BGIA
Institut für Arbeitsschutz der
Deutschen Gesetzlichen Unfallversicherung

BGIA Report 7/2006e

Self-tests for microprocessors incorporating safety functions

or:

“Quo vadis, fault?”

Authors: Mario Mai, Günter Reuss
BGIA – Institute for Occupational Safety and Health of the German Social Accident Insurance, Sankt Augustin

Editorial office: Central division of the BGIA

Published by: German Social Accident Insurance (DGUV)
BGIA – Institute for Occupational Safety and Health of the German Social Accident Insurance
Alte Heerstr. 111, 53757 Sankt Augustin, Germany
Telephone: +49 / 02241 / 231 – 01
Fax: +49 / 02241 / 231 – 1333
Internet: www.dguv.de
– Mai 2009 –

ISBN: 978-3-88383-792-5

ISSN: 1869-3491

Self-tests for microprocessors incorporating safety functions
or:
"Quo vadis, fault?"

Abstract

The increasingly widespread use of microprocessors in safety-related products such as controls and sensors has led to particular requirements being placed upon their safety. The response of the controller in the event of a fault must be deterministic, and hazardous operations must be preventable. For such supplementary measures which are to be taken during the design of safe controls, this report describes processor tests by which the systems concerned are to be made suitably robust for safety-related applications. The measures presented here are software modules written in the Assembler programming language. Based upon the arrangements for error detection required by the standards, these measures enable the required level of safety to be attained in conjunction with the system architecture (structure) employed. The measures described represent snapshots of possible solutions, and should be regarded as examples only.

Selbsttests für Mikroprozessoren mit Sicherheitsaufgaben
oder:
„Quo vadis Fehler“?

Kurzfassung

Mit zunehmendem Einsatz von Mikroprozessoren in sicherheitstechnischen Produkten, wie Steuerungen und Sensoren, entstanden auch besondere Anforderungen an deren Sicherheit. Die Reaktion der Steuerung im Fehlerfall muss deterministisch sein und gefährliche Aktionen müssen verhindert werden können. Dieser Report beschreibt für den Teil der zusätzlichen Maßnahmen, die beim Bau sicherer Steuerungen getroffen werden müssen, Rechnertests, um diese Systeme für eine sicherheitstechnische Anwendung zu ertüchtigen. Bei den hier vorgestellten Maßnahmen handelt es sich um Softwaremodule in der Programmiersprache Assembler. Damit kann in Anlehnung an die in den Normen geforderten Fehleraufdeckungen, zusammen mit der jeweils verwendeten Systemarchitektur (Struktur), die notwendige Sicherheit erreicht werden. Die Beispiele sind Momentaufnahmen möglicher Lösungen, die aber keinen Anspruch auf Vollständigkeit erheben können.

Tests automatiques pour les microprocesseurs ayant des tâches de sécurité
Ou :
« Défaut quo vadis » ?

Résumé

L’utilisation croissante de microprocesseurs pour les produits dédiés à la sécurité, comme les commandes et les capteurs, a entraîné des exigences particulières concernant leur sécurité. La réaction de la commande en cas de défaut doit être assurée et les opérations dangereuses doivent pouvoir être empêchées. En ce qui concerne les mesures supplémentaires devant être prises pour la construction de commandes sûres, ce rapport décrit les tests automatisés permettant une utilisation de ces systèmes dans des applications de sécurité. Les mesures présentées ici traitent de modules logiciels dans le langage de programmation Assembleur. La sécurité nécessaire peut ainsi être atteinte selon les découvertes de défauts exigées dans les normes, avec l’architecture de système utilisée dans chaque cas. Les exemples sont des épreuves instantanées de résolutions possibles qui ne peuvent cependant pas prétendre être exhaustifs.

Autoverificación de microprocesadores dotados de funciones de seguridad o:

¿„Quo vadis error“?

Resumen

Con el creciente empleo de microprocesadores en aplicaciones relevantes para la seguridad, como mandos y sensores, también surgen exigencias específicas en razón a su seguridad. La reacción del mando en caso de fallo deberá ser determinística y acciones peligrosas deberán poderse evitar. El presente Report expone pruebas por ordenador para aquellas medidas adicionales, que deberán emprenderse a la hora de fabricar mandos seguros, a fin de habilitar semejantes sistemas para aplicaciones en razón de la seguridad. Las medidas presentadas se refieren a módulos de software en el lenguaje de programación Assembler. Con ello, y en conjunto con la respectiva arquitectura de sistema (estructura) utilizada, se puede alcanzar la seguridad requerida en conformidad con la detección de errores exigida por la normativa. Los ejemplos presentados son instantáneas de posibles soluciones, que no pretenden brindar respuestas a todas las interrogantes del caso.

