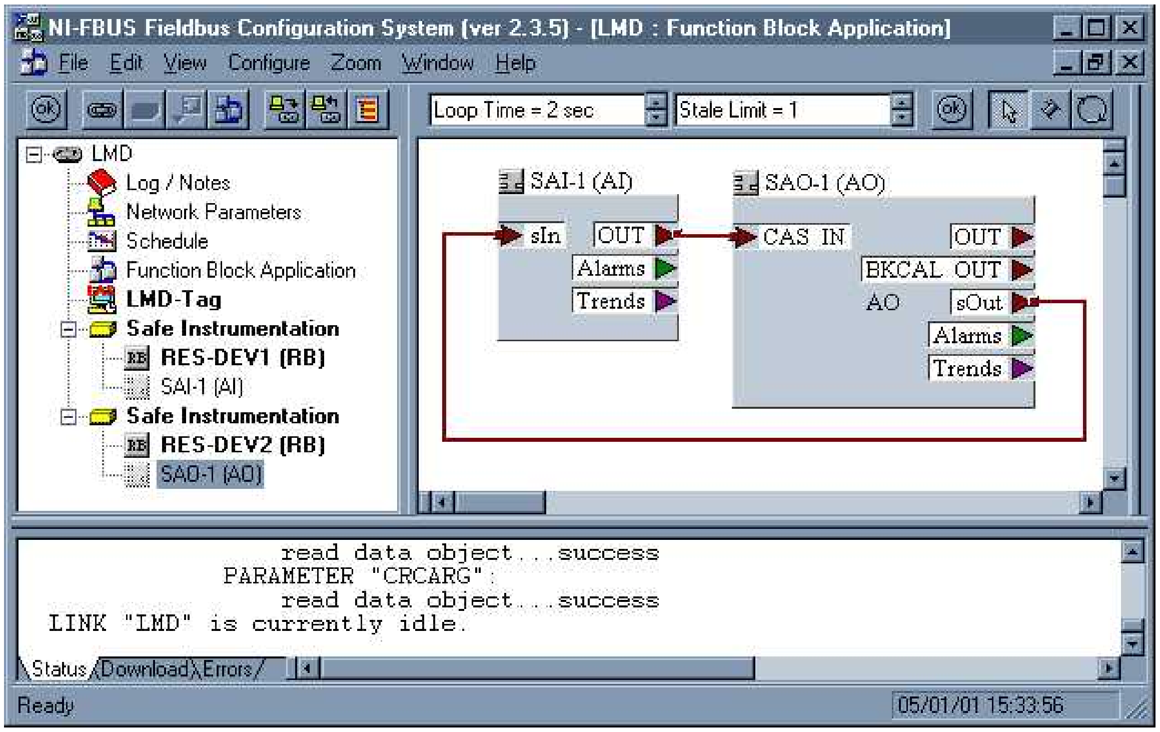
In part 2 of the “Function Block Application Process Foundation Specification” [19] we can read that the best way to make a new function block is to base the block on an already existing block. The procedure aims at first choosing a standard function block that closely match the new function block's variables and functionality and use it as a basis for implementing the new block.

For my demo implementation I picked two random types of function blocks which I customised by adding a safety layer. The type of function blocks I chose for my demo system was irrelevant from a safety-critical communication view, I only needed a block producing and sending data (a Publisher) and a block capable of receiving the data (a Subscriber). I ended up with an analog input (AI) and an analog output (AO) block.

To convert an ordinary and standard function block into a safe function block, the block must be equipped with a `safetyParams`-struct and the other safe parameters discussed in the previous section:

```
safetyParams_t    sParams;    /* Contains safety variables */
FF_VsFloat        fVal;       /* Simulates process values */
FF_VsFloat        sIn/sOut;   /* Receives ack / Sends ack */
FF_VsFloat        ack;
char              CRCarg[25];
```

Depending on whether the function block is a Subscriber or a Publisher, the function block gets an additional I/O parameter to respectively send or receive acknowledgements. In addition to these variables the block is equipped with a watchdog-function (See Appendix 8.2 for an example of a block implementation). The devices DDs must also be edited to reflect the modifications carried out on the function blocks. The modified DDs can be seen in Appendix 8.3). Figure 5-11 shows how I configured the demo system. I ended up putting the safe blocks in separate devices and configured them to send data every two seconds. I chose this relatively low loop time because any lower loop time would have made it impossible to monitor the execution of the function blocks with the equipment I had access to. This application or arrangement of function blocks are, as far I know, not of any practical relevance for process control, it is exclusively used for *demonstrating the communication aspects of my safety function between two arbitrary function blocks*.



```
              read data object...success
          PARAMETER "CRCARG":
              read data object...success
  LINK "LMD" is currently idle.
```

Figure 5-11, In the left panel of the Configurator we can see two fieldbus devices symbolized as yellow boxes. We can see that each device contains two blocks, one resource block and one function block. The right panel displays the how the blocks are connected with two linkages.

I will now describe how a safety frame is transmitted from the safe AI-block to the safe AO-block in the demo system. Let us say that device 1 contains the safe AI-block (SAI-1) and the safe AO-block (SAO-1) is situated in device 2.

The whole process starts with System Management in device 1 recognises that its time for the safe AI-block to execute. The Function Block Shell in the device makes a call to the callback function `cbExec()` with a handle to the AI-block. Inside the `cbExec()` I have implemented a `switch` statement that tests whether the blocks `fieldNo` matches a number of constant integer values ranging from 0 to 5. When a case matches the `fieldNo`, the actions related to the specific field are invoked. The first time the AI-block executes, the `fieldNo` is 0 and the block generates an acknowledgement. This acknowledgement represents the ack the SAI-block expects to receive for the frame it is preparing to start sending. The ack is made at this point because the SAI-block's `frameCounter` is incremented when field 2 of the current frame is sent and at this point is `CRCarg` free to be used. When SAI is in the middle of a frame transmission, `CRCarg` is filled with the contents of the fields in the frame. After the expected acknowledgement is made, a control loop input is simulated. This input is copied to `CRCarg` and the output parameter of the AI-block for transmission. Before the callback function is finished and returns, `fieldNo` is incremented so that the next field will be "inserted into the frame" next time the function blocks executes. The next thing that happens is that the LAS now issues a "Compel Data" message to device 1, which is in turn is forced to publish the contents of its output buffer. The simulated process value in the output buffer of the AI-block is transmitted via the bus and inserted into the input buffer of the AO-block. The System Management in device 2 is now aware of that it is time to start the execution of the safe AO-block (SAO-1). The FB shell in device 2 makes a call to `cbExec()` with a handle to the safe AO-block. When the AO-block executes the `fieldNo` is 0, this is an indication that the received data, is control loop process data. The input buffer is inserted into the AO-block's `CRCarg`, the `fieldNo` is incremented and the function returns. The first of a total of six macro cycles is now finished. During the next five macro-cycles, the same procedure is followed, but the `fieldNo` is one larger for each block execution and different fields of the frames are therefore copied into the output buffer and transmitted. When the AO-block executes for the sixth time, its `fieldNo` is 5. The AO-block calculates a checksum using the contents of its `CRCarg` as argument. `CRCarg` has been gradually filled with data arriving from SAI-1. If the locally calculated checksum is equal to the checksum SAO-1 received from device 1, the frame has arrived correctly with a very high probability (Ref. section 5.3.1.3 and 5.3.1.4). The AO-block then generates an acknowledgement message by taking the two LSB of the AO-blocks frameCounter together with the acknowledgement identifier and uses this to calculate a 16-bit checksum. The two LSB along with the checksum and the acknowledgement identifier are copied into the safe output buffer of the AO-block. A complete safety frame has been transmitted successfully from SAI to SAO. What happens now is that the LAS issues a "Compel Data" message to SAO that forces the block to publish the ack to SAI. When SAI is told by its SM to execute again and start on the laborious process of sending a new safety frame, it first checks if it has received an ack from SAO for the previous frame.

If the sender or receiver detects a communication failure of any kind during operation like erroneous addressing, wrong sequence number or absence of ack and so on, a predefined action must be taken. This action is totally dependent on the application. In my test implementation I just flag the error by setting a parameter to a certain value depending on what failure the safety function has detected, and the whole safety function halts.

Chapter 6 Discussion Of The Solution

I have now shown how the communications in a standard process control system can be made safer with a high-level safety layer running as a part of the user-application. In this chapter I will look at different aspects of the safety layer as described in the previous chapters. I will discuss why the proposed safety layer is not the optimal solution and point in directions of alternative solutions. Due to lack of time I have not been able to closely investigate and implement the issues discussed below.

6.1 Why The Proposed Safety Layer Is Not Optimal – Weaknesses and Uncertainties

This section discusses weaknesses with the design of the safety layer I discovered as my work progressed. I will also consider aspects I feel may could have been done better.

6.1.1 Weaknesses Related To The Addressing

There are a couple of weak or uncertain spots related to the addressing scheme in the safety layer. First would I like to raise a question related to the address-size of the safe function blocks. Do we really need 16-bit addresses or can we manage with just 8 bits? I do not have an opinion whether 256 addresses would suffice in a safety system. If not, we have to stick with the 16-bit addresses as they already are described earlier. Address space is of course application dependent. In practice one can think that 8 bits are enough to cover most safety systems, but in theory one can imagine larger systems requiring up to 2^{32} different addresses. If 8 bits were enough, we would free two bytes in the address field and use them to send other information.

Another problem related to addressing is that acknowledgements do not incorporate addressing information. The acks should incorporate addressing for the same reasons the safety frames include such information. It should be possible for an “ack-Subscriber” to validate the acks origin and destination. I have also realised that the ACKBIT and the other field identifiers are obsolete. Each field arriving is identified with the local field number and the field’s contents. In addition do the safety layer have the CRC which functions as a “field identifier” in such a way that it will detect any erroneous fields or fields arriving out of order. By choosing not to use the status byte to transport field identifiers, valuable space is liberated and can be used to hold other redundant safety related data.

From the discussion above we can construct a new ack-message and safety frame. By adopting an 8-bit addressing scheme and skipping the field-identification bits the safety frame can shrink with one field, into a total of five fields. We could further reduce the safety frame into a total of four fields by implementing a rollover-check for a 1- or 2-byte `frameCounter` (See Section 6.1.2). If we in addition choose to include only the least significant byte (LSB) of the `frameCounter` in the ack, the acknowledgements could contain more information that can be used for improved validation. An acknowledgement would have the following structure: 8 bit sender, 8 bit receiver, LSB of frame counter and 16 bit CRC. This acknowledgement contains enough data enabling the original frame sender (which is the receiver of the ack) to validate the ack in a value domain. The watchdog is used together with the `frameCounter` to validate the ack in a time domain. In a safety layer using a 16-bit addressing scheme, the ack-message can for example be distributed over two fields to allow the inclusion of addressing information in the acks.

6.1.2 Uncertain Issues Related To The frameCounter

Whether it is necessary to have a check for a rollover of the `frameCounter` can be discussed. A rollover of the `frameCounter` is of course determined of the amount of fields that forms a frame and the loop time of a safe function block. The loop time is in turn application dependent. In the design I describe in chapter 5, a complete safety frame consists of six fields (Process data, safety addresses, frameCounter, upper part of the timestamp, lower part of timestamp and the 24-bit CRC). If the loop time is one second the `frameCounter` will be incremented every sixth second and it will take more than 817 years for the counter to roll over. With a loop time of 100 ms `frameCounter` is incremented every 0.6 second, and this would result in a rollover of approximately 82 years. I think it is very likely to assume that during a period of more than 80 years, either the safety system would have been replaced or at some point shut down for maintenance. When powering up the system again, the counters will be reset. However there is nothing that renders it impossible to include a rollover check if that is required in a final implementation. I have not had time to measure how long a safe function block execution takes, therefore I have not been able to find out what the minimum loop-time a safe function block can be configured to run in. If a rollover-check is implemented an 8-bit counter would suffice. This would contribute to decrease the amount of fields in a frame and release bandwidth and the safety layer would be more effective.

6.1.3 General Weaknesses

The safety layer outlined in this thesis supports in reality only a one-to-one communication, this means that just two function blocks can more safely exchange data with the use of the safety function. It is possible to expand the safety function in a block to handle more than one communication partner. When having complex arrangements of function blocks (one function block communication with more than one block) the function blocks must have several sets of safety-related variables (`safetyParams`-structs). This solution occupies a lot of memory, and memory is a limited and scarce resource in the fieldbus devices. A solution involving several safety-structs is difficult to manage. A multi-safety structure configuration of a function block would require a large amount of time to execute. The implementation of the proposed safety layer is therefore not well suited to handle more than one-to-one communication.

The sequential sending of safety-relevant data degrades the throughput of process data quite substantially compared to normal, non-safe transmission. Compared to a solution where a whole safety frame can be transmitted at once, the performance provided by the safety layer is six times as slow. This can be compensated for down to a certain limit or point, by setting the loop time lower than the control loop normally would be configured with. With this adjustment of the loop time, the safety layer can roughly achieve the same throughput of process data as a normal process control application. Whether this compensation is possible is certainly application dependent. Safety systems with "hard" real time constraints (e.g. demands rapid responses and high throughput), will this safety function not be applicable. The safety layer's CRC sets a theoretical limit to how low the loop time can be configured. The CRC implemented in the safety layer was designed to handle a frequency of 100 telegrams per second (one telegram each 10ms), so it can theoretically handle a loop time of 10ms, that corresponds to a field transmission approximately each 1.6 mili second. I reckon this is just a theoretical performance. I have not been able to carry through calculations on a function block's execution time and how much added overhead the safety layer lead to. After all is FF designed for process control systems and not applications with strict real-time requirements.

