BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 58, No. 3, 2010
DOI: 10.2478/v10175-010-0040-4

A safe programmable electronic system

W.A. HALANG[1*] and M. ŚNIEŻEK[2]

[1] Fernuniversität, Lehrstuhl für Informationstechnik, 58084 Hagen, Germany
[2] Faculty of Electrotechnics and Informatics, Rzeszów University of Technology, 2 Pola St., 35-959 Rzeszów, Poland

Abstract. A dual-channel computer architecture for utilisation in programmable logic controllers is presented. Faults can be detected by novel high-speed comparators with fail-safe operation. The cyclic operating mode of PLCs and a specification-level, graphical programming paradigm based on the interconnection of application-oriented standard software function modules are architecturally supported. Thus, by design, there is no semantic gap between the programming and machine execution levels enabling the safety licensing of application software by an extremely simple, but rigorous method, viz., diverse back translation.

Key words: programmable electronic systems, safety, fail-safe comparison, function block programming, software verification.

1. Introduction

Economical considerations impose stringent constraints on development and utilisation of technical systems. This holds for safety-related systems as well. Since manpower is becoming increasingly expensive, also safety-related systems need to be highly flexible in order to be adjustable to changing requirements at low costs. In other words, safety-related systems must be program-controlled. Thus, the use of hard-wired safety systems is rapidly diminishing in favour of computerised ones, and in daily life the significance of programmable electronic systems in safety-related applications is increasing rapidly.

In society, on the other hand, there is increasing awareness of and demand for dependable technical systems in order not to endanger human life and to prevent environmental disasters. Computer-based technical systems have the special property to consist of hardware and software. Hardware is subject to wear and to faults occurring at random and possibly being transient. These sources of non-dependability can, to a very large extent, successfully be coped with by applying a wide spectrum of redundancy and fault tolerance measures. In software, on the other hand, there are no faults caused by wear or environmental events. Instead, all errors are design errors, i.e., of systematic nature, and their causes are always – latently – present. Hence, dependability of software cannot be achieved by reducing the number of errors contained by testing, checks, or other heuristic methods to low levels, generally greater than zero, but only by rigorously proving that it is error-free. Taking the high complexity of software into account, only in exceptional cases this objective can be reached with the present state of the art. Therefore, safety licensing of systems whose behaviour is largely program-controlled is still an unsolved problem. It is exacerbated by the fact that object code, i.e., a program's only version actually visible to and executed by a machine, must be licensed, because the transformation of a program representation from source to object code by a compiler or assembler may introduce errors into the object code.

To provide a remedy for this unsatisfactory situation, and to make a step into the direction of realising a workable programmable electronic system for industrial use, which can be safety licensed in its entirety, in [1–3] the intrinsic properties of a special, but not untypical case identified in industrial automation were exploited. Here the complexity turns out to be manageable, because attention is restricted to rather simple computing systems in the form of programmable logic controllers, and since application domains exist giving rise to software of limited variability only, which may be implemented in a well structured way by interconnecting carefully designed and rigorously verified blocks of software functionality. Despite the mentioned restrictions of generality, this approach is scientifically relevant and technologically useful, since its application area comprises the technical systems in charge of safety-critical automatic control.

The leading idea followed throughout this design is to combine already existing software engineering and verification methods of highest trustworthiness with novel architectural support, and to custom tailor an execution platform. Thus, the semantic gap between software requirements and hardware capabilities is closed, relinquishing the need for not safety-licensable compilers and operating systems. By keeping the complexity of each system component as low as possible, the safety licensing of the hardware in combination with application software is enabled on the basis of well established and proven techniques.

2. Safety-related software engineering

Special emphasis is dedicated to the software side, since software dependability still needs to catch up with the one already

*e-mail: wolfgang.halang@fernuni-hagen.de

achieved by hardware. The novelty of this design consists in providing comprehensive support for software verification already in the hardware architecture. The particular method supported is diverse back translation [4], the only one endorsed by the licensing authorities for the verification of larger programs.

A programming paradigm allowing for software to be easy to grasp and to verify with respect to both source and object code is available in form of the graphical language Function Block Diagram (FBD) defined within the standard IEC 61131-3 [5]. With its long tradition in control engineering, graphical programming in form of function block diagrams as depicted in Fig. 1 is already well established in automation technology.

The function block diagram language consists of only four different structural elements:

1. instances of functions and function blocks, represented by rectangular symbols,
2. dataflow lines, i.e., connection lines,
3. names, i.e., identifiers, and
4. (external) connection points.

Functions and function blocks according to IEC 61131-3 are highly application-dependent and re-usable elementary units of application programming on a higher level of abstraction. In principle, they are objects having inputs and outputs of arbitrary data types, and are able to perform arbitrary processing. Functions do not have any internal states. After being executed, they yield exactly one data element as a result, which may be multivalued. Multiple named instances, i.e., copies, can be created of function blocks. Each instance possesses an associated designator and a data structure, which contains its output and internal variables as well as possibly its input variables. All values of the output variables and the internal variables in such a data structure persist from one execution of a function block instance to the next. Therefore, invocation of a function block with the same arguments may not necessarily yield the same output values. This is necessary to be able to express feedback and internal storage behaviour. Only the input and output values are accessible outside a function block instance, i.e., a function block's internal variables are hidden from the outside and are, thus, strictly protected. By the connection lines within a function block diagram a data flow is represented.