Contents

1 Introduction 9

2 Type of self-tests 11

2.1 Microprocessor system tests 12

2.2 Peripherals tests 13

3 Tests of internal blocks and units of the CPU 15

3.1 Basic tests 15

3.1.1 Program counter test (PC_TEST.ASM) 15

3.1.2 Accumulator test (ACC_TEST.ASM) 17

3.1.3 PUSH, POP and RET stack instruction test (PPR_TEST.ASM) 18

3.2 Advanced instruction tests 19

3.2.1 Jump if not zero (JNZ_TEST.ASM) 20

3.2.2 Arithmetic instructions (ARI_TEST.ASM) 22

3.2.3 Logic instructions (ANL_TEST.ASM, ORL_TEST.ASM, XRL_TEST.ASM and CRS_TEST.ASM) 23

3.2.4 Logic instructions (BIT_TEST.ASM) 24

3.2.5 Transfer instructions (TRANTEST.ASM) 27

4 Memory tests 29

4.1 Program memory test (ROM_TEST.ASM) 29

4.2 Data memory test (XRAMTEST.ASM) 32

5 Special function register test (SFR_TEST.ASM) 37

6 Port tests (IO_TEST.ASM) 39

7 Main program 43

8 Concluding remarks 45

9 References 47

1 Introduction

Modern technology is now inconceivable without the microprocessor, and its use is now taken for granted in complex controls [1] and modern protective devices such as those used for the protection of persons on machinery and plant. Only by the use of such components can flexibility, versatility, reliability and adaptability to the production process be attained, not least also at acceptable cost.

In order for a microprocessor, which is a very complex component with a fault-mode behaviour which cannot be defined, to be made fit for use in safety-related applications, measures must be taken by means of which failures during operation can be detected and an appropriate safety-related response initiated. In practice, this means that besides fulfilling its essential function, the CPU must also continually perform a self-test. During this test, certain operations are executed, and the results compared with the expected results. An identified deviation from the expected results must trigger a defined, generally safety-related response.

In order for the various processing units and peripheral elements of a microprocessor system to be tested, special tests are required for each of these elements, such as a program memory test, a data memory test, instruction tests, etc. Once all tests have been passed, it is assumed that the system is adequately fault-free.

The tests described in this report were developed on a type 80C537 microcontroller. They are programmed and commented in modular form. Since the particular hardware features of this controller were not exploited during development, transfer of the tests to other derivatives of this popular processor family should not present much difficulty. Porting to other processor architectures should also present no problems. Attention must of course be paid to any major differences between hardware, and to the associated need for software adaptation. The essential principles upon which the design of these microprocessor tests is based are intended to provide valuable insights and assistance. Information on the effectiveness and frequency of the self-tests can be found in IEC 61508/VDE 0803 [2].

The software has been developed with care and in accordance with up-to-date good practice. It is made available to the user free of charge.

Users use the software at their own risk. To the extent permissible by law, no liability will be accepted for the software on any legal basis. In particular, no liability will be accepted for material defects or defects in title, whether in the software or in the associated documentation and information, particularly with regard to their correctness, freedom from errors, freedom from property rights and copyright of third parties, up-to-dateness, completeness and/or fitness for purpose, except in cases of malicious or wrongful intent.

The BGIA – Institute for Occupational Safety and Health of the German Social Accident Insurance undertakes to keep its website free of viruses; nevertheless, no guarantee can be given that the software and information provided are virus-free. Users are therefore advised to take appropriate precautions of their own and to use a virus scanner before downloading software, documentation or information.

The software has been developed by staff at the BGIA independently of customers and their applications. The results are therefore based upon the requirements of the applicable standards and can be made available in response to queries.