Another weakness with the proposed safety layer is that it only supports the cyclic service. If for instance an operator accesses parameters in a "safe device" and makes changes, there is no higher-level mechanism, like the safety function, that can ensure that the changes are received correctly at the peripheral device. The safety function demands that the LAS' schedule is correct and not disturbed during the first couple of macro cycles. The safety function needs two executions to initialise all the variables in the `safetyParams`-data structure before it is fully operational. For instance must the `prevTime` parameter be set properly before the safety layer can be able to determine whether a frame is on time. Therefore is a correct functioning of the safety layer dependent on that the first pair of frames is transmitted without any errors. The safety layer is also dependent on that the application clocks in the devices are synchronised correctly at start-up time. Once the system is up and running correctly, the safe function blocks will detect communication errors and loss of communication.

6.2 Discussion Of Different Solution Proposals

After I got a reasonable overview of the FF-technology I started to consider different possibilities for design of the safety layer. I will in the following section shortly outline two of the ideas I considered and why I discarded them. The solution I ended up with is a combination of the two designs sketched below.

6.2.1 Safety Frame Without Fields

The first design I thought of was to define a safe function block I/O parameter consisting of a data structure containing all the necessary safety parameter, including a copy of the standard process data. This parameter can be compared with a safety frame. The safe parameter was meant to be in addition to, not instead of, the regular I/O parameter, so that system engineers can choose between ordinary or safe communication by linking the safe or regular I/O parameters between the blocks. The major advantage of this design is that all of the safety data (equivalent to a safety frame) are transmitted as a single unit, thus makes the serial transmission of "fields" unnecessary. The safety parameters are collected in a structural manner and they are separated from the other block parameters. This would allow a higher throughput because there is no need for sequential transmission.

This approach was unfortunately not feasible because of the limited amount of data types supported by the NI FB shell for I/O parameters. As I have mentioned earlier does this shell only support three types of I/O parameters (FF_VsFloat, FF_VsBitString, FF_VsDiscrete). My guess is that all of the FB shells on the market today support the same types of I/O parameters, as the three supported types are the minimum required in a process control application. I know that the FB shell offered from Softing supports only these three data types for function block I/O parameters [31].

6.2.2 Multiple I/O Parameters

Another design that avoids the reduced throughput implied by the sequential transmission is to define one I/O parameter for each safety-related variable. This allows a function block to send all safety parameters in one macro cycle/in one execution of the function block. No serial transmission of the safety parameters is necessary. No time is lost to ensure safe communication compared to the ordinary non-safe communication. This solution is very messy and there is really no structuring of the safety variables and they are mixed with the regular block parameters. This multiple safety I/O parameter design result in multiple linkages between two communicating function blocks. During configuration of such a system the system engineer has to administrate many parameters and linkages. This situation can quickly be difficult to follow and it could be easy to make wrong connections/links between I/O parameters. This solution would have to use the same type of acknowledgement technique as my final safety function.

6.3 Practical Problems Encountered During The Work Process

In this section I will summarise some reflections I made during the practical phase of this thesis (the part involving implementation, debugging and testing). I experienced both rewarding and entertaining moments, as well as frustrating and less satisfactorily periods. I even got my share of bad luck.

6.3.1 Inconsistent User Manuals

First of all I encountered the well known, and what sometimes seems almost inevitable, disagreement between the user manuals and reality. When writing this, I specifically have

[31] Softing GmbH, "FF Basic Field Device Application Interface," Version 1.61, 24 September 1999

in mind the guidebooks from NI. On a general level do the user manuals describe the most basic things of the Starter Kit in an orderly manner. The manuals were easy to read and their contents were understandable. They provided a good background material and helped me on my theoretical comprehension of FF, but the parts that dealt with practical issues related to both installation and implementation, the manuals did not agree with reality and the actual actions required to make progression. When I started to install the kit and later using it, the manuals lacked important details. I frequently experienced unwanted and unnecessary disruptions due to the inaccurate manuals. Under installation and testing of the FF development kit, I was daily in contact with the support team just to be able to get the equipment up and running. It was sometimes annoying that my work was delayed by activities not related to the problem solving of my thesis. It felt meaningless and I was frustrated to regularly "waste" time on issues remotely related to the actual problem of my thesis. When this is said, I must admit in National Instruments' defence that their starter kit is geared toward people who are already familiar with Foundation Fieldbus. It is a starter kit that is designed to help experienced people start with NI's particular fieldbus products. Since I was a new fieldbus user there was probably less painful ways to learn about FF than to start fumbling with equipment for "experts."

The first user manual related problem I encountered was when I wanted to test the Fieldbus Configurator (See Section 2.4) and get familiarised with the NI tools. The manuals said that one just had to connect the round cards to the power hub and they would automatically appear in the left panel of the Configurator (See Figure 5-10). This panel is the project window and displays all the configurable devices connected to the fieldbus that comes along with the starter kit. Each object can be configured by double-clicking on the icons in the project window, doing so opens new windows for configuring the objects. Even though I exactly followed the instructions in the manuals, nothing happened. I got a detailed guide from the NI support team telling me exactly what to do step-by-step, but still did not the round cards appear in the project window. After numeral exchanges of mails with support, we found out that the person at support and I had two different processors on our round cards. She had an Intel based round card whilst I had round cards with a Motorola processor. The user manual described the correct procedure for Intel based round cards. The Motorola round card does not show up in the configuration-utility until after firmware have been downloaded to it unlike the Intel-based round card.

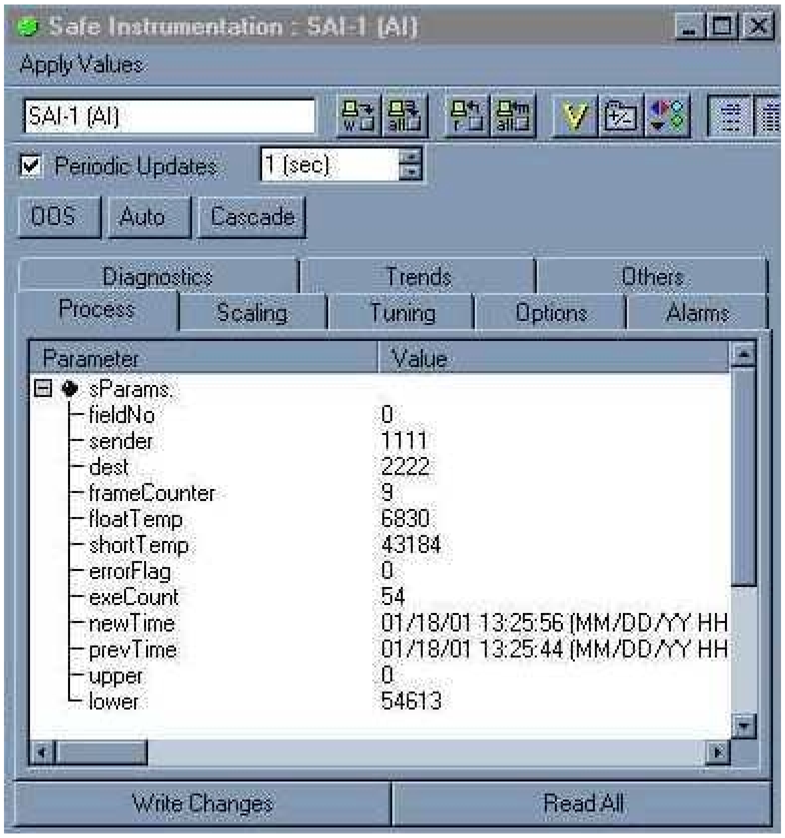
6.3.2 Compiler Problems

Another issue that is worth mentioning and caused me some worries in more than one way was the compiler during the implementation of the safety layer. Compiler related difficulties definitely delayed my progression the most. In the documentation from NI it said that the stack library provided with the Starter Kit, I had to link my function block code with, was made using the SDS CrossCode C/C++ version 7.11. The user manuals recommended using the same compiler or *a compatible* compiler to ensure maximum compatibility with the stack and the function block shell. This was all the information that was provided by the manuals on this issue. I think it is natural to interpret this statement in a way that one should think that a compatible compiler exists. I tried four different compilers, respectively Borland, Microsoft Developer Studio, Tornado II and National Instruments' Lab Windows. None were able to understand the object file format the library was compiled into. I checked the documentation for my compilers to see if they claimed to be compatible with SDS. I found nothing indicating compatibility. I contacted NI to ask whether they knew anything about the CrossCode Compiler, or if they could find out which object file format the compiler generated. They could not offer any help. When I contacted SDS I learnt that they had just merged with WindRiver (the one that delivers Tornado II). The people I spoke with at WindRiver were unfamiliar with the CrossCode Compiler and needed some time to locate information about that specific compiler. After a week or so, WindRiver could report that the object file format used in SDS was proprietary. There existed no such thing as a compatible compiler to the CrossCode from SDS, even though the NI user manuals indicated so. I then decided to try to get the pre-compiled stack library ported to a compatible format for one of the four compilers I had access to, and asked NI for help. NI was probably fed up of me nagging for help at this point and told me that they have stopped supporting the FF tools. They referred me to a partner company called "Fieldbus Inc." specialising in development tools. The people at Fieldbus Inc. were very helpful and were glad to provide me with an estimate of the cost to port the FF-stack library to a recognisable format. But at the same time, Fieldbus Inc. guaranteed that it would be less expensive for me to purchase the SDS compiler than for them to port the library. It was obviously time for contacting WindRiver again about the CrossCode compiler. To make things even more complicated and delaying my project even further, WindRiver told me that they had decided that the "end of life" for the CrossCode compiler was imminent, the CrossCode compiler was going to be phased out and they had stopped supporting it. For that reason was WindRiver not willing to sell me the CrossCode Compiler. This meat that I was in a situation where I had no development tools and the Fieldbus Device Starter Kit we bought from NI for over NOK 100.000 was as good as useless. After a lot of fuzz and e-mails back and forth I managed, with some help from colleagues at ABB, to borrow the CrossCode from WindRiver for a limited period of time, just long enough to implement the safety layer in two function blocks and set up a demo system. This whole process was very time consuming as most of the correspondence was conducted via e-mail. It was not unusual that responses arrived one or two weeks after my e-mail enquiries were sent.

Problems of this kind are difficult to foresee and in this case a consequence of bad luck, inexperience and ignorance shown by NI. When we purchased the Starter Kit from NI we could have conducted a more thorough research to find out whether additional tools were required, such as the Compiler. As I have mentioned earlier I think that NI could have spent more time on their user manuals to get them more accurate. Never the less, I have learnt a valuable lesson, and I will probably be faced with similar situations later.

6.3.3 Debugging And Testing Of The Safety Layer

As I mentioned in section 5.4.5.1 I have implemented a prototype of the safety layer. The prototype was implemented to demonstrate the feasibility of the proposed safety layer and that the safety layer detects the failure modes it is designed to do. I have not had right equipment required to conduct a proper and thorough test of my prototype implementation of the safety function. The only test that I have been able to carry through is to manually test the safety layer through the fieldbus Configurator-utility. The Configurator provides a window for each function block that can be used to view and change function block parameters and edit other settings. Figure 6-1 shows a block window for a safe AI block, SAI-1. The window displays the values of the safety layer's safetyParams-struct.



Parameter	Value
sParams	
fieldNo	0
sender	1111
dest	2222
frameCounter	9
floatTemp	6830
shortTemp	43184
errorFlag	0
exeCount	54
newTime	01/18/01 13:25:56 (MM/DD/YY HH
prevTime	01/18/01 13:25:44 (MM/DD/YY HH
upper	0
lower	54613

Figure 6-1, The block window. This window displays a function block's parameters as they appear in the block or you can choose which of the parameters you want to display. In this figure only the safetyParams-struct in the SAI-1 is block shown. The SAI-1 is here sending the first field of the tenth frame.

Actually was this picture taken during the development phase, therefore can we see additional help-variables that are not a part of the safety layer, but was used during the implementation. The values in the window can be updated periodically. I used this feature to monitor the safety parameters as the blocks executed. I also used the block window to change values in the function blocks to simulate errors. I inserted errors in the `sParams` structure and the `CRCarg` array, both in the sending and the receiving function block. All inserted errors were detected; faulty addresses, frames with wrong sequence numbering and timestamps, and errors in either the checksum or the message itself. I also physically disconnected one of the round cards from the "Power Hub" to se if the safe function block in the other device detected the loss of its communication partner. The safety layer in the function blocks detected all errors I inserted into the system.

I have tested the implementations of the CRC algorithms I coded. Both algorithms were tested separately before they were incorporated into the safety layer. Each algorithm was tested with two different bit-strings in the following way:

First I calculated a checksum for the bit-strings using my implementation of the CRC algorithm. Then I appended the checksum at the end of the bit-string and performed a polynomial division in Mathematica. The dividend was the original bit-string with the appended checksum, the generator polynomial was the divisor. If the quotient turned out to be 0, which it did for both algorithms, the implementation was very likely to be correct (See Appendix 8.1.2).

Chapter 7 Conclusion

During the work on this thesis I have been faced with challenges from a variety of technical disciplines, like fieldbus technologies, process control, safety systems and hardware related programming. I have also looked at issues within a branch of mathematics called coding theory through my studies of CRC routines. Most of these disciplines have not been covered in my previous studies. The effort required by me to acquire the necessary knowledge in order to solve the various problems encountered has been substantial.

Studies of the principles behind process control, fieldbus technology and how these two terms can be joined together started the work on this thesis. Next I moved on to the concept of safety and safety critical systems and the international standards applying to these systems. The phase including practical work such as design, implementation and other related activities offered many practical obstacles. I had to tackle problems, such as incompatible tools and inconsistent manuals. This represents an important experience that I will benefit from in my future career.