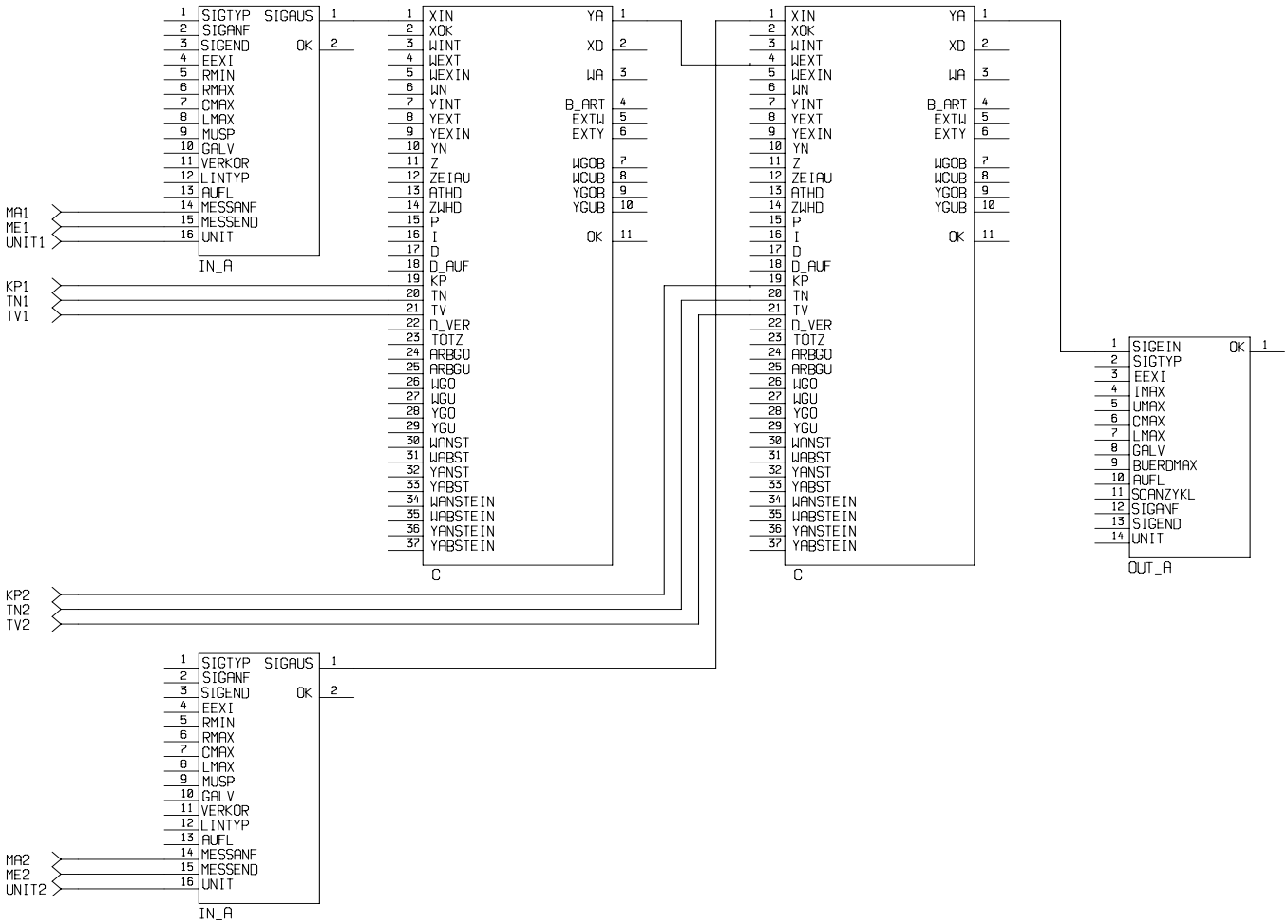


Fig. 1. A cascaded Proportional-Integral-Differential controller

Graphical, system-independent program development in form of function block diagrams is very easy, and is carried out in two steps:

1. *only once* building a library of functions and function blocks, and
2. *application-specific* interconnection of functions and function block instances.

The guideline VDI/VDE 3696 [6] states, that all automation programs occurring in process engineering can be constructed from a rather small library of (some 70) function blocks. Their correctness can be proven rigorously due to their limited complexities. Such blocks are used as elementary units of application programming. Safety-related automation software is then formulated graphically in form of function block diagrams according to IEC 61131-3. The following list gives an impression of these modules' functionalities:

- Monadic mathematical functions,
- Polyadic mathematical functions,
- Comparisons,
- Monadic Boolean function,
- Polyadic Boolean functions,
- Edge detectors,
- Selection functions,
- Counters, monostables, bistables, and timers,
- Process input/output,
- Network communication input/output,
- Dynamic elements and regulators,
- Conditioning for display and operation.

In order to give another typical example, the programming of emergency shut-down systems, which is usually performed graphically in form of functional logic diagrams to describe the mapping from Boolean inputs to Boolean outputs as functions of time such as, for instance,

> *if* a pressure is too high
> *then* a valve should be opened *and* an indicator should light up *after* 5 seconds

even requires as few as only four function modules, viz., three Boolean operators and a timer.

The still impossible formal verification of compilers transforming function block diagrams into conventional object code is not necessary, because only the block interconnections need to be verified. These are written as sequences of function block invocations and corresponding parameter passings, representing the application programs at the architectural level described in the next section. Only this part of the software is subject to application-specific verification by the architecturally supported method of diverse back translation, which can be employed very easily and cost-effectively, because the program code consists of only two types of instructions, and relates immediately to application-oriented objects. By inspecting this special kind of object code, function block diagrams describing control programs essentially on the *specification level* can be re-gained in a single and easy working step.

Many automation programs including safety-related ones have the form of sequence controls composed of steps and transitions. While in a step, an associated program, called action, represented as a function block diagram is being executed. For safety-related applications, linear sequences of steps and alternative branches of such sequences as shown in Fig. 2 are permitted, only. Parallel branches in sequential function charts must either be implemented by hardware parallelism or already resolved by the application programmer in form of explicit serialisation. Also, for clarity as well as for easy comprehension and verification only non-stored actions may be used. All other types of actions as defined in the language Sequential Function Charts according to IEC 61131-3 can be expressed in terms of non-stored ones and re-formulated sequential control logic.

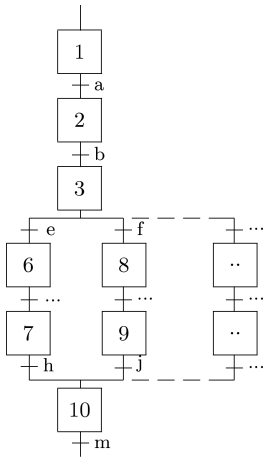


Fig. 2. Sequential function chart

3. Hardware architecture

When designing an execution platform closely matching and supporting software represented in form of function block diagrams and sequential function charts, it was not the objective to save hardware costs, but to facilitate the understandability of object programs and their execution process. This led to the architecture depicted in Fig. 3 with – conceptually – two different processors:

- a control flow processor (master) and
- a basic function block processor (slave).

These two processors are implemented by separate physical units. Thus, a clear and physical separation of concerns is achieved: execution of function blocks in the slave processor, and all other tasks, i.e., execution control, sequential function chart processing, and function block invocation, assigned to the master. This concept implies that application code is restricted to the control flow processor, on which project-specific safety licensing can concentrate. To enable the detection of faults in the hardware, a dual-channel configuration is chosen as displayed in Fig. 4, which also supports diversity as it allows for different master processors and different slave processors. All processing is simultaneously performed on two processors each, and all data communicated are subject to comparison.