2 Type of self-tests

The tests implemented here can be divided essentially into two classes. The first of these are microprocessor tests: these test proper operation of the central processing unit (CPU), the on-chip peripheral elements, and the internal and/or external random-access memory (RAM). The second class (peripheral tests) concerns tests of the interfacing between the microprocessor system and the outside (for example via the ports) for errors. Whereas microprocessor tests are largely independent of the practical application concerned, port tests for example must give consideration to connected external units which are to be controlled. The relevant hardware becomes an essential part of these peripheral tests. Its characteristics, such as filter effects, may therefore have a direct influence upon implementation of the corresponding port test [3].

Figure 1 shows a schematic diagram of a microprocessor system structure in wide-spread use in safety technology.

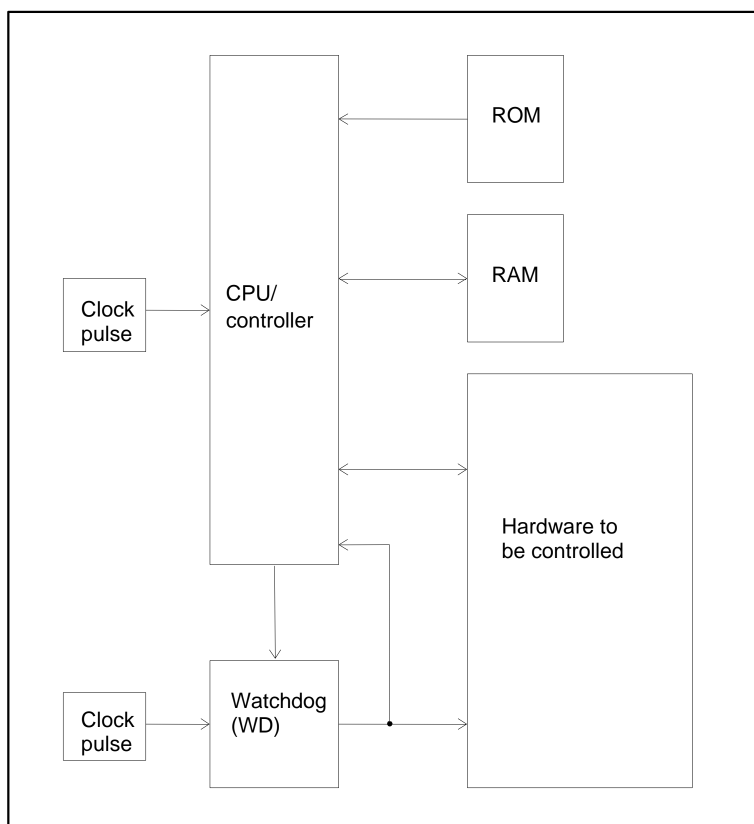


Figure 1:
General block structure of a microprocessor system

The microprocessor system consists of the CPU, which is the heart of the system; the read-only memory (ROM), into which the program for the CPU is loaded; and the

RAM, into which data are loaded during operation. A non-volatile, rewritable memory module may also be employed. The use of such a memory module is dependent upon the application, and is not considered in this model system. The model system is connected interactively, via suitable interfaces, to the hardware to be controlled.

An essential component in the microprocessor system shown here is the watchdog (WD), which monitors the function of the CPU. This structure requires the WD to have an independent de-energization procedure by means of which the hardware to be controlled can be switched into the safe state irrespective of the status of the CPU. A criterion for this is that the system must possess a safe state which can be attained by disconnection of the power source. The WD has a clock cycle of its own independent of that of the CPU, since the independence of the WD for de-energization would not be assured were it to share a common clock cycle.

Another feature of the WD, not shown here but nevertheless important, is that it is protected by a time window against multiple triggering within the time window. Selection of the time window enables the program to be coupled more closely to the program of the CPU. Multiple triggering within the time window may indicate that the program is being executed incorrectly and be used to initiate the safe state, as with failure of the trigger pulse to occur.

The structure of the microprocessor system shown in Figure 1 may vary with regard to the location of the memory. For the design of the self-test, it is largely irrelevant whether the memory (RAM, ROM) is integrated into the chip. The principle is that the control system must initiate or maintain the safe state of a machine or installation in the event of a self-test returning a negative result.

2.1 Microprocessor system tests

Examples of the following microprocessor system tests are provided below:

- Basic tests
  - Accumulator test, test of elementary jump instructions
  - Stack instruction test
  - Program counter test (jump to program “islands”)

- Advanced instruction tests
  - Test of the transfer instructions
  - Test of the logical operations
  - Test of the arithmetic operations
- Memory tests
  - Program memory test (ROM test)
  - Data memory test (RAM test)
- Special function register test
  - Test of the registers used for control of internal peripheral components (timers, counters, etc.)