The problem specification in section 1.2 lists the central questions defining the main challenges of this thesis. Through my work presented in this document I have answered all of these central questions and can draw the following conclusions for my work:

- Through my studies of process control- and safety-critical systems I have proposed a general safety layer concept which can be used together with any fieldbus technology. I have also discussed a concrete design of the safety layer concept adapted to the technology from Fieldbus Foundation. My thesis has further been extended with a prototype implementation of this design to show that a practical safety layer implementation is feasible.

- According to IEC 61508 the probability of a hazardous error per hour in the communication subsystem in a safety-critical system fulfilling SIL 3 requirements, shall not be larger than 10^{-9}. In chapter 5 I assumed that a safety layer for FF will not decode messages at a rate higher than maximum 100 telegrams per second. This implies that the residual error rate for a FF SIL 3 safety layer must be less than 10^{-14}. I have proposed two CRC generator polynomials (a 16 bit and a 24 bit), incorporated in the safety layer that offers an error detection performance well inside this requirement.

- To be able to transmit redundant safety-related data, the proposed safety layer splits the data into units called fields and sends them sequentially. Because the safety layer is mainly implemented in the function blocks, the maximum amount of data possible to transmit at a time is five bytes. The most orderly way to send larger frames with safety-relevant data is to split the safety data into smaller units and send them sequentially. This concept can be used in conjunction with other underlying communication technologies where the size of the maximum transmission unit is smaller than the total amount of safety-relevant data.

- I have shown how it is possible to achieve two-way scheduled and time deterministic communication between two FF function blocks. The safety layer incorporates an additional I/O parameter in each function block. This extra parameter enable a pair of blocks to have an extra Publisher/Subscriber VCR based linkage between them. This linkage is “reversed” compared to the regular linkage and allows the recipient to publish scheduled messages in a similar manner as it receives messages from another publishing block. The bi-directional flow of messages between function blocks enables acknowledgements of safety frames.

I have also given my thoughts and recommendations on alternative solutions and designs of the safety layer. Through my work I have realised that there are some issues that need to be investigated further if my safety layer is eventually implemented in industrial solutions.

7.1 Further Work

The safety layer has only been demonstrated in a one-to-one communication mode. The safety layer can in principle be extended to support one-to-many communication. This will increase the computation load for a Publisher of safety frames as it must handle the reception of multiple acknowledgements. This obviously includes more code in the safety layer. It would have been interesting to determine the amount of extra code necessary to make the safety layer able to handle a configuration where a Publisher is broadcasting safety frames to more than one Subscriber? I have not performed any research on implementation limitations, such as how much memory is available in a device. Another related and interesting issue that should be subject to further investigation is performance testing of the safety layer and all other future safety layer designs, like determining how long a function block execution takes with different designs of the safety layer.

One of the most obvious issues to investigate further is the function block shell. It would be interesting to look at the possibilities of updating the function block shell to accept I/O parameters of other types than the three it does today (VsFloat, VsBitstring and VsDiscrete). I imagine a shell that accepts a composite safe data type similar to `safetyParams_t`. This solution would permit the safety layers to transmit whole safety frames instead of splitting the frames and sending fields. Additionally it would be exciting to compare all of the development tools available on the market and look for the possibilities for improving the tools.

Finally would it be interesting to investigate problems related to implementation methods and techniques that would contribute to increase a device’s safety, like n-version programming and diverse programming of a safety layer.

Further, I have identified the following list of unsolved crucial questions:

- How does the safety layer perform when the loop time is decreased?
- What is the lowest loop time the safety layer design can handle?

References

[1] Lars Lidström, “Using Foundation Fieldbus in Safety Applications.” ABB NOCRC, June 1999.

[2] “Fieldbus Foundation låter instrumenten styra själva.”Automation, tidningen för modern produktionsteknik, mars 1999 nummer 2.

[3] ANSI/ISA, “Application of Safety Instrumented Systems for the Process Industry,” ANSI/ISA-S84.01-1996.

[4] ANSI/ISA, “Identification of Emergency Shutdown Systems That Are Critical to Maintaining Safety in Process Industries,“ANSI/ISA S91.01-1995, ISBN 1-55617-570-1.

[5] Based on a translated foil from Tor Onshus’ presentation on Safety Critical Systems at ABB NOCRC, March 13th 2000.

[6] Paul Gruhn and Harry L. Cheddie, ISA “Safety Shutdown Systems: Design, Analysis and Justification.”

[7] Neil Storey, “Safety-Critical Computer Systems.” Addison-Wesley 1996.

[8] David J. Bak. “A factory floor ‘Safety Net’” - Global Design News, April 1999.

[9]ARC Advisory Group “Critical Control & Safety Shutdown System World Wide Outlook” – Market analysis and forecast through 2004.

[10] IEC 61508 – International Electrotechnical Commission (IEC) standard, IEC 61508 – “Functional safety of electrical / electronic / programmable electronic safety-related systems.”

[11] IEC 61511 – IEC standard entitled “Functional Safety: Safety Instrumented Systems for the Process Industry Sector.” This standard adheres to the main attributes established in IEC 61508.

[12] DIN V 19250 (Deutche Industri Normen, DIN) Control technology – “Fundamental safety aspects to be considered for measurement and control equipment.”

[13] M. J. L. Ochsner, Ken Beatty. Technical Papers of ISA, Networking and Communications on the Plant Floor – Volume 392, 5-7 October, “Benefits and challenges experienced by Foundation Fieldbus Installations.”

[14] Datamonitor, “Developments and Customer Opinion on Fieldbus,” July 1999.

[15] PROFIBUS-DP/PA - ProfiSafe, Profile for Failsafe Technology, V1.0. Document No. 740257.

[16] FOUNDATION^{TM} Specification. System Architecture. Document FF-800, Rev. 1.3, May 8, 1998.

[17] ISO 7498 – International Standards Organization (ISO) Open Systems Interconnect (OSI) Reference Model (RM).

[18] IEC 61158 – International standard for fieldbus for use in industrial control systems, IEC 61158-2/ISA-S50.02-1992 Part 2 Physical Layer Specification and Service Definition.

[19] FOUNDATION^{TM} Specification. Function Block Application Process. Part 2. Document FF-891, Rev. 1.3, May 8, 1998.

[20] FOUNDATION^{TM} Fieldbus Technical Specifications. Fieldbus Foundation, http://www.fieldbus.org

[21] National Instruments, “MC 68331-Based Fieldbus Round Card User Manual.”

[22] Nancy G. Leveson. “Safeware. System safety and computers. A guide to preventing accidents and losses caused by technology.”

[23]Paris Stavrianidis. “What Regulations and Standards Apply to Safety Instrumented Systems?” Control Engineering Online. http://www.controleng.com/archives/2000/ctl0301.00/0003we1.htm

[24] Jack W. Crenshaw, “Implementing CRCs.” Embedded Systems Programming, January 1992.

[25] G. Castagnoli, J. Ganz, P. Graber. “Optimum Cyclic Redundancy-Check Codes with 16-Bit Redundancy.“ IEEE Transactions on Communication, vol. 38, No. 1, January 1990.

[26] Federal Standard 1037C, “Telecommunications: Glossary of Telecommunication Terms.”

[27] Larry L. Peterson & Bruce S. Davie. “Computer Networks – A Systems Approach.” Morgan Kaufmann Publishers Inc.

[28] Ferrel G. Stremler. “Introduction to Communication Systems,” section 9.9.3. Addison Wesley.

[29] John G Proakis. “Digital Communications,” Figure 5-2-4. McGraw-Hill Book Co.

[30] Ross N. Williams. “A painless guide to CRC error detection algorithms” http://www.repairfaq.org/filipg/LINK/F_crc_v3.html

[31] Softing GmbH, “FF Basic Field Device Application Interface,” Version 1.61, 24 September 1999

Chapter 8 Appendix

Note: I have used the same naming conventions as used in the sample code from National Instruments. It is important to notice this if parts of my code are to be used in other non-NI implementations. The NI names are defined in several included header files and makes this implementation a "hybrid" of the C programming language. The NI naming scheme may be helpful to understand the names and types related to FF. In the safety.h (CRC) file I have not used the NI type definitions because I placed the code in a separate file.