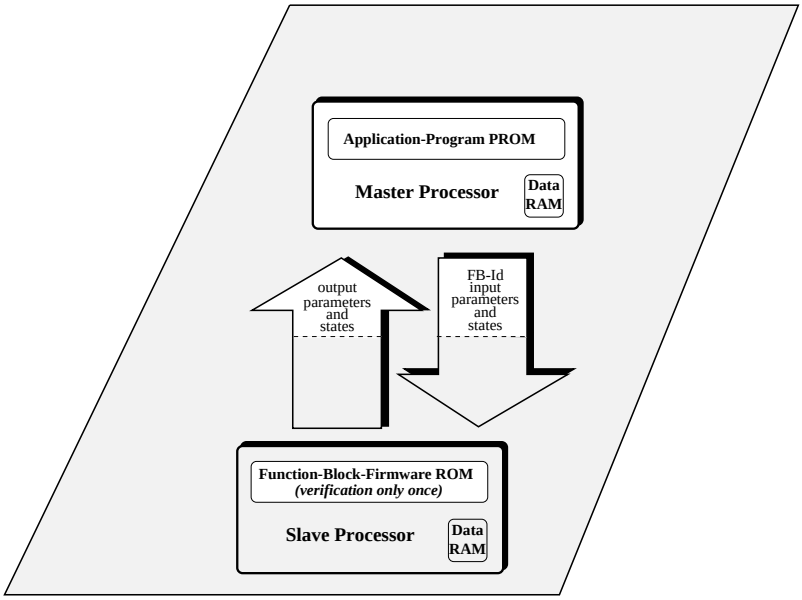


Fig. 3. Concept of a two-processor PES to execute FBD/SFC programs

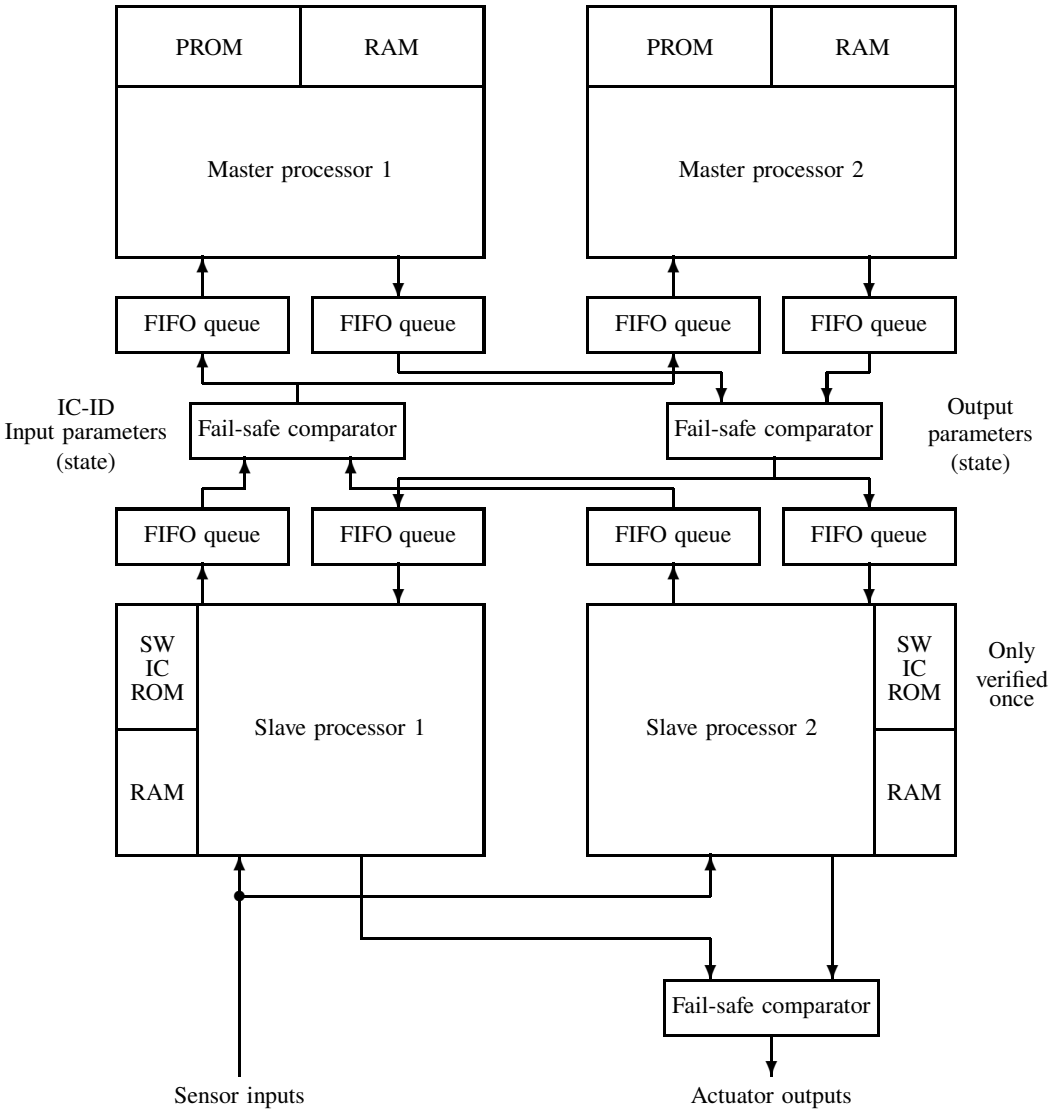


Fig. 4. Block diagram of fault – detecting dual-channel master-slave PES

The function block processors perform all data manipulations and take care of the communication with the environment. The master and slave processors communicate with each other through queues of first-in-first-out memory (FIFO). Clearly, the masters' and slaves' programs, though coordinated via communication, can be separated. This separation enables to migrate data access and data protection issues from software to hardware, thus increasing the controller's dependability. The master and slave processors execute programs in co-ordination with each other as follows. The master processors request the slaves to execute a function block by sending the latter's identification and the corresponding parameters and, if need be, also the block's internal state values via one of the FIFO queues to the slave processors. Here the object program implementing the function block is performed and the generated results and new internal states are sent to the master processors through the other FIFO queue. The elaboration of the function block ends with fetching these data from the output FIFO queue and storing them in the masters' RAM memories.

It is stressed that the results and internal states are stored in the masters' memories. The slaves' memories, if needed at all, are only used temporarily as scratchpads while elaborating function blocks. Hence, the slaves may be viewed as memoryless function co-processors or dedicated calculators. A number of fail-safe comparators as described in the next section checking the outputs from the master processors before they reach the slaves and vice versa completes a fault-detecting dual-channel configuration. Any inequality detected by the comparators generates an error signal (see below) which stops the entire system and sets the outputs to safe states. These states are provided by fail-safe hardware.

To prevent any modification by malfunctions, there is no program RAM, but all programs are provided in read-only memories (ROMs). The code of the function blocks resides in mask-programmed ROMs, which must be produced under supervision of and released by the licensing authorities, after the latter have rigorously established the correctness of the modules and their translation into object code. On the other hand, the sequences of module invocations together with the corresponding parameter passing, representing application programs at the architectural level, can be written into (E)PROMs by the user. This part of the software is subject to project-specific verification again to be performed by the licensing authorities, which finally still need to install and seal the (E)PROMs in the target systems. Here again it becomes obvious why the master/slave configuration was chosen, namely, to physically separate two system parts from one another: one whose software only needs to be verified once, and the other one performing the application-specific part of the software.

Besides program memory, the masters' address spaces also comprise RAM memory and various registers (cp. Fig. 5). The latter are:

1. the FIFO input register,
2. the FIFO output register,
3. two step registers, viz., step identifier and step initial address, and
4. the transition condition register.

Furthermore, it has a program counter (PC) and a single-bit step clock occurred register, which are not accessible to the programmer. Additionally, in the masters' address spaces other units are memory-mapped to create and receive control signals for the access of ROM, RAM, and FIFO queues. To fulfill their purpose, the master processors need just two instructions, viz.,

- MOVE and
- STEP.

The MOVE instruction has two operands, which directly point to locations in address space. Thus, the memories and the above-mentioned registers can be read and written. A read from a FIFO input register implies that the processor has to wait when the input FIFO queue register is empty. In case of writing into an output FIFO queue register, the processor also has to wait when the register is full. Execution of a MOVE implies incrementation of the program counter.