2.2 Peripherals tests

Peripherals tests include:

- Test of proper operation of the I/O ports
- Test of the external peripherals

3 Tests of internal blocks and units of the CPU

In order to illustrate the principal aspects, extracts from the assembler source code are shown in some of the following sections. For an explanation of variables, constants and functions, please refer to the complete source code. A declaration file (DEC.ASM), in which the constants and memory locations of the variables are defined, is available for this purpose.

3.1 Basic tests

3.1.1 Program counter test (PC_TEST.ASM)

The program counter is a component of a CPU which determines which point of the program to be run is to be executed. Should this pointer fail to operate correctly, owing for example to a stuck-at error of certain bits or crosstalk of bits, proper execution of the program is no longer assured. Instructions will be skipped, and the program will be executed at a completely different point to that intended by the programmer.

As a result, proper operation of the CPU is no longer assured. In order for such faults in the program counter to be detected and for the microprocessor to be placed in a safe state, a test of the program counter must be performed. In this example, jumps are performed to program "islands" located in different address ranges of the program memory, in order for a computing operation involving a comparison to be conducted there. Should a fault be detected, an endless loop is entered. As a general principle, entry of an endless loop leads to the external WD no longer being retriggered, as a result of which the system is placed in the safe state by means of the second de-energization procedure.

Implementation of the program counter test is shown in broad terms in Figure 2 (see Page 16).

Figure 2:
Program counter test procedure

PC_TEST

Program island 4000 h

Error occurred?
Yes
No

Program island 4100 h

Error occurred?
Yes
No

Program island 4200 h

Error occurred?
Yes
No

Program island 4400 h

Error occurred?
Yes
No

Program island 4800 h

Error occurred?
Yes
No

Program island 5000 h

Error occurred?
Yes
No

Program island 6000 h

Error occurred?
Yes
No

Program island 8000 h

Error occurred?
Yes
No

End of function

3.1.2 Accumulator test (ACC_TEST.ASM)

The accumulator is the central register of the CPU and is involved in virtually all operations. In many cases, it serves as both the source register and target register for a large proportion of the instructions that are executed. Proper operation of this register is therefore highly relevant to faultless operation of the entire CPU. Correct operation of the accumulator is tested by means of the test program. For this purpose, the test program "pushes" a "1" through the accumulator; upon completion of the push routine, it checks whether the routine has required the same number of push operations as intended during design of the test. In the event of an error, a corresponding number is loaded into an error memory, and the program enters an endless loop.

```
; Module: ACC_TEST.ASM

; Version 1.00          BIA, sub-division 5.2                  Date: 15.09.2000

; Function:
; Test of the accumulator by means of a walking 1. The background cannot be tested,
; since the register is required as an arithmetic register for background analysis; its
; content is therefore constantly changing. Owing to its frequent use, however, faults in
; the accumulator are also detected by other means. Should the carry be set after fewer
; than 7 steps, the cycle counter has not yet reached 0, and an accumulator fault is
; therefore probable.
; This routine could also be modified such that the initial value of R1 is 07. The additional
; DEC R1 command would then not be necessary. In the arrangement shown however,
; the carry is also tested deliberately after 7 iterations of the cycle, with the result that
; firstly, JC is tested, and secondly, a continuous 00 in the accumulator is detected.

; Input parameters: none
; Output parameters: none
; Changed registers: A, R1, PSW
; Changed memory ranges: none

; Exit with error: none

; Higher-level modules: CPU_TEST.ASM
; Lower-level modules: none
; Called by: MAIN
; Calls: -
; Stack depth: 2 bytes
; Register bank used: current bank
; ***************************************************************************
ACC_TEST:      CLR     C
               MOV     R1,#08               ; Let R1 be the cycle counter
               MOV     A,#01                ; Test byte: 0000$0001
```

```
ACC_1:      RLC     A                   ; Walking 1 owing to carry flag
            JC      ACC_2               ; If the 1 has arrived in the carry, jump to ACC_2
            DJNZ    R1, ACC_1           ; Decremented cycle counter
ACC_2:      DEC     R1                  ; Decremented cycle counter again
            CJNE    R1, #00h, ACC_ERR   ; The accumulator test proper is performed here
            SJMP    ACC_ENDE            ; End of the routine
ACC_ERR:    MOV     FEHLER,#FEHLER_ACC  ; Load error memory
            SJMP    ACC_ERR             ; Endless loop
ACC_ENDE:   RET
```

3.1.3 PUSH, POP and RET stack instruction test (PPR_TEST.ASM)

The PUSH and POP instructions have the function of placing data temporarily on the stack. These instructions are often used in functions in order to "rescue" register contents following function entry and to restore the previous register content prior to function exit. During this test, a check is performed of whether the stack pointer is incremented and decremented correctly. In addition, a check is performed of whether the value stored in the stack is retrieved again correctly from it.