8.1 CRC

8.1.1 CRC-Code and tables

```
#ifndef SAFE__H
#define SAFE__H

#include <stdlib.h> /* To be able to use size_t */
#include <fbtypes.h>
#include <stdio.h>
#include <string.h>

#define MAXBUF 64
#define ACKBIT 0x01    /* Bitpattern 00000001, indicates ACKNOWLEDGEMET (ACK). */
#define CRC24BIT 0x80  /* Bitpattern 10000000, indicates use of 24 bit CRC. */
#define CRC16BIT 0x40  /* Bitpattern 01000000, indicates use of 16 bit CRC. */
#define ADDRBIT 0x20   /* Bitpattern 00100000, indicates address field */
#define FCBIT  0x10    /* Bitpattern 00010000, indicates frameCounter field */
#define LOWERBIT 0x08  /* Bitpattern 00001000, indicates lower part of timestamp. */
#define UPPERBIT 0x04  /* Bitpattern 00000100, indicates upper part of timestamp. */

/* function prototypes */
unsigned short updateCRC16(unsigned short tempCRC, char *outPtr, size_t bufSize);
unsigned long updateCRC24(unsigned long tempCRC, char *outPtr, size_t bufSize);

static unsigned short crc16Table[256] =
{
       0x0000L, 0xf839L, 0x084bL, 0xf072L, 0x1096L, 0xe8afL,
       0x18ddL, 0xe0e4L, 0x212cL, 0xd915L, 0x2967L, 0xd15eL,
       0x31baL, 0xc983L, 0x39f1L, 0xc1c8L, 0x4258L, 0xba61L,
       0x4a13L, 0xb22aL, 0x52ceL, 0xaaf7L, 0x5a85L, 0xa2bcL,
       0x6374L, 0x9b4dL, 0x6b3fL, 0x9306L, 0x73e2L, 0x8bdbL,
       0x7ba9L, 0x8390L, 0x84b0L, 0x7c89L, 0x8cfbL, 0x74c2L,
       0x9426L, 0x6c1fL, 0x9c6dL, 0x6454L, 0xa59cL, 0x5da5L,
       0xadd7L, 0x55eeL, 0xb50aL, 0x4d33L, 0xbd41L, 0x4578L,
       0xc6e8L, 0x3ed1L, 0xcea3L, 0x369aL, 0xd67eL, 0x2e47L,
       0xde35L, 0x260cL, 0xe7c4L, 0x1ffdL, 0xef8fL, 0x17b6L,
       0xf752L, 0x0f6bL, 0xff19L, 0x0720L, 0xf159L, 0x0960L,
       0xf912L, 0x012bL, 0xe1cfL, 0x19f6L, 0xe984L, 0x11bdL,
       0xd075L, 0x284cL, 0xd83eL, 0x2007L, 0xc0e3L, 0x38daL,
       0xc8a8L, 0x3091L, 0xb301L, 0x4b38L, 0xbb4aL, 0x4373L,
       0xa397L, 0x5baeL, 0xabdcL, 0x53e5L, 0x922dL, 0x6a14L,
       0x9a66L, 0x625fL, 0x82bbL, 0x7a82L, 0x8af0L, 0x72c9L,
       0x75e9L, 0x8dd0L, 0x7da2L, 0x859bL, 0x657fL, 0x9d46L,
       0x6d34L, 0x950dL, 0x54c5L, 0xacfcL, 0x5c8eL, 0xa4b7L,
       0x4453L, 0xbc6aL, 0x4c18L, 0xb421L, 0x37b1L, 0xcf88L,
       0x3ffaL, 0xc7c3L, 0x2727L, 0xdf1eL, 0x2f6cL, 0xd755L,
       0x169dL, 0xeea4L, 0x1ed6L, 0xe6efL, 0x060bL, 0xfe32L,
       0x0e40L, 0xf679L, 0x1a8bL, 0xe2b2L, 0x12c0L, 0xeaf9L,
       0x0a1dL, 0xf224L, 0x0256L, 0xfa6fL, 0x3ba7L, 0xc39eL,
       0x33ecL, 0xcbd5L, 0x2b31L, 0xd308L, 0x237aL, 0xdb43L,
       0x58d3L, 0xa0eaL, 0x5098L, 0xa8a1L, 0x4845L, 0xb07cL,
       0x400eL, 0xb837L, 0x79ffL, 0x81c6L, 0x71b4L, 0x898dL,
       0x6969L, 0x9150L, 0x6122L, 0x991bL, 0x9e3bL, 0x6602L,
       0x9670L, 0x6e49L, 0x8eadL, 0x7694L, 0x86e6L, 0x7edfL,
       0xbf17L, 0x472eL, 0xb75cL, 0x4f65L, 0xaf81L, 0x57b8L,
       0xa7caL, 0x5ff3L, 0xdc63L, 0x245aL, 0xd428L, 0x2c11L,
       0xccf5L, 0x34ccL, 0xc4beL, 0x3c87L, 0xfd4fL, 0x0576L,
       0xf504L, 0x0d3dL, 0xedd9L, 0x15e0L, 0xe592L, 0x1dabL,
        0xebd2L, 0x13ebL, 0xe399L, 0x1ba0L, 0xfb44L, 0x037dL,
        0xf30fL, 0x0b36L, 0xcafeL, 0x32c7L, 0xc2b5L, 0x3a8cL,
        0xda68L, 0x2251L, 0xd223L, 0x2a1aL, 0xa98aL, 0x51b3L,
        0xa1c1L, 0x59f8L, 0xb91cL, 0x4125L, 0xb157L, 0x496eL,
        0x88a6L, 0x709fL, 0x80edL, 0x78d4L, 0x9830L, 0x6009L,
        0x907bL, 0x6842L, 0x6f62L, 0x975bL, 0x6729L, 0x9f10L,
        0x7ff4L, 0x87cdL, 0x77bfL, 0x8f86L, 0x4e4eL, 0xb677L,
        0x4605L, 0xbe3cL, 0x5ed8L, 0xa6e1L, 0x5693L, 0xaeaaL,
        0x2d3aL, 0xd503L, 0x2571L, 0xdd48L, 0x3dacL, 0xc595L,
        0x35e7L, 0xcddeL, 0x0c16L, 0xf42fL, 0x045dL, 0xfc64L,
        0x1c80L, 0xe4b9L, 0x14cbL, 0xecf2L
};


/* CRC lookup table 24 bit generator polynomial */
/* g(x) = x^24 + x^23 + x^21 + x^20 + x^19 + x^17 + x^16 + x^15 + x^13 + x^8 + x^7 + x^5
+ x^4 + x^2 + 1 */
static unsigned long crc24Table[256] =
{
        0x00000000L, 0x00bba1b5L, 0x00cce2dfL, 0x0077436aL,
        0x0022640bL, 0x0099c5beL, 0x00ee86d4L, 0x00552761L,
        0x0044c816L, 0x00ff69a3L, 0x00882ac9L, 0x00338b7cL,
        0x0066ac1dL, 0x00dd0da8L, 0x00aa4ec2L, 0x0011ef77L,
        0x0089902cL, 0x00323199L, 0x004572f3L, 0x00fed346L,
        0x00abf427L, 0x00105592L, 0x006716f8L, 0x00dcb74dL,
        0x00cd583aL, 0x0076f98fL, 0x0001bae5L, 0x00ba1b50L,
        0x00ef3c31L, 0x00549d84L, 0x0023deeeL, 0x00987f5bL,
        0x00a881edL, 0x00132058L, 0x00646332L, 0x00dfc287L,
        0x008ae5e6L, 0x00314453L, 0x00460739L, 0x00fda68cL,
        0x00ec49fbL, 0x0057e84eL, 0x0020ab24L, 0x009b0a91L,
        0x00ce2df0L, 0x00758c45L, 0x0002cf2fL, 0x00b96e9aL,
        0x002111c1L, 0x009ab074L, 0x00edf31eL, 0x005652abL,
        0x000375caL, 0x00b8d47fL, 0x00cf9715L, 0x007436a0L,
        0x0065d9d7L, 0x00de7862L, 0x00a93b08L, 0x00129abdL,
        0x0047bddcL, 0x00fc1c69L, 0x008b5f03L, 0x0030feb6L,
        0x00eaa26fL, 0x005103daL, 0x002640b0L, 0x009de105L,
        0x00c8c664L, 0x007367d1L, 0x000424bbL, 0x00bf850eL,
        0x00ae6a79L, 0x0015cbccL, 0x006288a6L, 0x00d92913L,
        0x008c0e72L, 0x0037afc7L, 0x0040ecadL, 0x00fb4d18L,
        0x00633243L, 0x00d893f6L, 0x00afd09cL, 0x00147129L,
        0x00415648L, 0x00faf7fdL, 0x008db497L, 0x00361522L,
        0x0027fa55L, 0x009c5be0L, 0x00eb188aL, 0x0050b93fL,
        0x00059e5eL, 0x00be3febL, 0x00c97c81L, 0x0072dd34L,
        0x00422382L, 0x00f98237L, 0x008ec15dL, 0x003560e8L,
        0x00604789L, 0x00dbe63cL, 0x00aca556L, 0x001704e3L,
        0x0006eb94L, 0x00bd4a21L, 0x00ca094bL, 0x0071a8feL,
        0x00248f9fL, 0x009f2e2aL, 0x00e86d40L, 0x0053ccf5L,
        0x00cbb3aeL, 0x0070121bL, 0x00075171L, 0x00bcf0c4L,
        0x00e9d7a5L, 0x00527610L, 0x0025357aL, 0x009e94cfL,
        0x008f7bb8L, 0x0034da0dL, 0x00439967L, 0x00f838d2L,
        0x00ad1fb3L, 0x0016be06L, 0x0061fd6cL, 0x00da5cd9L,
        0x006ee56bL, 0x00d544deL, 0x00a207b4L, 0x0019a601L,
        0x004c8160L, 0x00f720d5L, 0x008063bfL, 0x003bc20aL,
        0x002a2d7dL, 0x00918cc8L, 0x00e6cfa2L, 0x005d6e17L,
        0x00084976L, 0x00b3e8c3L, 0x00c4aba9L, 0x007f0a1cL,
        0x00e77547L, 0x005cd4f2L, 0x002b9798L, 0x0090362dL,
        0x00c5114cL, 0x007eb0f9L, 0x0009f393L, 0x00b25226L,
        0x00a3bd51L, 0x00181ce4L, 0x006f5f8eL, 0x00d4fe3bL,
        0x0081d95aL, 0x003a78efL, 0x004d3b85L, 0x00f69a30L,
        0x00c66486L, 0x007dc533L, 0x000a8659L, 0x00b127ecL,
        0x00e4008dL, 0x005fa138L, 0x0028e252L, 0x009343e7L,
        0x0082ac90L, 0x00390d25L, 0x004e4e4fL, 0x00f5effaL,
        0x00a0c89bL, 0x001b692eL, 0x006c2a44L, 0x00d78bf1L,
        0x004ff4aaL, 0x00f4551fL, 0x00831675L, 0x0038b7c0L,
        0x006d90a1L, 0x00d63114L, 0x00a1727eL, 0x001ad3cbL,
        0x000b3cbcL, 0x00b09d09L, 0x00c7de63L, 0x007c7fd6L,
        0x002958b7L, 0x0092f902L, 0x00e5ba68L, 0x005e1bddL,
        0x00844704L, 0x003fe6b1L, 0x0048a5dbL, 0x00f3046eL,
        0x00a6230fL, 0x001d82baL, 0x006ac1d0L, 0x00d16065L,
        0x00c08f12L, 0x007b2ea7L, 0x000c6dcdL, 0x00b7cc78L,
        0x00e2eb19L, 0x00594aacL, 0x002e09c6L, 0x0095a873L,
        0x000dd728L, 0x00b6769dL, 0x00c135f7L, 0x007a9442L,
        0x002fb323L, 0x00941296L, 0x00e351fcL, 0x0058f049L,
        0x00491f3eL, 0x00f2be8bL, 0x0085fde1L, 0x003e5c54L,
        0x006b7b35L, 0x00d0da80L, 0x00a799eaL, 0x001c385fL,
        0x002cc6e9L, 0x0097675cL, 0x00e02436L, 0x005b8583L,
        0x000ea2e2L, 0x00b50357L, 0x00c2403dL, 0x0079e188L,
        0x00680effL, 0x00d3af4aL, 0x00a4ec20L, 0x001f4d95L,
        0x004a6af4L, 0x00f1cb41L, 0x0086882bL, 0x003d299eL,
        0x00a556c5L, 0x001ef770L, 0x0069b41aL, 0x00d215afL,
        0x008732ceL, 0x003c937bL, 0x004bd011L, 0x00f071a4L,
        0x00e19ed3L, 0x005a3f66L, 0x002d7c0cL, 0x0096ddb9L,
        0x00c3fad8L, 0x00785b6dL, 0x000f1807L, 0x00b4b9b2L
};

/* Calculates the checksum of a buffer. */
/* tempCRC: initial value of the checksum (CRC register). */
/* *outPtr: initially points to the first character in the buffer/message/string. */
/* bufSize: number of characters in the buffer. */
unsigned short updateCRC16(unsigned short tempCRC, char *outPtr, size_t bufSize)
{
/* A register declaration advises the compiler that the variable will be heavily */
/* used. The idea is that register variables are to be placed in machine */
/* registers, which may result in smaller and faster programs. */
    register unsigned short crc = tempCRC;
    register char *cp = outPtr;
    register unsigned int count = bufSize;

    while(count--)
        crc = (crc<<8) ^ crc16Table[(crc>>8) ^ *cp++];
        return(crc);
}

/* Calculates the checksum of a buffer. */
/* tempCRC: initial value of the checksum (CRC register). */
/* *bufPtr: initially points to the first character in the buffer/message/string. */
/* bufSize: number of characters in the buffer. */
unsigned long updateCRC24(unsigned long tempCRC, char *outPtr, size_t bufSize)
{
    register unsigned long crc = tempCRC;
    register char *cp = outPtr;
    register unsigned int count = bufSize;
        int i;

        i=0;
    while(count--)
        {
            i = ( (int)(crc>>16) ^ *cp++ ) & 0xff;
            crc = (crc<<8) ^ crc24Table[i];
        }
        return crc & 0xffffffL;
}
#endif /* SAFE__H */
```

8.1.2 Test of CRC implementations

8.1.2.1 24-bit CRC test

CHAR	HEX	BIN
T	54	01010100
H	48	01001000
E	45	01000101
,	2C	00101100
Q	51	01010001
U	55	01010101
I	49	01001001
C	43	01000011
K	4B	01001011
,	2C	00101100
B	42	01000010
R	52	01010010
O	4F	01001111
W	57	01010111
N	4E	01001110

char	hex	bin
t	74	01110100
h	68	01101000
e	65	01100101
,	2C	00101100
q	71	01110001
u	75	01110101
i	69	01101001
c	63	01100011
k	6B	01101011
,	2C	00101100
b	62	01100010
r	72	01110010
o	6F	01101111
w	77	01110111
n	6E	01101110

,	2C	00101100
F	46	01000110
O	4F	01001111
X	58	01011000
,	2C	00101100
0	30	00110000
1	31	00110001
2	32	00110010
3	33	00110011
4	34	00110100
5	35	00110101

,	2C	00101100
f	66	01100110
o	6F	01101111
x	78	01111000
,	2C	00101100
0	30	00110000
1	31	00110001
2	32	00110010
3	33	00110011
4	34	00110100
5	35	00110101

Table 8-1, The test strings for the 24-bit CRC polynomial.

Bit string “UPPER CASE”: (26 chars => 208 bits):
0101010001001000010001010010110001010001010101010100100101000011010010110
0010110001000010010100100100111101010111010011100010110001000110010011110
0101100000101100001100000011000100110010001100110011010000110101

Bit string “lower case”: (26 chars => 208 bits):
0111010001101000011001010010110001110001011101010110100101100011011010110
0010110001100010011100100110111101110111011011100010110001100110011011110
0111100000101100001100000011000100110010001100110011010000110101

HEX	BIN
21	00100001
75	01110101
3C	00111100

Hex	bin
5F	1011111
24	00100100
3A	00111010

Table 8-2, The checksums generated for the test string with the 24-bit CRC polynomial.

Checksum UPPER CASE:	0x21753C = 001000010111010100111100
Checksum lower case:	0x5F243A = 010111110010010000111010

Clipping from the Mathematica-test of the CRC-24 with the lower case bit string:
Comment: *s is a binary representation of the generator polynomial.*
s2 is a binary representation of the bit string with the appended checksum.
g24 is the generator polynomial.
f is the polynomial representation of the bit string with the checksum.
Out[7] prints the generator polynomial and Out[8] shows the result of the polynomial division.

```
In[7]:= s = {1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1};
        s2 =
        {0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1,
        0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0,
        0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0,
        1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1,
        0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1,
        0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0,
        1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0,
        0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1,
        0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0};
        g24 = 0;
        Do[g24 = g24 + s [[26 – i]] *x^ (i - 1), {i, 25}];
        g24
        f = 0;
        Do[f = f + s2[[233 – i]] *x^ (i - 1), {i, 232}];
        PolynomialMod[PolynomialReminder[f, g24, x], 2]
```

Out[7]= $1 + x^2 + x^4 + x^5 + x^7 + x^8 + x^{13} + x^{15} + x^{16} + x^{17} + x^{19} + x^{20} + x^{21} + x^{23} + x^{24}$

Out[8]= 0

8.1.2.2 16-bit CRC test

Char	HEX	BIN
J	4A	01001010
A	41	01000001
N	4E	01001110

char	Hex	Bin
j	6A	01101010
a	61	01100001
n	6E	01101110

Table 8-3, The test strings for the 16-bit CRC polynomial.

Bit string “UPPER CASE”: 010010100100000101001110
Checksum UPPER CASE: 0x1303 = 0001001100000011

Bit string “lower case”: 011010100110000101101110
Checksum lower case: 0xc5aa = 1100010110101010

Clipping from the Mathematica-test of the CRC-16 with the upper case bit string:
Comment: *s is a binary representation of the 12 bit BCH-code.*
g12 is the 12 bit generator polynomial.
g16 is the 16 bit generator polynomial(expansion of the 12 bit BCH-code).
f is the polynomial representation of the bit string with the checksum.
Out[27] shows the polynomial representation the bit string and the checksum. Out[28] shows the result of the polynomial division.

```
In[22]:= s = {1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1};
        g12 = 0;
        Do[g12 = g12 + s [[14 – i]] *x^ (i - 1), {i, 13}];
        g12
        Factor[g12, Modulus → 2]
        g16 = Expand[g12 (1 + x) (1 + x + x³), Modulus → 2]
        Factor[g16, Modulus → 2]
```

Out[22]= $1 + x^2 + x^4 + x^7 + x^8 + x^9 + x^{12}$

Out[23]= $(1 + x^5 + x^6) (x^1 + x^2 + x^4 + x^5 + x^6)$

Out[24]= $1 + x^3 + x^4 + x^5 + x^{11} + x^{12} + x^{13} + x^{14} + x^{15} + x^{16}$

Out[25]= $(1 + x) (1 + x + x3) (1 + x^5 + x^6) (x^1 + x^2 + x^4 + x^5 + x^6)$

```
In[26]:= s = {0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0,
0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1};
        f = 0;
        Do[f = f + s2[[233 – i]] *x^ (i - 1), {i, 232}];
        PolynomialMod[PolynomialReminder[f, g24, x], 2]
```

Out[26]= $1 + x + x^8 + x^9 + x^{12} + x^{17} + x^{18} + x^{19} + x^{22} + x^{24} + x^{30} + x^{33} + x^{35} + x^{38}$

Out[27]= 0

8.2 Safe function blocks

I have chosen to just include parts of the code constituting the two blocks I implemented. All code that is related to the safety layer has been included.