The programs executed by the master processors consist of sequences of steps. Behind the program segment of each step a STEP instruction, with a next-step address as operand, is inserted, which checks whether the segment was executed within a step cycle frame or not. The step cycle is a periodic signal generated by the system clock and establishing the basic time reference for operation as programmable logic controller (Fig. 6). The length of the cycle is selected in a way as to accommodate during its duration the execution of the most time-consuming step occurring in an application (class). If the execution of a segment does not terminate within a step cycle, an error signal is generated, which indicates an overload situation or a run time error. Then, program execution is stopped immediately, and suitable error handling is carried through by external fail-safe hardware. Normally, however, segment execution terminates before the instant of the next step cycle signal. Then, the processors wait until the end of the present cycle period. When the clock signal finally occurs, the step clock occurred registers are set. According to the contents of the transition condition registers it is decided, whether the step segment is executed once more, or whether the execution of the logically subsequent step is commenced, i.e., whether the program counters are re-loaded from the step initial address registers, or if another segment's initial program address is loaded from the STEP instruction's operand called next-step address. Since only one step is active at any given time, and since program branching is only possible in this restricted form within the framework of executing STEP instructions, this mechanism very effectively prevents erroneous access to code of other (inactive) steps as well as to program locations other than the beginnings of step segments.

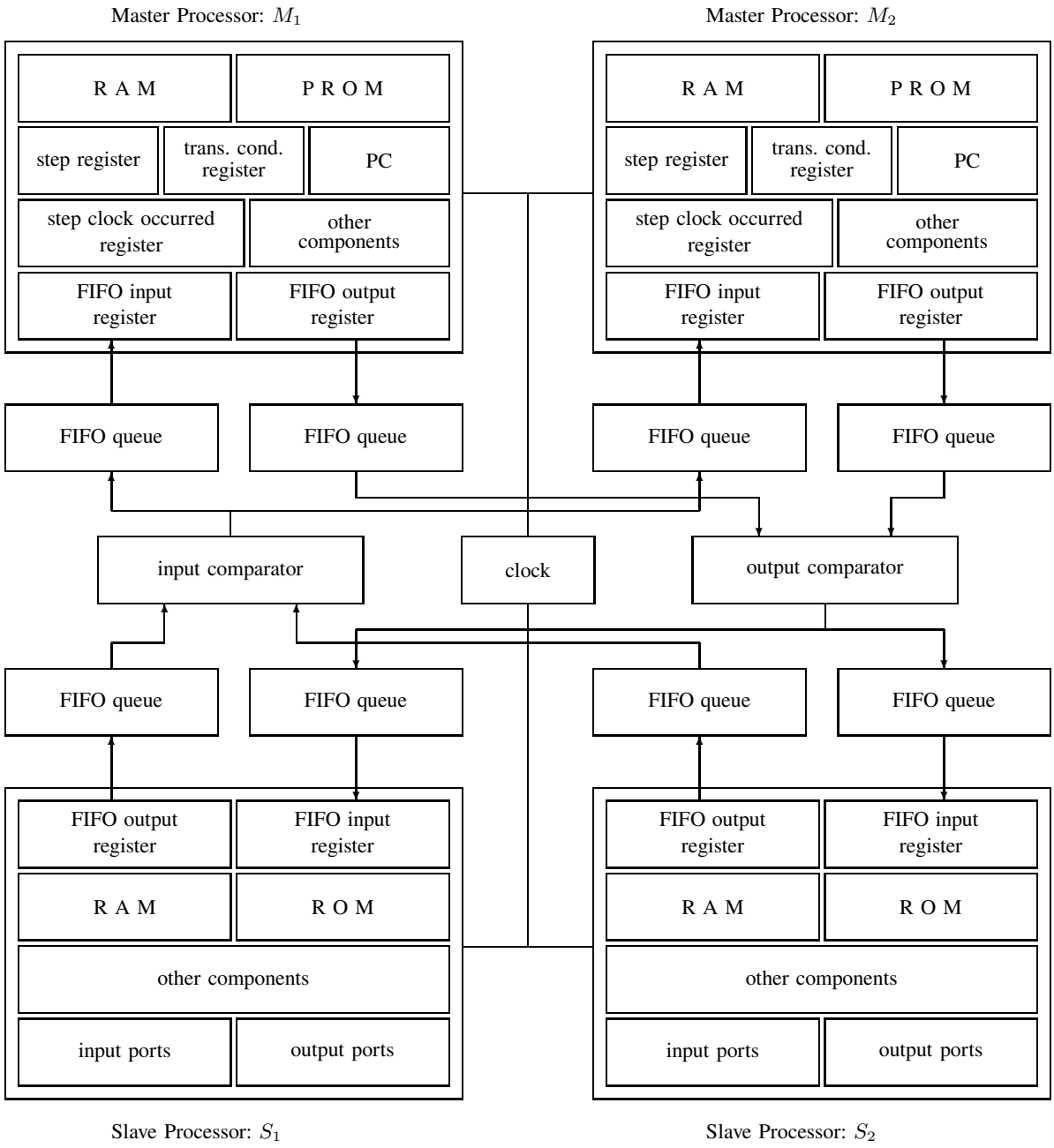


Fig. 5. Fault-detecting master-slave architecture

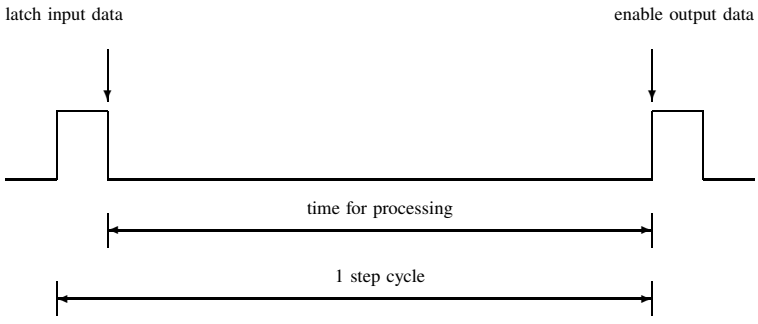


Fig. 6. Basic time reference for operation

The design objective of providing FIFOs is to implement easily synchronisable and understandable communication links, which decouple the master and slave processors with respect to their execution speeds. The FIFO queues consist of a fall-through memory and two single-bit status registers each, viz., FULL and EMPTY, which indicate the filling states of the FIFOs. The status registers are not user-accessible. They are set and reset by the FIFO control hardware and, if set, they cause a MOVE to a FIFO's input port or from an output port, respectively, to wait until space in the FIFO becomes available or data arrive. The comparison for equality of the outputs from the two master processors and of the inputs from the two slave processors, respectively, is carried out by the two fast comparators placed into the FIFO queues. Since the responsibility for detecting errors in the system rests on these comparators, they need to meet high dependability requirements and are, therefore, implemented in fail-safe technology as described below. A comparator is connected to two FIFOs' outputs. The first data elements from each input queue are latched and subsequently compared with each other. If both latches do not hold the same value, then an error signal is generated, which stops the operation of the entire system. Otherwise, the value is transferred into both output FIFOs. The comparison of FIFO data is shown in Fig. 7. By this set-up, errors that may appear in the processor modules and manifesting themselves by inconsistent data are detected by the comparators which continuously check consistency of data flow through the FIFO queues. They are triggered by any kind of inconsistency, equally by brief disturbances and permanent failures.