The function of the RET instruction is to leave subroutines again; the return address has been placed in the stack automatically beforehand by the CPU. In order to test the RET command, a defined return address is placed in the stack, and the RET instruction then executed. The CPU must then jump to the previously determined address and continue program execution from that point.

```
; Module: PPR_TEST.ASM

; Version 1.00          BIA, sub-division 5.2                   Date: 15.09.2000

; Function:
; Proper operation of the PUSH and POP instructions and of the RET instruction are tested.

; Input parameters: none
; Output parameters: none
; Changed registers: A, R0, R1, PSW
; Changed memory ranges: none

; Exit with error: branch to ERROR function

; Higher-level modules: CPU_TEST.ASM
; Lower-level modules: none
; Called by: MAIN
; Calls: -
```

```
; Stack depth: 4 bytes
; Register bank used: current bank
; ***************************************************************************
PPR_TEST:
PPR_PUSHPOP:    MOV     R0,SP               ; Save stack pointer position
                MOV     R1,SP
                INC     R1                  ; Create address of the byte to be pushed
                MOV     A,#0AAh
                PUSH    ACC
                MOV     A,R1                ; Expected result and comparison
                CJNE    A,SP,PPR_err1

                CLR     A
                MOV     A,@R1
                CJNE    A,#0AAH,PPR_err1    ; Expected result and comparison

                CLR     A
                POP     ACC
                CJNE    A,#0AAH,PPR_err1    ; Expected result and comparison

                MOV     A,R0                ; Expected result and comparison
                CJNE    A,SP,PPR_err1


PPR_RET:        MOV     DPTR,#PPR_RET1      ; Save return address in 16-bit register
                PUSH    DPL                 ; Return address in stack
                PUSH    DPH
                RET                         ; Return (to PPR_RET1)
                NOP
                NOP
                NOP
                NOP
                NOP
                SJMP    PPR_err1
PPR_RET1:       SJMP    PPR_ENDE
                NOP
                NOP
                NOP
                NOP
                NOP
PPR_err1:       MOV     FEHLER,#FEHLER_PPR  ; Load error memory
                LJMP    ERROR
PPR_ENDE:       RET
```

3.2 Advanced instruction tests

In order to test proper functioning of the individual machine instructions, each instruction has been used within the instruction test in all possible addressing variants, and the result compared with an expected result.