8.2.1 SAI-1 block

```
/*      Sai.C -
 *
 *      Sample of a "safe" application program built on top of FBSh
 *      version 2.15
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <fbsh.h>
#include <hart.h>
#include "safe.h"

/*
SAFECYCLE = 6 * loop time
Loop time = 2 sec
1 sec = 32000 periods
Loop time = 2 * 32000 = 64000 periods
SAFECYCLE = 64000 * 6 = 384000
*/
#define SAFECYCLE 384000
#define MAXJITTER 19200
/*      sigma for each cycle is assumed to be 100ms
        there is six fields in a frame, thus worst case
        delay is 6 * 100ms = 600ms.
        MAXJITTER should be 600ms =>
        MAXJITTER = 60% of one second
        Thus MAXJITTER = (32000 * 0.6) = 19200
*/
#define SIMULATE_CHANNEL      1
#define SENSOR_CHANNEL        2
#define HI_HI_ALM_IDX         33

typedef struct sFF_Time_t
{
        int32 upper;
        int32 lower;
}sFF_Time_t;

typedef struct wDog_t
{
        FF_Time now;
        FF_Time FBtime; /* sampled at each FB execution */
} wDog_t;

typedef struct safetyParams_t
{
        uint16          fieldNo;        /* 1 */
        union{
                uint16 num;
                unsigned char c[2];
        }sender;                        /* 2 */
        union{
                uint16 num;
                unsigned char c[2];
        }dest;                          /* 3 */
        union{
                uint32 num;
                unsigned char c[4];
        }frameCounter;                  /* 4 */
        union{
                float f;
                unsigned char c[4];
        }floatTemp;                     /* 5 */
        union{
                uint16 num;
                unsigned char c[2];
        }shortTemp;                     /* 6 */
        float           errorFlag;      /* 7 */
        FF_Time         newTime;        /* 8 */
        FF_Time         prevTime;       /* 9 */
} safetyParams_t;

typedef struct resBlock_t
{
        /* Resource block parameters */
        /* I have omitted these parameters in the appendix to save space */
        /* as they do noy have anything to do with the safety layer. */
} resBlock_t;

typedef struct transBlock_t
{
        /* Transducer block parameters. */
        /* I have omitted these parameters in the appendix to save space */
        /* as they do noy have anything to do with the safety layer. */
} transBlock_t;

typedef struct aiBlock_t
{
        uint16                  st_rev;
        FF_OctetStr             tag_desc[32];
        uint16                  strategy;
        uint8                   alert_key;
        FF_Mode                 mode_blk;
        uint16                  block_err;
        FF_VsFloat              pv;
        FF_VsFloat              out;
        … (some parameters have been omitted)…
        /* Safety additions */
        safetyParams_t sParams;
        FF_VsFloat              fVal;
        FF_VsFloat              sIn; /* 39 - Receives ACK */
        FF_VsFloat              ack; /* 40 - Holds acknowledgement */
        char                    CRCarg[25]; /* 41 */
} aiBlock_t;

resBlock_t              resBlock;
transBlock_t            transBlock;
aiBlock_t               aiBlock;
wDog_t                  wDog;
bool_t                  firstTime;

extern nihDesc_t hartCMDDesc[3];

HDL_VFD         hVfd = 1;       /* VFD handle, it is 1 since there is only one VFD */

/* block handles */
HDL_BLOCK       hResBlock = 1, hTransBlock = 2, hAiBlock = 3;

uint8 SystemClockSpeed = 1;     /* Choose clock speed of 4 MHz */
uint8 RamSize = 2;              /* Choose RAM size of 256K bytes */

void initialize_static_param();
void initialize_nv_param();
void initialize_dynamic_param();
void initApp();
void watchdog();

/* Help function prototypes: */
void simulateInput(HDL_BLOCK hBlock);
void insertInCRCarg(HDL_BLOCK hBlock, int pos);
void generateACK(HDL_BLOCK hBlock);
```

```
uint32 calculateCRC(HDL_BLOCK hBlock, int bits);
bool_t checkACK(HDL_BLOCK hBlock);
void initCRCarg(HDL_BLOCK hBlock);
void fatalError(float error);
int onTime(FF_Time *n, FF_Time *p);
FF_TimeComp(FF_Time *ne, FF_Time *pr);
struct sFF_Time_t FF_TimeSubtraction(FF_Time *a, FF_Time *b);

/*
* The code for the callback functions cbRead(), cbWrite(), cbNotifyRead(),
*cbNotifyWrite() have I not included here, as they are of no significance for the safety
*layer.
*/

/*
 * Callback function for function block execution. Called by the fbsh when a
 * function block is scheduled to run.
 */
void cbExec(HDL_BLOCK hBlock)
{
       FF_Time        curTime;
       int16 r, stat;
       bool_t   stale;
       char    data[20];

       /* AI-BLOCK. */
       if (hBlock == hAiBlock)
       {
              shWaitBlockSem(hVfd, hAiBlock); /* Aquire the semaphore for the aiBlock */
              wDog.FBtime = shGetTime(); /* sample time for watchdof function */

              if (aiBlock.channel == SENSOR_CHANNEL)
              {
                     /* Standarized AI Block things */
              }
       else
/* SIMULATE_CHANNEL – I have just implemented the safety function for the simulate mode
of this block. */
              {
                     if (aiBlock.sParams.errorFlag == 0)
                     {
                            /* fieldNo indicates what to put in the out-parameter. */
                            switch (aiBlock.sParams.fieldNo)
                            {
                            case 0:
                                   if (aiBlock.sParams.frameCounter.num == 0)
                                   { /* No ack to be checked this time, but */
                                   /* generate a copy of expected ack for frame 0. */
                                          initCRCarg(hBlock);
                                          generateACK(hBlock);
                                          initCRCarg(hBlock);
                                          /* Simulate an input value. */
                                          simulateInput(hBlock);
                                          /* Put data into CRCarg for later CRC
                                          calculation */
                                          insertInCRCarg(hBlock, 0);
                                          /* Capture time when first field of frame is
                                          put in out-buffer */
                                          aiBlock.sParams.prevTime = shGetTime();
                                          aiBlock.sParams.fieldNo = 1;
                                          break;
                                   }
                                   else if (aiBlock.sParams.frameCounter.num == 1)
                                   {
                                          /* Do not check ack. Simulate an input
                                          value. */
                                          simulateInput(hBlock);
                                          /* Put data into CRCarg for later CRC
                                          calculation */
                                          insertInCRCarg(hBlock, 0);
                                          aiBlock.sParams.newTime = shGetTime();
```

```
                /* Capture time when first field of frame is
                put in out buffer */
                aiBlock.sParams.fieldNo = 1;
                break;
        }
        else
        {
        /* Check if a correct acknowledgement have been
        received. */
        /* Generate next expected ack before frameCounter */
        /* is incremented and CRCarg is filled up */
        /* gradually with safety related data. */
                initCRCarg(hBlock);
                generateACK(hBlock);
                initCRCarg(hBlock);
                if (checkACK(hBlock))
                {
                        /* Simulate an input value. */
                        simulateInput(hBlock);
                        /* Put data into CRCarg for later CRC
                        calculation */
                        insertInCRCarg(hBlock, 0);
                        /* save previous frame's timestamp in
                        prevTime */
                        aiBlock.sParams.prevTime =
                        aiBlock.sParams.newTime;
                        aiBlock.sParams.newTime =
                        shGetTime();
                        aiBlock.sParams.fieldNo = 1;
                        break;
                }
                else
                {
                        /*fatalError(0.5);*/
                        break;
                }
        }

case 1: /* "Safe addresses */
        aiBlock.sParams.floatTemp.c[0] =
        aiBlock.sParams.sender.c[0];
        aiBlock.sParams.floatTemp.c[1] =
        aiBlock.sParams.sender.c[1];
        aiBlock.sParams.floatTemp.c[2] =
        aiBlock.sParams.dest.c[0];
        aiBlock.sParams.floatTemp.c[3] =
        aiBlock.sParams.dest.c[1];
        aiBlock.out.f = aiBlock.sParams.floatTemp.f;
        aiBlock.out.status = ADDRBIT;
        insertInCRCarg(hBlock, 5);
        aiBlock.sParams.fieldNo = 2;
        break;
case 2: /* frameCounter */
        /* increment fC. */
        aiBlock.sParams.frameCounter.num++;
        aiBlock.out.f =
        (float)aiBlock.sParams.frameCounter.num;
        aiBlock.out.status = FCBIT;
        insertInCRCarg(hBlock, 10);
        aiBlock.sParams.fieldNo = 3;
        break;
case 3: /* lower part of time stamp */
        if (aiBlock.sParams.frameCounter.num == 1)
        {
                aiBlock.out.f =
                (float)aiBlock.sParams.prevTime.lower;
                aiBlock.out.status = LOWERBIT;
                insertInCRCarg(hBlock, 15);
                aiBlock.sParams.fieldNo = 4;
                break;
        }
```

```
                                        else
                                        {
                                                aiBlock.out.f =
                                                (float)aiBlock.sParams.newTime.lower;
                                                aiBlock.out.status = LOWERBIT;
                                                insertInCRCarg(hBlock, 15);
                                                aiBlock.sParams.fieldNo = 4;
                                                break;
                                        }

                                case 4: /* upper part of time stamp */
                                        if (aiBlock.sParams.frameCounter.num == 1)
                                        {
                                                aiBlock.out.f =
                                                (float)aiBlock.sParams.prevTime.upper;
                                                aiBlock.out.status = UPPERBIT;
                                                insertInCRCarg(hBlock, 20);
                                                aiBlock.sParams.fieldNo = 5;
                                                break;
                                        }
                                        else
                                        {
                                                aiBlock.out.f =
                                                (float)aiBlock.sParams.newTime.upper;
                                                aiBlock.out.status = UPPERBIT;
                                                insertInCRCarg(hBlock, 20);
                                                aiBlock.sParams.fieldNo = 5;
                                                break;
                                        }

                                case 5:
                                        aiBlock.out.f = (float)calculateCRC(hBlock, 24);