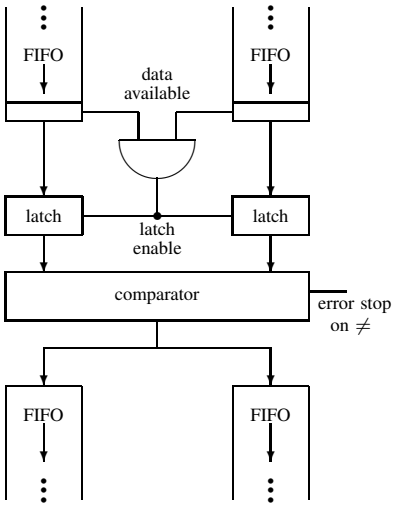


Fig. 7. Comparison of FIFO data

Communication with external technical processes takes place through fault-detecting input/output driver units attached to the slave processors. Output data words generated by the two slaves are first checked for equality in a fail-safe comparator and, subsequently, they are latched in an output port. If output data are not identical, an error signal is generated leading to a system stop. To achieve full determinism of execution time behaviour, the basic cycle was introduced as maximum step execution period. Although it exactly determines a priori the cyclic execution of the single steps, the processing instants of the various operations within a cycle, however, may still vary and, thus, remain undetermined. Since precisely predictable temporal behaviour is only important for input and output operations, temporal predictability is achieved as follows. Input data are read by the drivers at the beginning of each cycle and stored en bloc in two independent RAM buffers assigned to the respective slaves. The cycle start is signaled by the step clock occurred register. Only after that, the data are made available for further processing, thus providing predictability in timing. Output data generated by the slaves are latched in registers before the end of every cycle. When the step clock occurred register is set, the data are first checked for equality in fail-safe comparators and, subsequently, they are transferred to output ports to become effective to the environment. If output bytes are not identical, however, an error signal is generated leading to a system stop.

The FIFO queue and output comparators mentioned above are the components of a global comparator unit, which also receives operation monitoring signals from processor watch-dog timers and some correctness signals from other units. Based on all these, a global correctness signal, in other words, a negated global error, is generated and fed back to all units of the programmable electronic system. Naturally, each one can operate if this signal indicates "no error". Otherwise, the system stops and the outputs are set to safe states. The global error signal is also provided as output, allowing to trigger some external hardware as well.

4. Fast fail-safe comparator

The classical control components switches, relays, thermoswitches etc. have high probabilities to assume a certain "natural" switching state in case of failure, namely, the state of disconnection. This allows to design controllers performing safety-related tasks in such a way that in case of component failures the entire controllers assume states being safe for persons and the processes controlled ("fail-safe principle"). Semiconductor devices, the other hand, assume unpredictable states such as short-circuits or complete disruptions in cases of defect. Therefore, it is impossible to associate an unambiguous error state with a semiconductor component. One way to solve this problem is to use redundant controller structures. In technical applications this means that control components are replicated to fulfill the same task. Furthermore, switching to a state other than a safe one is subject to monitoring with majority or unanimity voting. Controlled devices or processes are brought to safe states in case of disagreement.

Fail-safe logic gates (And, Or, Not) were developed in special technologies which assure predictable behaviour of the outputs in case of any failure. Normally, such behaviour is equivalent to fall into the switched-off state. Galvanic separation of inputs and outputs is a common feature. There exist a few families of fail-safe gates applying different principles of operation. The HIMA Planar Logic [7, 8] and GTI

MagLog 24 [9] are well established in the market. The former logic family was the first one for which an official safety license was granted, in 1971, for use in electronic control systems. The mentioned fail-safe logic families, of which comparator circuits can be built, exhibit far too long reaction times, viz., several milliseconds. In the controller presented here, however, the reaction time of a comparator must correspond to the speed of data transfer between master and slave, i.e., a few microseconds per single comparison. This eliminates HIMA or MagLog modules from consideration, creating the need to design and construct a sufficiently fast comparator. Naturally, the principles of fail-safe technology must be followed.

Comparators are essentially combinatorial (memoryless) circuits. To increase safety, the outputs of the fast comparators are monitored by a much slower HIMA module. Since this is unable to detect very short signal changes, the comparators must be equipped with circuitry to hold the information on any – even intermittent or spurious – inconsistency occurring, in other words, they must be provided with memory. The holding circuit requires an initialisation signal. These considerations give rise to the following list of requirements to be met in developing fast fail-safe comparators:

- A comparator is activated with an initialisation signal, which must be kept for a while at high state after switching on the controller.
- When the comparator is active, the compared signals are equal, and all elements operate normally, its output is high.
- In case of any difference between compared signals or a hardware fault, the output is set to low, being also the safe state.
- Low level at the output is kept permanently, even if the signals compared have become equal again or the hardware resumed normal operation. Another activation of the comparator is possible by means of the initialisation signal, only.

Similarly as in HIMA fail-safe gates, alternating signals are used to carry information and generate output. The comparator designed [10] consists of a primary unit generating rectangular alternating signals, and a secondary one comparing the alternating signals. The primary unit is a 4-bit dual-channel comparator assembled of standard integrated circuits. To increase safety despite of employing semiconductor devices, two 4-bit comparator chips monitoring the same signal are employed in the primary unit. Such double comparison unit is still much smaller than a collection of single-bit comparators built of discrete elements. The secondary unit is a single-bit comparator of rectangular signals built in fail-safe technology.

The primary unit's circuitry diagram is shown in Fig. 8. The system compares two 4-bit words arriving via the lines A0–A3 and B0–B3. Two integrated comparator chips U1, U2 are used. The latter compares negated words for increased functional safety. If U1, U2 operate normally, the outputs Q1, Q2 are equal. The relatively slowly varying inputs are converted to alternating signals inside the comparators by means of the cascade inputs A=B (direct) and A<B (inverted), which receive rectangular signals from the square wave generator implemented with U5. As a consequence, the outputs Q1, Q2 produce either identical rectangular signals, when the words at A0–A3 and B0–B3 are equal, or low state. If a fault occurs inside the integrated comparator, one cannot predict the behaviour at the output. When the chip is still able to transmit the undisturbed rectangular wave, the fault remains undetected. In all other cases the outputs Q1, Q2 are different. The difference is recognised by the secondary unit, and then by the monitoring module which eventually stops the controller.

The secondary unit compares the signals Q1, Q2 produced by the primary one. Its circuitry diagram is shown in Fig. 9. The circuit is based on the design for fail-safe comparators of analogue signals presented in [11], some parts of which have been replaced and new values for its components selected to increase speed. The most important new feature of the digital comparator is the self-holding safe state, which it assumes after detecting any signal differences or own malfunctioning, and in which it then persists. Thus, as a whole, the comparator becomes a memory device in contrast to the one according to [11], which is a purely combinatorial circuit.