3.2.1 Jump if not zero (JNZ_TEST.ASM)

In this conditional branch, a jump is made dependent upon whether the accumulator contains the value zero or a value not equal to zero. In order to perform the test, a value is written into the accumulator, and the same value into the auxiliary accumulator. The two values are subtracted from each other and the result of the operation anticipated in the accumulator. Following this subtraction operation, the JNZ instruction is used to check whether the result is correct. In the event of an error, the error memory is written to, and the program enters an endless loop. The check is performed with two different values. Should, during execution of a jump, the stack pointer point into the body of an instruction, the NOP instructions have the function of clearing the STACK and enable the CPU to "recover" during the next iteration.

```
; Module: JNZ_TEST.ASM

; Version 1.00          BIA, sub-division 5.2                   Date: 15.09.2000

; Function:
; Test by JNZ for complementary accumulator contents AAh
; and 55h at 5 return steps.

; Input parameters: none
; Output parameters: none
; Changed registers: A, B, PSW
; Changed memory ranges: none

; Exit with error: none

; Higher-level modules: CPU_TEST.ASM
; Lower-level modules: none
; Called by: MAIN
; Calls: -

; Stack depth: 2 bytes
; Register bank used: none
; ***************************************************************************
JNZ_TEST:      CLR     C
               MOV     A,#0AAh               ; Load first test byte into accumulator
               MOV     B,#0AAh               ; Load first test byte into register B
               SUBB    A,B                   ; Compare A and B by subtraction
Error1:        NOP
               NOP
               NOP
               NOP
               NOP
```

```
                MOV     FEHLER,#FEHLER_JNZ      ; Load error memory
                JNZ     Error1                  ; Should an error occur in the registers or
                                                ; during subtraction, JNZ forces an endless loop,
                                                ; the relative address in this case is always 05.
                MOV     FEHLER,#0               ; Clear error memory
                JZ      Error2                  ; If result OK, jump to CJNE test
                MOV     FEHLER,#FEHLER_JNZ      ; Load error memory
                NOP
                NOP
                NOP
                NOP
                NOP
Error8:         SJMP    Error8                  ; Both jumps have failed > endless loop
Error2:         NOP
                NOP
                NOP
                NOP
                NOP
                MOV     FEHLER,#FEHLER_JNZ      ; Load error memory
                CJNE    A,#00h,Error2           ; Should JNZ be ignored, the controller
                                                ; becomes trapped in an endless loop
                MOV     FEHLER,#0               ; Clear error memory

; Same procedure for the value 55h
                MOV     A,#055h                 ; Load second test byte into accumulator
                MOV     B,#055h                 ; Load second test byte into register B
                SUBB    A,B                     ; Compare A and B by subtraction
Error3:         NOP
                NOP
                NOP
                NOP
                NOP
                MOV     FEHLER,#FEHLER_JNZ      ; Load error memory
                JNZ     Error1                  ; Should an error occur in the registers or
                                                ; during subtraction, JNZ forces an endless loop,
                                                ; the relative address in this case is always 05.
                MOV     FEHLER,#0               ; Clear error memory
                JZ      Error4                  ; If result OK, jump to the CJNE test
                MOV     FEHLER,#FEHLER_JNZ      ; Load error memory
                NOP
                NOP
                NOP
                NOP
                NOP

Error9:         SJMP    Error9                  ; Both jumps have failed > endless loop
Error4:         NOP
                NOP
                NOP
```

```
                NOP
                NOP
                MOV     FEHLER,#FEHLER_JNZ    ; Load error memory
                CJNE    A,#00h,Error4         ; Should JNZ be ignored, the controller becomes
                                              ; trapped in an endless loop
                MOV     FEHLER,#0             ; Clear error memory
                RET
```

3.2.2 Arithmetic instructions (ARI_TEST.ASM)

To permit testing of the CPU's various arithmetic instructions, selected values are written into the relevant registers. The arithmetic operation is executed, the result compared with the expected value, and if applicable a branch is performed to an error routine together with a saved error number. The tests also take into account different addressing variants of the arithmetic instructions such as direct addressing, indirect addressing or an operation with a concrete value. By way of example, only one of the tests is shown here.

```
; Module: ARI_TEST.ASM

; Version 1.00          BIA, sub-division 5.2                    Date: 15.09.2000

; Function:
; Test of arithmetic instructions: ADD, ADDC, SUBB, INC, DEC, MUL, DIV, DA

; Input parameters: none
; Output parameters: none
; Changed registers: see individual functions
; Changed memory ranges: none, except for memory cells specially reserved
; for these tests

; Exit with error: branch to ERROR function

; Higher-level modules: LOGITEST.ASM
; Lower-level modules: none

; Register bank used: current bank
; ***************************************************************************



; ******************** ADD_TEST 1 *****************************************
; Function: test of ADD A,R0

; Called by: ADD_TEST
```

```
; Calls: -

; Changed registers: A, R0, PSW
; Stack depth: 2 bytes
; ***************************************************************************
ADD_TEST1:      MOV     A,#0AAh                 ; Load test pattern 1
                MOV     R0,#55h                 ; Load test pattern 2
                ADD     A,R0                    ; Instruction test
                CJNE    A,#0FFh,ADD_err1        ; Comparison with expected result
                RET
ADD_err1:       LJMP    ERROR_ARI
```