                                        initCRCarg(hBlock);
                                        aiBlock.sParams.fieldNo = 0;
                                        break;

                                default:
                                        aiBlock.sParams.fieldNo = 0;
                                        fatalError(6.1);
                                        break;
                                } /* end of switch */
                        }/* end if no detected errors */
                } /* end SIMULATE_CHANNEL */
                /* Release the semaphore of the function block */
                shSignalBlockSem(hVfd, hAiBlock);
        }
} /* end of cdExec() */

/* Simulates a process value. */
void simulateInput(HDL_BLOCK hBlock)
{
        aiBlock.out = aiBlock.fVal;
        aiBlock.out.f += 1.2;
        if (aiBlock.out.f > 22.2)
                aiBlock.out.f = 0.0;
        aiBlock.out.status = 0x80;

        /* Save field vale and its status for next sampling. */
        aiBlock.fVal = aiBlock.out;
        return;
}

/* This function is used to paste field-data into the array storing the */
/* argument for the CRC calculation. The pos-variable identifies the offset */
/* at which the first character of the field is to be inserted. */
void insertInCRCarg(HDL_BLOCK hBlock, int pos)
{
        int i = 0;

        i = pos;
        aiBlock.sParams.floatTemp.f = 0;
        aiBlock.sParams.floatTemp.f = aiBlock.out.f;
        aiBlock.CRCarg[i]   = aiBlock.sParams.floatTemp.c[0];
        aiBlock.CRCarg[++i] = aiBlock.sParams.floatTemp.c[1];
        aiBlock.CRCarg[++i] = aiBlock.sParams.floatTemp.c[2];
        aiBlock.CRCarg[++i] = aiBlock.sParams.floatTemp.c[3];
        aiBlock.CRCarg[++i] = aiBlock.out.status;
}

uint32 calculateCRC(HDL_BLOCK hBlock, int bits)
{
        uint16 checksum16;
        uint32 checksum24;

        switch(bits)
        {
        case 16:
                checksum16 = 0x0000;
                checksum16 = updateCRC16(checksum16, aiBlock.CRCarg, 3);
                return (uint32)checksum16;

        case 24:
                checksum24 = 0x00000000L;
                checksum24 = updateCRC24(checksum24, aiBlock.CRCarg, 25);
                aiBlock.out.status = CRC24BIT;
                return checksum24;

        default:
                return 0;
        }
}

/* Generates a local ack-message. This message is compared with the ack from the
destination. IMPRTANT NOTE: CRCarg must be initiated before calling this function. */
void generateACK(HDL_BLOCK hBlock)
{
        /* Get the two LSB from frameCounter and insert into CRCarg. */
        aiBlock.CRCarg[0] = aiBlock.sParams.frameCounter.c[2];
        aiBlock.CRCarg[1] = aiBlock.sParams.frameCounter.c[3];
        aiBlock.CRCarg[2] = ACKBIT;   /* Insert ack ID byte into CRCarg */
        /* Calculate CRC-16 for ack-message */
        aiBlock.sParams.shortTemp.num = (uint16)calculateCRC(hBlock, 16);
        /* Insert two LSB from frameCounter into temp */
        aiBlock.sParams.floatTemp.c[0] = aiBlock.sParams.frameCounter.c[2];
        aiBlock.sParams.floatTemp.c[1] = aiBlock.sParams.frameCounter.c[3];
        /* Insert CRC into temp */
        aiBlock.sParams.floatTemp.c[2] = aiBlock.sParams.shortTemp.c[0];
        aiBlock.sParams.floatTemp.c[3] = aiBlock.sParams.shortTemp.c[1];
        /* aiBlock.ack.f = floatTemp.f; */
        aiBlock.ack.f = aiBlock.sParams.floatTemp.f;
        /* Insert ack ID byte into sOut.status */
        aiBlock.ack.status = ACKBIT;
        return;
}

/* Compares the locally generated ack with received ack. */
bool_t checkACK(HDL_BLOCK hBlock)
{
        if (aiBlock.sIn.status == ACKBIT)
        {
                if (onTime(&aiBlock.sParams.newTime, &aiBlock.sParams.prevTime))
                {
                        if (aiBlock.ack.f == aiBlock.sIn.f)
                        {
                                return TRUE;
                        }
                        else
                        {
                                fatalError(0.2);
                                return FALSE;
                        }
                }
```

```
               else
               {
                       fatalError(0.3);
                       return FALSE;
               }
       }
       else
       {
               fatalError(0.1);
               return FALSE;
       }
}

/* Initialises the CRC array. */
void initCRCarg(HDL_BLOCK hBlock)
{
       int i;
       for (i=0; i<25; i++)
       {
               aiBlock.CRCarg[i] = 0;
       }
       return;
}

void fatalError(float error)
{
       aiBlock.sParams.errorFlag = error;
       return;
}

/* Checks if the function block executes regularly. */
int onTime(FF_Time *n, FF_Time *p)
{
       int res;
       res = FF_TimeComp(n, p);
       if (res <= 0) /* n==p or n<p */
               return 0;
       else if (res == 1) /* n>p */
       {
               struct sFF_Time_t difference;
               difference = FF_TimeSubtraction(n, p);
               if (insideTimeWindow(&difference))
               {
                       return 1;
               }
               else
                       return 0;
       }
       else
               return 0;
}

/* Compares two timestamps of type FF_Time. */
/* returns 0 if equal, 1 if ne>pr, -1 if ne<pr */
int FF_TimeComp(FF_Time *ne, FF_Time *pr)
{
       int retv;
       retv = -1;
       if (ne->upper > pr->upper)
               retv = 1;
       else if (ne->upper == pr->upper)
       {
               if (ne->lower > pr->lower)
                       retv = 1;
               else if (ne->lower == pr->lower)
                       retv = 0;
       }
       return retv;
}
```

```
/* Subtracts upper part of timestamp b from upper part of timestamp b. */
struct sFF_Time_t FF_TimeSubtraction(FF_Time *a, FF_Time *b)
{
	struct sFF_Time_t diff;

	diff.upper = a->upper - b->upper;
	diff.lower = a->lower - b->lower;
	return diff;
}

/* Checks that the difference between newTime and prevTime is less than MAXJITTER. */
int insideTimeWindow(sFF_Time_t *d)
{
	if (d->upper == 0)
	{
		if ((abs(d->lower) - SAFECYCLE)  <= MAXJITTER) /* inside time window */
			return 1;
		else
			return 0;
	}
	else
	{
		return 0;
		fatalError(0.4);
	}
}

/* This is the starting point of the application. This function is called once after the
kernel boots up. Inside this function the callback functions are registered and the
function block shell is initialised. Application-specific initialisations are also
performed here. */
void userStart()
{
	bool_t		status;
	RETCODE		retcode;
	PARAM_PTR	param[41];
	int			i;

	retcode = shRegisCallback(1, cbRead, cbWrite, cbNotifyRead, cbNotifyWrite,
					cbExec, cbAckAlertNotify, cbAlarmAck, NULL);
	if (retcode)
		fatalError(0);

	/* initialize hart interface */
	hart_init_sequence();
	for (i = 0; i < 41; i++)
		param[i].offset = i + 1;
	…
	param[0].ptr = &aiBlock.st_rev;
	param[1].ptr = &aiBlock.tag_desc;
	param[2].ptr = &aiBlock.strategy;
	param[3].ptr = &aiBlock.alert_key;
	param[4].ptr = &aiBlock.mode_blk;
	… (some parameters have been omitted)…
	param[36].ptr = &aiBlock.sParams;
	param[37].ptr = &aiBlock.fVal;
	param[38].ptr = &aiBlock.sIn;
	param[39].ptr = &aiBlock.ack;
	param[40].ptr = &aiBlock.CRCarg;
	retcode = shRegisParamPtr(1, hAiBlock, 41, param);
	if (retcode != R_SUCCESS)
		fatalError(0);

	retcode = shInitShell(&firstTime);
	if (retcode != R_SUCCESS) {
		fatalError(0);
	}

	/* users do their own initializations here */
	initApp();
	retcode = shStartExecLoop(watchdog, 2000);
```

```
/* watchdog() runs in 2000ms intervals (the length of one macrocycle). */
}

/* Application-specific initialisations are performed in this function. */
void initApp()
{
        if (firstTime) {
        /* for STATIC parameter, only initialize them when firstTime is TRUE. */
                initialize_static_param();
                initialize_nv_param();
        }

        /* since NVM storage of parameters with non-volatile storage types
         * are not yet implemented in the Funcion Block Shell, these parameters
         * still needs to be initialized everytime the program starts
         */

        /* always initialize the dynamic parameter */
        initialize_dynamic_param();
}

#define XD_SCALE_OFFSET       10
#define MODE_OFFSET           5
#define HI_HI_LIM_OFFSET      26
#define HI_HI_PRI_OFFSET      25
#define MANUFAC_ID_OFFSET     10
#define DEV_TYPE_OFFSET       11
#define DEV_REV_OFFSET        12
#define DD_REV_OFFSET         13
#define CONFIRM_TIME_OFFSET   33

/* Initialisation of static parameters. */
void initialize_static_param()
{
        aiBlock.xd_scale.EU100 = 100;
        aiBlock.xd_scale.EU0 = 10;
        aiBlock.xd_scale.unitIndex = 3;
        aiBlock.xd_scale.decPoint = 2;
         shWriteNVM(hVfd, hAiBlock, XD_SCALE_OFFSET, &aiBlock.xd_scale);


        aiBlock.mode_blk.target = 0x08;
        aiBlock.mode_blk.permitted = 0x0c;
        aiBlock.mode_blk.normal = 0x08;
         shWriteNVM(hVfd, hAiBlock, MODE_OFFSET, &aiBlock.mode_blk);

        aiBlock.hi_hi_lim = 500;
        aiBlock.hi_hi_pri = 4;
        aiBlock.channel = SIMULATE_CHANNEL;
         shWriteNVM(hVfd, hAiBlock, HI_HI_LIM_OFFSET, &aiBlock.hi_hi_lim);
         shWriteNVM(hVfd, hAiBlock, HI_HI_PRI_OFFSET, &aiBlock.hi_hi_pri);

        resBlock.manufac_id = 0x1234;
        resBlock.dev_type = 1;
        resBlock.dev_rev = 1;
        resBlock.dd_rev = 1;
        resBlock.confirm_time = 32000;
        shWriteNVM(hVfd, hResBlock, MANUFAC_ID_OFFSET, &resBlock.manufac_id);
                shWriteNVM(hVfd, hResBlock, DEV_TYPE_OFFSET, &resBlock.dev_type);
                shWriteNVM(hVfd, hResBlock, DEV_REV_OFFSET, &resBlock.dev_rev);
                shWriteNVM(hVfd, hResBlock, DD_REV_OFFSET, &resBlock.dd_rev);
                shWriteNVM(hVfd, hResBlock, CONFIRM_TIME_OFFSET, &resBlock.confirm_time);
}

/* Initialisation of dynamic parameters. */
void initialize_dynamic_param()
{
        int i;
        aiBlock.sParams.fieldNo = 0;
        aiBlock.sParams.sender.num = 1111; /* This device */
        aiBlock.sParams.dest.num = 2222; /* Receivers address */
        aiBlock.sParams.frameCounter.num = 0;
        aiBlock.sParams.floatTemp.f = 0;
        aiBlock.sParams.shortTemp.num = 0;
        aiBlock.sParams.errorFlag = 0;

        aiBlock.fVal.f = 0;
        aiBlock.fVal.status = 0;
        aiBlock.sIn.f = 0;
        aiBlock.sIn.status = 0;
        aiBlock.ack.f = 0;
        aiBlock.ack.status = 0;

        for (i=0; i<32; i++)
        {
                aiBlock.CRCarg[i] = 0;
        }
}

/* This function checks whether the device's SM triggers the function      */
/* blocks accoring to the configuration. This is done by comparing         */
/* wDog.FBtime and wDog.now. If the difference between FBtime (sampled     */
/* each time the function block executes) and now is bigger than an        */
/* accepted jitter (SAFECYCLE), something is wrong.                        */
/* THIS FUNCTION DO NOT RUN AS A PART OF THE FUNCTION BLOCK! */
void watchdog()
{
        uint32 dif;
        struct sFF_Time_t differ;

        wDog.now = shGetTime();
        differ = FF_TimeSubtraction(&wDog.now, &wDog.FBtime);
        if (differ.upper == 0)
        {
                dif = abs(differ.lower);
                /* Application dependent acceptable jitter. Here: 2 seconds */
                if (dif <= 64000)
                        return;
                else
                {
                        fatalError(100.1);
                        return;
                }
        }
        else
        {
                fatalError(100.2);
                return;
        }
}
```