The signals Q1, Q2 are transmitted through transoptors U1, U2 provided that these are not blocked. Unblocking requires high state at the finger of potentiometer PR1, connected to the transoptor pins 7. After switching on, by means of relay PK1 the initialisation signal RESET connects PR1 to the voltage UC providing high state. When the entire unit is activated and high state appears at output VGL, the relay is switched off, but the voltage still remains at PR1, this time transmitted from VGL through the diodes D1–D3. Note that they do not allow the voltage UC to be transmitted to output VGL when the relay is still switched on, either. If there is no activation, there will be no voltage at PR1, so transoptors U1, U2 will remain blocked keeping output VGL at low state.

Rectangular signals from the outputs of U1, U2 affect, after being amplified by transistors T1, T3, transistors T2, T4 with transformers TR1, TR2 in their emitter circuits. In the secondary windings of TR1, TR2 pulses are generated which briefly unblock transistors T5, T6. If the Q1 and Q2 input signals are identical, the unblocking occurs simultaneously, triggering oscillations in the resonant circuit composed of capacitor C1 and the primary winding of transformer TR3. The oscillations are transmitted by TR3 and amplified by transistors T9, T10, provided that they are energised. Energising depends on transmission of oscillations by two other amplifiers formed of transistors T7, T8 and T11, T12, respectively. They use the transformers TR4, TR5 to convey energy. For safety reasons, T9 and T10 are fed from different circuits. Naturally, the transformers provide Galvanic separation as well. The signal produced by transistor T10 is amplified by T13 and passed to rectifier GL3 by transformer TR6. Capacitor C2 provides filtering together with R0. The voltage across C2 becomes the output signal VGL. As indicated above, VGL is fed back to unblock transoptors U1, U2. Capacitor C2 is continuously being charged with pulses appearing at TR1, TR2, and discharged via the input resistance R0 of the subsequent monitoring module and via potentiometer PR1.

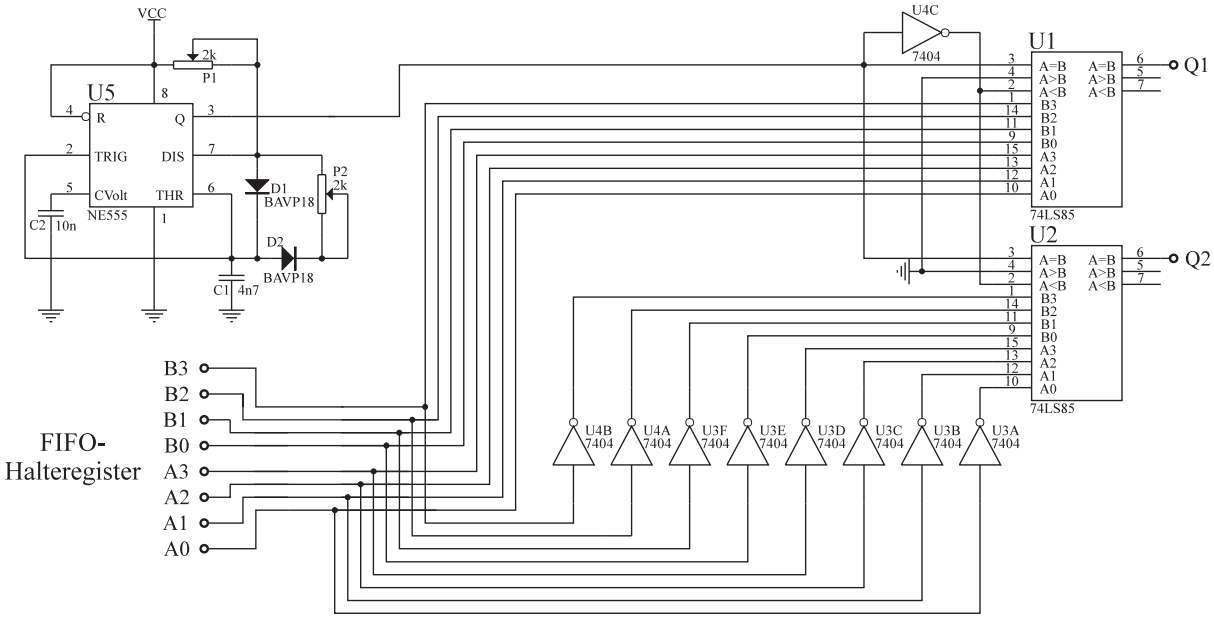


Fig. 8. Primary unit of the fast fail-safe comparator

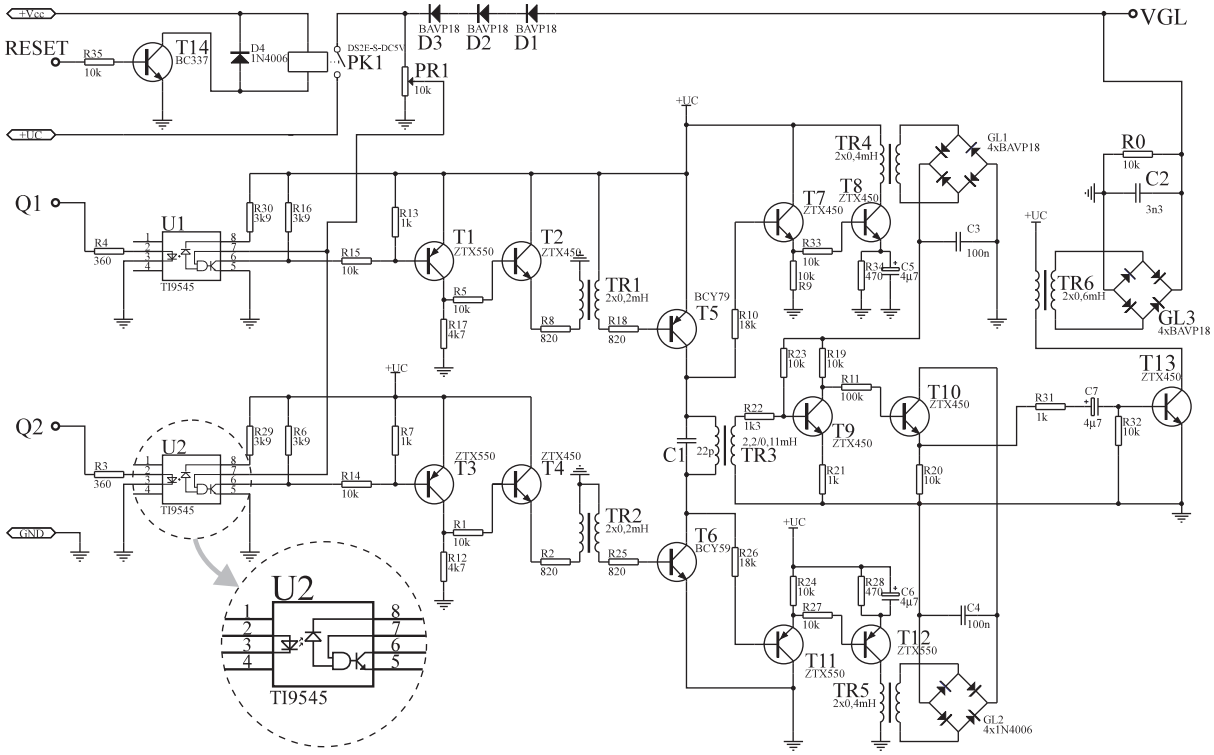


Fig. 9. Secondary unit of the fast fail-safe comparator

If one or both signals Q1, Q2 disappear temporarily or permanently, if these signals get out of phase, or if a device fails, the rectangular wave does not reach transformers TR1, TR2 anymore, oscillations are not generated, and the voltage across C2 goes down. This, in turn, blocks transoptors U1, U2 bringing this voltage completely to zero. Output VGL remains permanently in low state until relay PK1 is switched on for re-initialisation. This is the way the comparator's memory behaviour is realised.