Other tests: see source code of ARI_TEST.ASM

3.2.3 Logic instructions (ANL_TEST.ASM, ORL_TEST.ASM, XRL_TEST.ASM and CRS_TEST.ASM)

The test of the logic instructions is similar to that of the arithmetic instructions. Here too, selected values are written to the relevant registers, the operations are executed, and the result is compared with the expected result. Different addressing variants are considered. Once again, only one of the tests is shown by way of example.

```
; Module: ANL_TEST.ASM

; Version 1.00          BIA, sub-division 5.2                   Date: 15.09.2000

; Function:
; The subroutines test the logic instruction ANL as follows: an AND
; operation is performed on AAh (10101010B) and 56h (01010110B)
; and the computed result compared with the expected result of 02h (00000010B).

; Input parameters: none
; Output parameters: none
; Changed registers: see individual functions
; Changed memory ranges: none, except for memory cells specially reserved for
; these tests

; Exit with error: branch to ERROR function

; Higher-level modules: LOGITEST.ASM
; Lower-level modules: none
; Called by: ANL_TEST
; Calls: -

; Stack depth: 2 bytes
```

```
; Register bank used: current bank
; ***************************************************************************



; ******************** ANL_TEST 1 ******************************************
; Function: test of ANL A,R0
; Changed registers: A, R0, PSW
; ***************************************************************************
ANL_TEST1:     MOV     A,#0AAh               ; Load test pattern 1
               MOV     R0,#56h               ; Load test pattern 2
               ANL     A,R0                  ; Instruction test
               CJNE    A,#02h,ANL_err1       ; Comparison with expected result
               RET
ANL_err1:      LJMP    ERROR_ANL
```

Other tests: refer to source code of ANL_TEST.ASM, ORL_TEST.ASM, XRL_TEST.ASM, CRS_TEST.ASM

3.2.4 Logic instructions (BIT_TEST.ASM)

In these tests, the bit-addressable memory range and proper execution of the bit instructions are tested.

The memory range is tested by the deliberate modification of bits followed by inspection of the rest of the bit-addressable memory range for changes. The bit-oriented logic instructions are tested in a separate function by application of the various bit instructions and comparison of each result with the expected result.

```
; Module: BIT_TEST.ASM

; Version 1.00         BIA, sub-division 5.2                   Date: 15.09.2000

; Function:
; The subroutines test the BIT addressing in the bit-addressable memory range
; from 20H to 2FH (internal RAM). This accounts for 128 bit addresses. Owing to
; the time required for this test, eight addresses are selected deliberately for the
; test (walking 1 against 0 background in the bit address).
; Procedure:
; The memory range is first cleared.
; The selected bit is then set and tested.
; The remainder of the memory is then inspected to ensure that its content is correct.
; The set bit is then cleared again.
; (The bit commands SETB bit and CLR bit are tested at the same time during these operations).
; The commands for logical bit operations are tested in a further test.
```

```
; Input parameters: none
; Output parameters: none
; Changed registers: see individual functions
; Changed memory ranges: see individual functions

; Exit with error: branch to ERROR function

; Higher-level modules: LOGITEST.ASM
; Lower-level modules: none

; Register bank used: current bank
; ***************************************************************************



; ******************** BIT_TEST1 *******************************************
; Function: test bit address 01h

; Called by: BIT_TEST_A
; Calls: BIT_SPEICHER_RUECKL

; Changed registers: A, PSW
; Changed memory ranges: test bit (bit 01h)

; Stack depth: 2 bytes + called functions
; ***************************************************************************
BIT_TEST1:      MOV     BYTE_ADR,#20h           ; Preset flags for exclusion of byte addresses
                SETB    01h                     ; Sets test bit
                MOV     A,INTRAM20
                CJNE    A,#02h,Bit_err          ; Comparison with expected result
                LCALL   BIT_SPEICHER_RUECKL     ; Check remaining bit-addressable memory
                CLR     01h                     ; Clears test bit
                RET
```