8.2.2 SAO-1 block

```
/*      Sao.C -
 *
 *      Sample of a "safe" application program built on top of FBSh
 *      version 2.15
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <fbsh.h>
#include <hart.h>
#include "safe.h"

/*
SAFECYCLE = 6 * loop time
Loop time = 2 sec
1 sec = 32000 periods
```

```
Loop time = 2 * 32000 = 64000 periods
SAFECYCLE = 64000 * 6 = 384000
*/
#define SAFECYCLE 384000
#define MAXJITTER 19200
/*      sigma for each cycle is assumed to be 100ms
        there is six fields in a frame, thus worst case
        delay is 6 * 100ms = 600ms.
        MAXJITTER should be 600ms =>
        MAXJITTER = 60% of one second
        Thus MAXJITTER = (32000 * 0.6) = 19200
*/

/*
* Type definitions of sFF_Time, wDog_t and safetyParams_t are identical as in SAI-1.
*/

typedef struct resBlock_t
{
        /* Resource block parameters */
        /* I have omitted these parameters in the appendix to save space */
        /* as they do noy have anything to do with the safety layer. */
} resBlock_t;

typedef struct aoBlock_t {
        uint16                  st_rev;
        FF_OctetStr             tag_desc[32];
        uint16                  strategy;
        uint8                   alert_key;
        FF_Mode                 mode_blk;
        FF_VsFloat              cas_in;
        … (some parameters have been omitted)…
        /* Safety additions: */
        safetyParams_t sParams;
        FF_VsFloat              fVal;
        FF_VsFloat              sOut; /* Sends ACK */
        FF_VsFloat              ack; /* Holds acknowledgement */
        char                    CRCarg[25]; /* 35 */
} aoBlock_t;

resBlock_t      resBlock;
aoBlock_t       aoBlock;
wDog_t          wDog;
bool_t          firstTime;

extern nihDesc_t hartCMDDesc[3];

HDL_VFD         hVfd = 1;       /* VFD handle, it is 1 since there is only one VFD */

/* block handles */
HDL_BLOCK               hResBlock = 1, hAoBlock = 2;

/* Some required global variables for Stack */
uint8 SystemClockSpeed = 1; /* Choose clock speed of 4 MHz */
uint8 RamSize = 2;      /* Choose RAM size of 256K bytes   */
void initialize_static_param();
void initialize_nv_param();
void initialize_dynamic_param();
void initApp();
void watchdog();

/* Help function prototypes: */
void insertInCRCarg(HDL_BLOCK hBlock, int pos);
uint32 calculateCRC(HDL_BLOCK hBlock, int bits);
void generateACK(HDL_BLOCK hBlock);
void initCRCarg(HDL_BLOCK hBlock);
void fatalError(float error);
int onTime(FF_Time *n, FF_Time *p);
FF_TimeComp(FF_Time *ne, FF_Time *pr);
struct sFF_Time_t FF_TimeSubtraction(FF_Time *a, FF_Time *b);
int insideTimeWindow(sFF_Time_t *d);
```

```
/*
* The code for the callback functions cbRead(), cbWrite(), cbNotifyRead(),
*cbNotifyWrite() have I not included here, as they are of no significance for the safety
*layer.
*/

/*
 * Callback function for function block execution. Called by the fbsh when a
 * function block is scheduled to run.
 */
void cbExec(HDL_BLOCK hBlock)
{
        /* AO-BLOCK. */
        if (hBlock == hAoBlock)
        {
                shWaitBlockSem(hVfd, hAoBlock);
                wDog.FBtime = shGetTime(); /* sample time for watchdof function */
                if (aoBlock.sParams.errorFlag == 0)
                {
                        /* fieldNo indicates what type of data that comes from the out-
                        parameter of the aiBlock. */
                        switch (aoBlock.sParams.fieldNo)                    {
                        case 0: /* field value */
                                aoBlock.fVal.f = aoBlock.cas_in.f;
                                insertInCRCarg(hBlock, 0);
                                aoBlock.sParams.fieldNo = 1;
                                break;

                        case 1: /* "Safe addresses */
                                insertInCRCarg(hBlock, 5);
                                /* Check received sender address */
                                aoBlock.sParams.shortTemp.c[0] =
                                aoBlock.sParams.floatTemp.c[0];
                                aoBlock.sParams.shortTemp.c[1] =
                                aoBlock.sParams.floatTemp.c[1];
                                if (aoBlock.sParams.shortTemp.num !=
                                aoBlock.sParams.sender.num) /* sender == 111 */
                                {
                                        fatalError(1.1);
                                        break;
                                }
                                /* Check received destination address */
                                aoBlock.sParams.shortTemp.c[0] =
                                aoBlock.sParams.floatTemp.c[2];
                                aoBlock.sParams.shortTemp.c[1] =
                                aoBlock.sParams.floatTemp.c[3];
                                if (aoBlock.sParams.shortTemp.num !=
                                aoBlock.sParams.dest.num) /* dest == 222 */
                                {
                                        fatalError(1.2);
                                        break;
                                }
                                aoBlock.sParams.fieldNo = 2;
                                break;

                        case 2: /* frameCounter */
                                insertInCRCarg(hBlock, 10);
                                /* Validate frameCounter */
                                if (aoBlock.sParams.frameCounter.num == 0 &&
                                (uint32)aoBlock.cas_in.f == 0)
                                {/* First frame, no special action required */
                                        aoBlock.sParams.fieldNo = 3;
                                        break;
                                }
                                else
                                {
                                        if ((uint32)aoBlock.cas_in.f !=
                                        (aoBlock.sParams.frameCounter.num+1)
                                                && aoBlock.cas_in.status != FCBIT)
                                        {
```

```
                        fatalError(2.1);
                        break;
                }
                else
                {
                        aoBlock.sParams.frameCounter.num =
                        (uint32)aoBlock.cas_in.f;
                        aoBlock.sParams.fieldNo = 3;
                        break;
                }
        }

case 3: /* lower part of timestamp */
        insertInCRCarg(hBlock, 15);
        /* No validation here, wait until upper has arrived. */
        /* Just save in newTime.lower and check the status byte. */
        if (aoBlock.cas_in.status != LOWERBIT)
        {
                fatalError(3.1);
                break;
        }
        else if (aoBlock.sParams.frameCounter.num == 0)
        {
                aoBlock.sParams.prevTime.lower =
                (uint32)aoBlock.cas_in.f;
                aoBlock.sParams.fieldNo = 4;
                break;
        }
        else
        {
                aoBlock.sParams.newTime.lower =
                (uint32)aoBlock.cas_in.f;
                aoBlock.sParams.fieldNo = 4;
                break;
        }

case 4: /* upper part of timestamp */
        insertInCRCarg(hBlock, 20);
        if (aoBlock.cas_in.status != UPPERBIT)
        {
                fatalError(4.1);
                break;
        }
        else
        {/* if this is the first frame arriving, then set prevTime
        = newTime */
                if (aoBlock.sParams.frameCounter.num == 0)
                {
                        aoBlock.sParams.prevTime.upper =
                        (uint32)aoBlock.cas_in.f;
                        aoBlock.sParams.fieldNo = 5;
                        break;
                }
                else if (aoBlock.sParams.frameCounter.num == 1)
                {/* Must do this to be able to determine
                the jitter. */
                        aoBlock.sParams.newTime.upper =
                        (uint32)aoBlock.cas_in.f;
                        aoBlock.sParams.prevTime =
                        aoBlock.sParams.newTime;
                        aoBlock.sParams.fieldNo = 5;
                        break;
                }
                else
                /* Check if newTime is correct in relation with
                prevTime */
                {
                        aoBlock.sParams.newTime.upper =
                        (uint32)aoBlock.cas_in.f;
```

```
                                        if(onTime(&aoBlock.sParams.newTime,
                                        &aoBlock.sParams.prevTime))/* is newTime >
                                        prevTime */
                                        {
                                                aoBlock.sParams.prevTime =
                                                aoBlock.sParams.newTime;
                                                aoBlock.sParams.fieldNo = 5;
                                                break;
                                        }
                                        else
                                        {
                                                fatalError(4.2);
                                                break;
                                        }
                                }
                        }

                case 5:
                        if (aoBlock.cas_in.status != CRC24BIT)
                        {
                                fatalError(5.1);
                                break;
                        }
                        else
                        /* No field errors detected so far. Check whole frame */
                        {
                                aoBlock.sParams.floatTemp.f =
                                (float)calculateCRC(hBlock, 24);
                                if (aoBlock.sParams.floatTemp.f == aoBlock.cas_in.f)
                                /* frame is received OK */
                                {
                                        initCRCarg(hBlock);
                                        generateACK(hBlock);
                                        initCRCarg(hBlock);
                                        aoBlock.sOut.f = aoBlock.ack.f;
                                        aoBlock.sOut.status = aoBlock.ack.status;
                                        aoBlock.sParams.fieldNo = 0;
                                        break;
                                }
                                else
                                {
                                        fatalError(5.2);
                                        break;
                                }
                        }

                default:
                        aoBlock.sParams.fieldNo = 0;
                        break;
                } /* end switch. */
        } /* end if no detected errors */
        shSignalBlockSem(hVfd, hAoBlock);
    }
} /* end of cdExec() */

/* This function is used to paste field-data into the array storing the */
/* argument for the CRC calculation. The pos-variable identifies the offset */
/* at which the first character of the field is to be inserted. */
void insertInCRCarg(HDL_BLOCK hBlock, int pos)
{
        /* Same as in SAI-1, just change aiBlock with aoBolck. */
}

uint32 calculateCRC(HDL_BLOCK hBlock, int bits)
{
        /* Initialises the CRC array, except that aoBlock.CRCarg is sent as argument in
        the updateCRC-functions. */
}
```

```
/* IMPRTANT NOTE: CRCarg must be initiated before calling this function. Therefore can an
ack only be generated before or after CRCarg is filled up with data */
void generateACK(HDL_BLOCK hBlock)
{
        /* Get the two LSB from frameCounter and insert into CRCarg. */
        aoBlock.CRCarg[0] = aoBlock.sParams.frameCounter.c[2];
        aoBlock.CRCarg[1] = aoBlock.sParams.frameCounter.c[3];
        aoBlock.CRCarg[2] = ACKBIT;   /* Insert ack ID byte into CRCarg */
        /* Calculate CRC-16 for ack-message */
        aoBlock.sParams.shortTemp.num = (uint16)calculateCRC(hBlock, 16);
        /* Insert two LSB from frameCounter into temp */
        aoBlock.sParams.floatTemp.c[0] = aoBlock.sParams.frameCounter.c[2];
        aoBlock.sParams.floatTemp.c[1] = aoBlock.sParams.frameCounter.c[3];
        /* Insert CRC into temp */
        aoBlock.sParams.floatTemp.c[2] = aoBlock.sParams.shortTemp.c[0];
        aoBlock.sParams.floatTemp.c[3] = aoBlock.sParams.shortTemp.c[1];
        /* ack.f = floatTemp.f; */
        aoBlock.ack.f = aoBlock.sParams.floatTemp.f;
        /* Insert ack ID byte into sOut.status */
        aoBlock.ack.status = ACKBIT;
        return;
}

/* Initialises the CRC array. */
void initCRCarg(HDL_BLOCK hBlock)
{
        int i;
        for (i=0; i<32; i++)
        {
                aoBlock.CRCarg[i] = 0;
        }
        return;
}

void fatalError(float error)
{
        aoBlock.sParams.errorFlag = error;
}

/* Checks if the function block executes regularly. */
int onTime(FF_Time *n, FF_Time *p)
{
        /* Identical as in SAI-1. */
}

/* Compares two timestamps of type FF_Time. */
/* returns 0 if equal, 1 if ne>pr, -1 if ne<pr */
int FF_TimeComp(FF_Time *ne, FF_Time *pr)
{
        /* Identical as in SAI-1. */
}

/* Subtracts upper part of timestamp b from upper part of timestamp b. */
struct sFF_Time_t FF_TimeSubtraction(FF_Time *a, FF_Time *b)
{
        /* Identical as in SAI-1. */
}

/* Checks that the difference between newTime and prevTime is less than MAXJITTER. */
int insideTimeWindow(sFF_Time_t *d)
{
        if (d->upper == 0)
        {
                /* Identical as in SAI-1. */
        }
        else
        {
                return 0;
                fatalError(4.3);
        }
}
```

```
void userStart()
{
        /* identical as userStart() in SAI-1 except: */

        param[0].ptr = &aoBlock.st_rev;
        param[1].ptr = &aoBlock.tag_desc;
        … (some parameters have been omitted)…
        param[30].ptr = &aoBlock.sParams;
        param[31].ptr = &aoBlock.fVal;
        param[32].ptr = &aoBlock.sOut;
        param[33].ptr = &aoBlock.ack;
        param[34].ptr = &aoBlock.CRCarg;
        retcode = shRegisParamPtr(1, hAoBlock, 35, param);
        if (retcode != R_SUCCESS)
                fatalError(0);

/* rest of userStart() is identical with SAI-1. */
}

void
initApp()
{
        /* Identical to initApp in SAI-1 */
}

#define XD_SCALE_OFFSET       10
#define MODE_OFFSET           5
#define HI_HI_LIM_OFFSET      26
#define HI_HI_PRI_OFFSET      25
#define CONFIRM_TIME_OFFSET   33
#define MANUFAC_ID_OFFSET     10
#define DEV_TYPE_OFFSET       11
#define DEV_REV_OFFSET        12
#define DD_REV_OFFSET         13

void initialize_static_param()
{
        resBlock.manufac_id = 0x1234;
        resBlock.dev_type = 2;
        resBlock.dev_rev = 1;
        resBlock.dd_rev = 1;
        resBlock.confirm_time = 32000;
        shWriteNVM(hVfd, hResBlock, MANUFAC_ID_OFFSET, &resBlock.manufac_id);
                shWriteNVM(hVfd, hResBlock, DEV_TYPE_OFFSET, &resBlock.dev_type);
                shWriteNVM(hVfd, hResBlock, DEV_REV_OFFSET, &resBlock.dev_rev);
                shWriteNVM(hVfd, hResBlock, DD_REV_OFFSET, &resBlock.dd_rev);
                shWriteNVM(hVfd, hResBlock, CONFIRM_TIME_OFFSET, &resBlock.confirm_time);
}

void initialize_dynamic_param()
{
        int i;
        aoBlock.sParams.fieldNo = 0;
        aoBlock.sParams.sender.num = 1111;
        aoBlock.sParams.dest.num = 2222;
        aoBlock.sParams.frameCounter.num = 0;
        aoBlock.sParams.floatTemp.f = 0;
        aoBlock.sParams.shortTemp.num = 0;
        aoBlock.sParams.errorFlag = 0;
        aoBlock.sParams.newTime.upper = 0;
        aoBlock.sParams.newTime.lower = 0;
        aoBlock.sParams.prevTime.upper = 0;
        aoBlock.sParams.prevTime.lower = 0;

        aoBlock.fVal.f = 0;
        aoBlock.fVal.status = 0;
        aoBlock.sOut.f = 0;
        aoBlock.sOut.status = 0;
        aoBlock.ack.f = 0;
        aoBlock.ack.status = 0;
        for (i=0; i<32; i++)
        {
                aoBlock.CRCarg[i] = 0;
        }
}

void watchdog()
{
        /* Identical to watchdog in SAI-1 */
}
```