The frequency of generator U5 in Fig. 8 producing the rectangular wave for the primary unit is 100 kHz. The comparator is very sensitive – if the secondary unit does not transfer a single pulse, output VGL becomes zero. Hence, the shortest time needed for comparison is 10 μs. Transfer to safe state takes about 40 μs because of the exponential discharge of capacitor C2. The monitoring module reacts after 10–20 ms setting the controller outputs to safe state.

5. Software verification

Corresponding to their development procedure, the verification of programs constructed in the form of function block diagrams is carried out in two steps:

1. Before being released, first all functions and function blocks contained in a library are verified employing appropriate, usually formal methods. Such a rather expensive safety licensing needs to be carried through only once for a certain application area after a suitable set of function blocks has been identified. The licensing costs justified by the safety requirements can, therefore, be spread over many implementations, leading to relatively low costs for each single automation project. In general, rather few library elements are sufficient to formulate all programs in a particular area of automation. As the details of the function blocks' implementation on the slave processors are part of the architecture, they remain invisible from the application programming point of view.
2. Then, for any given application program, only the correct implementation of the corresponding interconnection pattern of invoked functions and function block instances (i.e., a certain dataflow) needs to be verified.

Application software is safety licensed by subjecting the object code loaded into the master processors to diverse back translation [4]. This technique consists of reading machine programs out of computer memory and giving them to a number of teams working without any mutual contact. Only by human labour, these teams disassemble and decompile the code, from which they finally try to re-gain the specification. A safety license is granted to a software if its original specification agrees with the inversely obtained re-specifications. Needless to say that this method is, in general, extremely cumbersome, time-consuming, and expensive. This is due to the semantic gap between a specification formulated in terms of user functions on one hand and the usual machine instructions carrying them out on the other. Applying the programming paradigm of function block diagrams, however, a specification is directly mapped onto sequences of procedure invocations and parameter passing. It takes only minimum effort to verify a master program by interpreting such code, which just implements a particular module interconnection pattern, and by re-drawing the corresponding graphical program specification. Diverse back translation is especially well-suited to verify the correct implementation of graphically specified programs on the architecture introduced above for the following reasons:

- The method is essentially informal, easily comprehensible, and immediately applicable without any training. Thus, it is extremely well-suited to be used on the application-programming level by people with most heterogeneous educational backgrounds. Its ease of understanding and use inherently fosters error-free application.
- The effects of high-complexity utility and compiler-like programs, whose correctness cannot be established rigorously, are verified, too.
- Since graphical programming based on application-oriented function blocks has the quality of specification-level problem description, and because by design there is no semantic gap in the architecture of the execution platform between the levels interfacing to humans and to the machine, diverse back translation leads back in one easy step from machine code to problem specification.
- For this architecture, the effort required to utilise diverse back translation for the safety licensing of application programs is by several orders of magnitude smaller than for the von Neumann architecture, once a certain set of function blocks has been formally proven correct.

6. Example of program verification

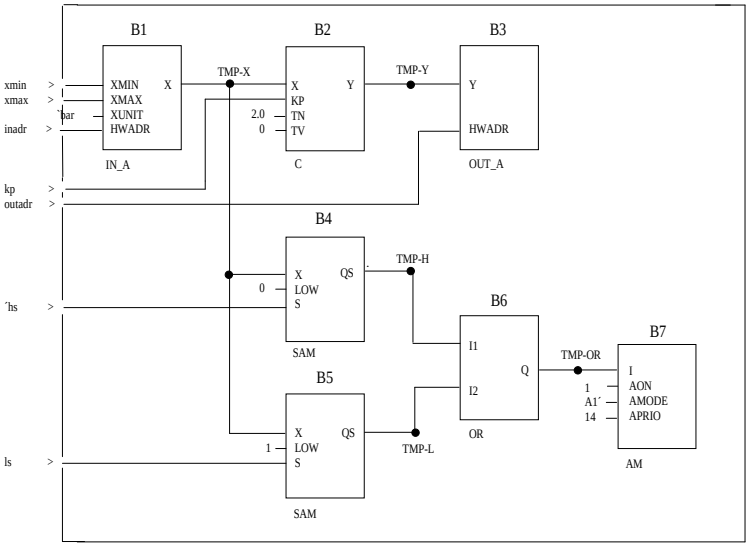
The application of back translation is now illustrated by elaborating a relatively simple, but realistic example. The program representation levels function block diagram and object code for the master processor are shown in full detail. It will become evident that it is straightforward and very easy to draw a function block diagram from a given object program establishing the feasibility of back translation as a software verification method for the presented architecture.

Figure 10 shows on the left a typical industrial automation program in graphical form. It performs supervision and regulation of a pressure. The program is expressed in terms of standard function blocks as defined in the guideline [6]. An analogue measuring value, the controlled variable, is acquired by a function block of type IN_A from the input channel with address INADR, and scaled within the range from XMIN to XMAX to a physical quantity with unit XUNIT. The controlled variable is fed into a function block of type C performing proportional-integral-differential (PID) regulation subject to the control parameters KP, TN, and TV. The resulting regulating variable is converted to an analogue value by a type OUT_A output function block, and switched onto the channel addressed by OUTADR. In addition, the controlled variable is also supervised, with the help of two instances of the SAM limit switch standard function block type, to be within the limits given by the parameters LS and HS. If the controlled variable is outside of this range, one of the QS outputs of the two SAM instances becomes logically true and, hence, the output of the type OR function block as well. This, in turn, causes the type AM alarm and message storing function block to create a timed alarm record. The inputs of the standard function blocks comprised by the program which are neither fed by externally visible inputs of the program itself nor internally by outputs of other standard function blocks are given constant values.