8.3 Device Descriptions

DD for the device containing the SAI-1 block. The DD for the device containing SAO-1, except that the AO block gets an output-parameter (sOut) instead of an input-parameter (sIn).

```
/* safeai.ddl 3. jan 2001
** This is the DD of my safe AI field device.
** © Tobias (2000) */

MANUFACTURER 0x1234, DEVICE_TYPE 1, DEVICE_REVISION 1, DD_REVISION 1
#include "std_defs.h"
#include "com_tbls.h"
/*
** Get generic block characteristics parameters
*/
IMPORT MANUFACTURER __FF, DEVICE_TYPE __STD_PARM, DEVICE_REVISION __STD_PARM_rel_dev_rev,
DD_REVISION __STD_PARM_rel_dd_rev
{
        EVERYTHING ;
}
/***********************************************************
         Import standard Resource Block
 ***********************************************************/
IMPORT MANUFACTURER __FF, DEVICE_TYPE __RES_BLOCK, DEVICE_REVISION
__RES_BLOCK_rel_dev_rev, DD_REVISION __RES_BLOCK_rel_dd_rev
{
         EVERYTHING ;
}
/***********************************************************
         Import standard Analog Input Function Block
 ***********************************************************/
IMPORT MANUFACTURER __FF, DEVICE_TYPE __AI_BLOCK, DEVICE_REVISION __AI_BLOCK_rel_dev_rev,
DD_REVISION __AI_BLOCK_rel_dd_rev
{
    EVERYTHING ;
    REDEFINITIONS
    {
                BLOCK   __analog_input_block

                {
                        PARAMETERS
                        {
                                ADD SPARAMS,                    sParams ;
                                ADD FVAL,                       fVal ;
                                ADD SIN,                        sIn ;
                                ADD ACK,                        ack ;
                                ADD CRCARG,                     CRCarg ;
                        }
                }
        }
}
/*
**********************************************************************
sParams:
RECORD sParams
        VARIABLE fieldNo
        VARIABLE sender
        VARIABLE dest
        VARIABLE frameCounter
        VARIABLE floatTemp
        VARIABLE shortTemp
        VARIABLE errorFlag
        VARIABLE newTime
        VARIABLE prevTime
**********************************************************************
*/
VARIABLE fieldNo
{
        LABEL           "|en|fieldNo" ;
        HELP            "|en|fieldNo indicates which field that is sent or received what "
                        "kind of safety related data that is being sent or received. "
                        "0-proc data, 1-addr, 2-frameCounter, 3-tS.lower, 4-tS.upper, "
                        "5-CRC)" ;
        CLASS           CONTAINED ;
        TYPE            UNSIGNED_INTEGER (2) ;
        HANDLING        READ & WRITE ;
}
VARIABLE sender
{
        LABEL           "|en|sender" ;
        HELP            "|en|The logical safety address of this device. (The sender)" ;
        CLASS           CONTAINED ;
        TYPE            UNSIGNED_INTEGER (2) ;
        HANDLING        READ & WRITE ;
}
VARIABLE dest
{
        LABEL           "|en|dest" ;
        HELP            "|en|The logical safety address of the receiving device." ;
        CLASS           CONTAINED ;
        TYPE            UNSIGNED_INTEGER (2) ;
        HANDLING        READ & WRITE ;
}
VARIABLE frameCounter
{
        LABEL           "|en|frameCounter" ;
        HELP            "|en|FrameCounter counts the logical data frames sent." ;
        CLASS           CONTAINED ;
        TYPE            UNSIGNED_INTEGER (4) ;
        HANDLING        READ & WRITE ;
}
VARIABLE floatTemp
{
        LABEL           "|en|floatTemp" ;
        HELP            "|en|Variable used to hold temporary float values during "
                        "various calculations." ;
        CLASS        CONTAINED ;
        TYPE         FLOAT ;
        HANDLING     READ & WRITE ;
}
VARIABLE shortTemp
{
        LABEL           "|en|shortTemp" ;
        HELP            "|en|Variable used to hold temporary short values during various "
                        "calculations." ;
        CLASS           CONTAINED ;
        TYPE            UNSIGNED_INTEGER (2) ;
        HANDLING        READ & WRITE ;
}
VARIABLE errorFlag
{
        LABEL           "|en|errorFlag" ;
        HELP            "|en|A variable that indicated the type of a detected error." ;
        CLASS           CONTAINED ;
        TYPE            FLOAT ;
        HANDLING        READ & WRITE ;
}
VARIABLE newTime
{
        LABEL           "|en|newTime" ;
        HELP            "|en|newTime holds the time stamp for the newly arrived frame." ;
        CLASS           CONTAINED ;
        TYPE            TIME_VALUE ;
        HANDLING        READ & WRITE ;
}
VARIABLE prevTime
{
        LABEL           "|en|prevTime" ;
        HELP            "|en|prevTime holds the time stamp for the previous arrived "
                        "frame." ;
```

```
       CLASS          CONTAINED ;
       TYPE           TIME_VALUE ;
       HANDLING       READ & WRITE ;
}
RECORD sParams
{
       LABEL          "|en|sParams." ;
       HELP           "|en|Safety parameters data struct, contains logical addressing, "
                      "a safety frame counter, a temporary variable used during "
                      "variaous calculations, time stamps and a boolean variable that "
                      "is TRUE when the function Block is in running mode." ;
       MEMBERS
       {
              FIELDNO,               fieldNo ;
              SENDER,                sender ;
              DEST,                  dest ;
              FRAMECOUNTER,          frameCounter ;
              FLOATTEMP,             floatTemp ;
              SHORTTEMP,             shortTemp ;
              ERRORFLAG,             errorFlag ;
              NEWTIME,               newTime ;
              PREVTIME,              prevTime ;
       }
}
/*
**********************************************************************
fVal:
RECORD fVal
       VARIABLE contained_float
       VARIABLE contained_status
**********************************************************************
*/
VARIABLE contained_status
{
       LABEL          "|en|Contained status" ;
       HELP           "|en|The status byte of fVal" ;
       CLASS          CONTAINED ;
       TYPE           INTEGER(1) ;
       HANDLING       READ & WRITE ;
}
VARIABLE contained_float
{
       LABEL          "|en|Contained float" ;
       HELP           "|en|The fVal holds the previous simulated field Value" ;
       CLASS          CONTAINED ;
       TYPE           FLOAT ;
       HANDLING       READ & WRITE ;
}
RECORD fVal
{
       LABEL          "|en|fVal" ;
       HELP           "|en|Field Value is just a variable used to store the simulated "
                      "process data between each iteration to achieve the "
                      "incrementation of process data." ;
       MEMBERS
       {
              CONTAINED_STATUS, contained_status ;
              CONTAINED_FLOAT,  contained_float ;
       }
}
/*
**********************************************************************
sIn:
RECORD sIn
       VARIABLE input_float
       VARIABLE input_status
**********************************************************************
*/
VARIABLE input_status
{
       LABEL          "|en|Input status" ;
       HELP           "|en|The in parameters status byte" ;
       CLASS          INPUT ;
       TYPE           INTEGER (1) ;
       HANDLING       READ & WRITE ;
}
```

```
VARIABLE input_float
{
       LABEL          "|en|Input float" ;
       HELP           "|en|The input parameters float variable" ;
       CLASS          INPUT ;
       TYPE           FLOAT ;
       HANDLING       READ & WRITE ;
}
RECORD sIn
{
       LABEL "|en|sIn" ;
       HELP "|en|Input parameter in sAI block." ;
       MEMBERS
       {
              INPUT_STATUS, input_status ;
              INTPUT_FLOAT, input_float ;
       }
}
/*
**********************************************************************
ack:
RECORD ack
       VARIABLE input_float
       VARIABLE input_status
**********************************************************************
*/
RECORD ack
{
       LABEL          "|en|ack" ;
       HELP           "|en|ack is a parameter used to hold an acknowledgement "
                      " message consisting of an ack identifier (ACKBIT), LSB of "
                      "the frameCounter and a 16 bit CRC. " ;
       MEMBERS
       {
              CONTAINED_STATUS, contained_status ;
              CONTAINED_FLOAT,  contained_float ;
       }
}
/*
**********************************************************************
CRCarg:
ARRAY  CRCarg
       VARIABLE CRCarg_entry
**********************************************************************
*/
VARIABLE CRCarg_entry
{
     LABEL            "|en|CRCarg_entry" ;
     HELP             "|en|A character in the CRC argument array" ;
     CLASS            CONTAINED ;
     TYPE             INTEGER (1) ;
     CONSTANT_UNIT    [blank] ;
     HANDLING         READ & WRITE ;
}
ARRAY  CRCarg
{
     LABEL            "|en|CRCarg" ;
     HELP             "|en|The CRCarg array holds the argument for the CRC "
                      "calculations." ;
     TYPE     CRCarg_entry ;
     NUMBER_OF_ELEMENTS 25 ;
}
```

8.4 Device Template & Device Configuration

8.4.1 Device Configuration

8.4.1.1 Safe AI

```
; Config file to be used in a basic device at address 0x20.
; Assumes AIAO function blocks - so does not contain client VCRs to
; talk to NMA or SMA of other devices.
;

[data link]
devClass = BASIC
nodeAddress= 0x20

[MIB]
devId= Safe AI - 1.0
pdTag= Safe Instrumentation
```

8.4.1.2 Safe AO

```
; Config file to be used in a basic device at address 0x20.
; Assumes AIAO function blocks - so does not contain client VCRs to
; talk to NMA or SMA of other devices.
;

[data link]
devClass = BASIC
nodeAddress= 0x22

[MIB]
devId= Safe A0 - 1.0
pdTag= Safe Instrumentation
```

8.4.2 Device Template

I have omitted some of the parameters in the blocks to save space. All of the parameters that constitutes the safety layer is shown in the template below:

```
VFD
       Safe Instruments       // vendor name
       S-AI                   // model name
       Rev1.0                 // revision
       12, 22                 // profile numbers
       1                      // number of user defined types
       1                      // number of transducer blocks
       1                      // number of function blocks
       10                     // maximum number of linkage objects
       10                     // maximum number of alert objects
       5                      // maximum number of trend float objects
       1                      // maximum number of trend Discrete objects
       1                      // maximum number of trend bitstring objects
       15                     // maximum number of variable list objects

USER_TYPE
9
       // type, size, offset
       6, 2, 0        // 1 - uint16 fieldNo
       6, 2, 2        // 2 - uint16 sender
       6, 2, 4        // 3 - uint16 dest
       7, 4, 6        // 4 - uint32 frameCounter
       8, 4, 10       // 5 - float floatTemp
       6, 2, 14       // 6 - uint16 shortTemp
       8, 4, 16       // 7 - float errorFlag
       21, 8, 20      // 8 - (time value) FF_Time newTime
       21, 8, 28      // 9 - (time value) FF_Time prevTime

BLOCKS

BLOCK
       RES-DEV1                     // tag
       RESOURCE                     // block type
       0x80020315, 0x80020310, 1    // characteristic DD item, DD item, DD revision
       0x010B, 0, 0, 0, 0           // profile, profile rev, execution time,
       40                           // number of parameters
              0xc001017a, 0x8002017a, ST_REV,  USER_PTR           // 1, ST_REV
              0xc0010180, 0x80020180, TAG_DESC,  USER_PTR         // 2, TAG_DESC
              0xc001017e, 0x8002017e, STRATEGY,  USER_PTR         // 3, STRATEGY
              0xc0010096, 0x80020037, ALERT_KEY,  USER_PTR        // 4, ALERT_KEY
              0xc0010126, 0x80020126, MODE_BLK,  USER_PTR         // 5, MODE_BLK
              0xc00100ac, 0x800200ac, BLOCK_ERR,  USER_PTR        // 6, BLOCK_ERR
              …
              0xc001018c, 0x8002018c, WRITE_ALM,  USER_PTR        // 40, WRITE_ALM

 BLOCK
       Transducer                   // tag
       TRANSDUCER                   // block type
       11, 10, 2                    // DD name, DD item, DD revision
       13, 23, 0, 0, 0              // profile, profile rev, execution time,
       8                            // number of parameters
              0xc001017a, 0x8002017a, ST_REV,  USER_PTR           // 1, ST_REV
              0xc0010180, 0x80020180, TAG_DESC,  USER_PTR         // 2, TAG_DESC
              …
              0, 0, ARRAY, Float, 5, C, S, USER_PTR               //8 f_array

 BLOCK
       SAI-1                        // tag
       FUNCTION                     // block type
       0x800201d7, 0x800201D0, 3004  // characteristic DD item, DD item, DD revision
       0x0101, 0, 2560, 0, 0        // profile, profile rev, execution time,
                                    // number of view3 and number of view 4
       41                           // number of parameters
              0xc001017a, 0x8002017a, ST_REV,  USER_PTR           // 1, ST_REV
              0xc0010180, 0x80020180, TAG_DESC,  USER_PTR         // 2, TAG_DESC
              0xc001017e, 0x8002017e, STRATEGY,  USER_PTR         // 3, STRATEGY
              0xc0010096, 0x80020037, ALERT_KEY,  USER_PTR        // 4, ALERT_KEY
              0xc0010126, 0x80020126, MODE_BLK,  USER_PTR         // 5, MODE_BLK
              0xc00100ac, 0x800200ac, BLOCK_ERR,  USER_PTR        // 6, MODE_PER
              0xc0010136, 0x80020136, PV,  USER_PTR               // 7, PV
              0xc001012a, 0x8002012a, OUT,  USER_PTR              // 8, OUT
              …
              // MY OWN CUSTOM SAFETY ADDITIONS:
              0, 0, RECORD, USER_TYPE1, C, D, USER_PTR            // 37, sParams
              0, 0, RECORD, VS Float, C, D, USER_PTR              // 38, fVal
              0, 0, RECORD, VS Float, IN, D, USER_PTR             // 39, sIn
              0, 0, RECORD, VS Float, C, D, USER_PTR              // 40, ack
              0, 0, ARRAY, int8, 25, C, D, USER_PTR               // 41, CRCarg[25]

TRENDS
…
VARLISTS
…
```

I have only appended the block definition from the template containing the SAO-1 block, because the rest of the block is almost identical to the SAI-1 template. If we compare the custom safety additions made in SAI-1 and SAO-1, we can se that they are identical except that SAI-1 gets an input-parameter (sIn) where the SAO-1 gets an output-parameter (sOut).

```
BLOCK
       SAO-1                        // tag
       FUNCTION                     // block type
       0x800201f7, 0x800201F0, 1    // DD info
       0x0102, 0, 2560, 0, 0
       35                           // number of parameters
              0xc001017a, 0x8002017a, ST_REV,  USER_PTR          // 1, ST_REV
              0xc0010180, 0x80020180, TAG_DESC,  USER_PTR        // 2, TAG_DESC
              0xc001017e, 0x8002017e, STRATEGY,  USER_PTR        // 3, STRATEGY
              0xc0010096, 0x80020037, ALERT_KEY,  USER_PTR       // 4, ALERT_KEY
              0xc0010126, 0x80020126, MODE_BLK,  USER_PTR        // 5, MODE_BLK
```

```
0xc00100ac, 0x800200ac, BLOCK_ERR,  USER_PTR        // 6, MODE_PER
…
0xc00100b0, 0x800200b0, CAS_IN,  USER_PTR           // 17, CAS_IN
…
// MY OWN CUSTOM SAFETY ADDITIONS:
0, 0, RECORD, USER_TYPE1, C, D, USER_PTR             // 31, sParams
0, 0, RECORD, VS Float, C, D, USER_PTR               // 32, fVal
0, 0, RECORD, VS Float, OUT, D, USER_PTR             // 33, sOut
0, 0, RECORD, VS Float, C, D, USER_PTR               // 34, ack
0, 0, ARRAY, int8, 25, C, D, USER_PTR                // 35, CRCarg[25]
```