The object code of this example program for the master processor is listed on the right hand side of Fig. 10. It shows a (readable) assembly language version in which, for denotational simplification, MOVE instructions from memory to the memory-mapped FIFO input registers are denoted by $\leftarrow$, and from the FIFO output registers to memory by $\rightarrow$. Of the different function block types instantiated in the example, C, SAM, and AM have internal state variables, viz., C has three

and the other two types have one each. This object code illustrates that all function block instance invocations occurring in a program are directly mapped onto procedure calls. Each of them commences with a MOVE instruction, which transfers the identification tag (e.g., ID-C) of the corresponding block out of an appropriate ROM location to the slave's input FIFO. Then, the input parameters are supplied by reading appropriate ROM (for constants) or RAM (for program parameters and intermediate values) cells. Finally, if there are any, the values of the procedure's internal state variables are read from appropriate RAM locations. There is a set of correspondingly labeled (e.g., RAM-loc-B2-isv*i*) locations for each instance of a function block with internal states. When the slave processor has received all these data, it executes the procedure and returns, if there are any, values of output parameters and/or internal state variables, which are then stored into corresponding RAM locations. A connection between an output of one function block and an input of another one is implemented by two MOVE instructions: the former storing the output value in a RAM location for a temporary value (e.g., TMP-X), and the latter loading it from there. In other words, each connection in a function block diagram gives rise to exactly one transfer from the slave's output FIFO to a RAM cell, and to one or more transfers from there to the slave's input FIFO. The implementation details of the various procedures are part of the architecture's firmware and, thus, remain invisible.

According to the above-described structure of the masters' object programs, the process of back translation – disassemble and decompile object code – turns out to be very easy. To perform back translation, first the STEP instructions are searched, which clearly separate the different steps – in the sense of the language Sequential Function Chart – contained in a program from each other. The code between two STEP instructions corresponds to one function block diagram. Then, the first ← instruction is interpreted. It identifies a function block instance to be drawn into the function block diagram to be set up. By comparing the subsequent MOVE instructions with the function block's description contained in the library used, correct parameter passing can be verified easily. Moreover, for each such MOVE which corresponds to a proper parameter (and not to an internal state variable) a link is drawn into the diagram. There are two kinds of links. The first one are connections from program inputs or constants to inputs of function blocks, or from function block outputs to program outputs. The second kind are, so to speak, half connections, namely, from function block outputs to named connection points in the diagram, or from such points to function block inputs. When the diagram is completely drawn, the names of these points can be removed. With respect to the internal state variables, it needs to be verified that the corresponding locations in the master processors' RAM are correctly initialised, and that the new values resulting from a function block execution are written to exactly the same locations from where the internal states were read in the course of the block's invocation. The process of function block identification, parameter passing verification, as well as drawing of the block's symbol and of the corresponding connections is repeated until a STEP instruction is reached, which terminates the step and, thus, the corresponding function block diagram.



← ROM-loc-ID-IN_A	← RAM-loc-HS
← RAM-loc-XMIN	← RAM-loc-B4-isv
← RAM-loc-XMAX	→ RAM-loc-TMP-H
← ROM-loc-BAR	→ RAM-loc-B4-isv
← RAM-loc-INADR	← ROM-loc-ID-SAM
→ RAM-loc-TMP-X	← RAM-loc-TMP-X
← ROM-loc-ID-C	← ROM-loc-1
← RAM-loc-TMP-X	← RAM-loc-LS
← RAM-loc-KP	← RAM-loc-B5-isv
← ROM-loc-2.0	→ RAM-loc-TMP-L
← ROM-loc-0.0	→ RAM-loc-B5-isv
← RAM-loc-B2-isv1	← ROM-loc-ID-OR
← RAM-loc-B2-isv2	← RAM-loc-TMP-H
← RAM-loc-B2-isv3	← RAM-loc-TMP-L
→ RAM-loc-TMP-Y	→ RAM-loc-TMP-OR
→ RAM-loc-B2-isv1	← ROM-loc-ID-AM
→ RAM-loc-B2-isv2	← RAM-loc-TMP-OR
→ RAM-loc-B2-isv3	← ROM-loc-1
← ROM-loc-ID-OUT_A	← ROM-loc-A1
← RAM-loc-TMP-Y	← ROM-loc-14
← RAM-loc-OUTADR	← ROM-loc-B7-isv
← ROM-loc-ID-SAM	→ RAM-loc-B7-isv
← RAM-loc-TMP-X	STEP
← ROM-loc-0	

Fig. 10. Function block diagram of a program for pressure regulation and supervision and its master processor's object code representation

7. Conclusion

A real and pressing problem was addressed. Not all open questions in safety-related computing could be solved, but a beginning was made which is practically feasible and applicable to a wide class of common control problems. Hence, it is hoped that the concept presented here leads to the breakthrough that, ultimately, discrete or relay logic can be replaced by programmable electronic systems executing safety-licensed high-integrity software to take care of safety-critical functions in industrial processes. Meeting the need of society for more dependable computing systems under the prevailing economical restrictions, the concept is expected to give rise to workable industrial implementations.

In a constructive way, and using presently available methods and hardware technology only, for the first time a computer architecture was defined, which enables the safety licensing of complete programmable electronic systems including their software. Special emphasis was dedicated to the software side, since it is felt that software dependability still needs to catch up with the one already achieved for hardware. The solution presented deviates from the mainstream approach by using hardware as much as possible, but not necessarily in the most (hardware-) cost-effective way, and by enforcing the (re-) use of pre-engineered off-the-shelf software modules. The former deviation is in line with the technological development: there is cheap hardware in abundance and it ought to be used to achieve the objective of implementing inherently safe systems. The other deviation represents leaving the tradition of the von Neumann architecture allowing maximum flexibility – also to commit errors and to be unsafe.

REFERENCES

[1] W.A. Halang, S.-K. Jung, B. Krämer, and J. Scheepstra, *A Safety Licensable Computing Architecture*, World Scientific, Singapore, 1993.

[2] W.A. Halang and M. Śnieżek, "Digitale Datenverarbeitungsanlage für sicherheitsgerichtete Automatisierungsaufgaben zur Ausführung als Funktions- und Ablaufpläne dargestellter Programme", *German patent* 198 41 194 (1998).

[3] M. Śnieżek and W.A. Halang, *A Safe Programmable Logic Controller*, Oficyna Wydawnicza of Rzeszów University of Technology, Rzeszów, 1998.

[4] H. Krebs and U. Haspel, "Ein Verfahren zur Software-Verifikation", *Regelungstechnische Praxis* 26, 73–78 (1984).

[5] International Electrotechnical Commission Standard IEC 61131-3, *Programmable Controllers, Part 3, Programming Languages* Geneva, 1992.

[6] VDI/VDE-Richtlinie 3696, "Vendor independent configuration of distributed process control systems", Beuth Verlag, Berlin, 1995.

[7] Paul Hildebrandt GmbH & Co. KG, "Main catalogue – the HIMA planar system", *Brochure* HK 90.11, Brühl (1991).

[8] Paul Hildebrandt GmbH & Co. KG, "Fail-safe electronic controls – the HIMA planar system", *Brochure* TI 92.08. Brühl (1992).

[9] GTI Industrial Automation, *An Introduction to MagLog 24 Inherently Fail-safe Logic Technology – Eliminating the Unexpected*, GTI, Apeldoorn, 1993.

[10] W.A. Halang and M. Śnieżek: "Elektronischer Vergleicher zweier Binärworte mit ausfallsicherheitsgerichtetem Ausgabeverhalten", *German patent* 198 61 281 (1998).

[11] H. Schuck, "Analoger fensterkomparator in fail-safe-technik", *Doctoral dissertation*, Technische Universität Braunschweig, Braunschweig, 1987.