Other bit tests and help functions: see source code of BIT_TEST.ASM

```
; ***************************************************************************
Bit_err:        MOV     FEHLER,#FEHLER_BIT      ; Load error memory
                LJMP    ERROR
; ***************************************************************************



; ******************** BIT_SPEICHER_RUECKL *********************************
; Function: examines the bit-addressable memory range (20h-2Fh) for correct
;           content, with the exception of the byte that has just been written to.
```

```
; Called by: BIT_TEST1 - BIT_TEST_8
; Calls: -

; Changed registers: A, R0, PSW
; Changed memory ranges: none

; Stack depth: 2 bytes
; ****************************************************************************
BIT_SPEICHER_RUECKL:
                MOV     R0,#20h                 ; Beginning of the bit-addressable range
comp2:          MOV     A,R0
                CJNE    A,BYTE_ADR,comp3        ; Position of the data byte which is =! 0
                INC     R0                      ; If yes, increment position
comp3:          CJNE    R0,#30h,comp4           ; End of the bit-addressable range?
                SJMP    comp5
comp4:          CJNE    @R0,#00h,BIT_err        ; Current byte==0 ?
                INC     R0                      ; Increment position
                SJMP    comp2
comp5:          RET


; ******************** BIT INSTRUCTIONS ******************************************
; Function: checks the bit-oriented instructions for correct operation

; Called by: BIT_TEST_D
; Calls: -

; Changed registers: PSW
; Changed memory ranges: bit address 01h

; Stack depth: 2 bytes
; ****************************************************************************
BIT BEFEHLE:    SETB    01h
                MOV     C,01h                   ; Transfer
                JNC     BIT_err                 ; Result OK?
                CPL     C
                ANL     C,01h
                JC      BIT_err                 ; Result OK?
                ORL     C,01h
                JNC     BIT_err                 ; Result OK?
                CLR     C
                JC      BIT_err                 ; Result OK?
                SETB    C
                CLR     01h
                MOV     01h,C                   ; Transfer
                JNB     01h,BIT_err             ; Result OK?
                ANL     C,/01h
                JC      BIT_err                 ; Result OK?
                ORL     C,/01h
```

```
                JC      BIT_err                 ; Result OK?
                CLR     01h
                JB      01h,BIT_err             ; Result OK?
                CPL     01h
                JNB     01h,BIT_err             ; Result OK?
                RET
```

3.2.5 Transfer instructions (TRANTEST.ASM)

In order for the transfer instructions to be tested, the individual instructions are executed with the different addressing variants, and the result compared with the expected result.

```
; Module: TRANTEST.ASM

; Version 1.00          BIA, sub-division 5.2                   Date: 15.09.2000

; Function:
; Test of transfer instructions: MOV, XCH, MOVX, MOVC

; Input parameters: none
; Output parameters: none
; Changed registers: see individual functions
; Changed memory ranges: none, except for memory cells specially reserved
; for these tests

; Exit with error: branch to ERROR function

; Higher-level modules: CPU_TEST.ASM
; Lower-level modules: none

; Stack depth: see individual functions
; Register bank used: current bank


; Note:
; The functions are to be called collectively from within the user program.
; ***************************************************************************




; ******************** MOV_TEST1 *******************************************
; Function: test of MOV A,R0

; Called by: MOV_TEST
```

```
; Calls: -

; Changed registers: A, R0, PSW

; Stack depth: 2 bytes
; ***************************************************************************
MOV_TEST1:      MOV     A,#55h                  ; Initialize accumulator
                MOV     R0,#0AAh                ; Load test pattern 1
                MOV     A,R0                    ; Instruction test
                CJNE    A,#0AAh,TRAN_err1       ; Comparison with expected result
                RET
```

Other tests: see source code of TRANTEST.ASM

4 Memory tests

4.1 Program memory test (ROM_TEST.ASM)

The program memory of a CPU is the component which determines the function to be performed by the system. A control processor for a video recorder can be converted into a control processor for a bucket loader merely by loading of a different software into the program memory.

For this reason, it is extremely important that the software be prevented from undergoing any undesired changes over the entire life of a system. This could occur even should only a single bit of the program memory change in value. Such a change may change the significance of an instruction such that the CPU then executes a completely different instruction at a certain point of the program. In a best-case scenario, this results in the processing "crashing". Undesired execution of certain functions, possibly hazardous, cannot be ruled out, however.

In order to prevent such changes within a memory cell from passing undetected, the content of the program memory is also checked continually by the CPU. This can be achieved by the formation of a signature, or checksum (in this case CRC 16), from the stored values [4]. This is then compared with a signature generated when the program memory is first programmed and then stored separately. Any change to the signature is detected, and the CPU is able to take measures by executing a previously defined error response, remaining in an endless loop, and thereby placing the system in the safe state.

In the test implemented here (Figures 3 and 4, see Pages 30 and 31), the program memory is divided into m segments. Each of these segments is integrated seamlessly into formation of the signature. Once a segment has been processed, the CPU is available for other tasks. The result is that the test is divided into manageable small time slices, and the processor is able to observe the response time required for execution of the control task.

Figure 3:
Segmentation of the ROM test

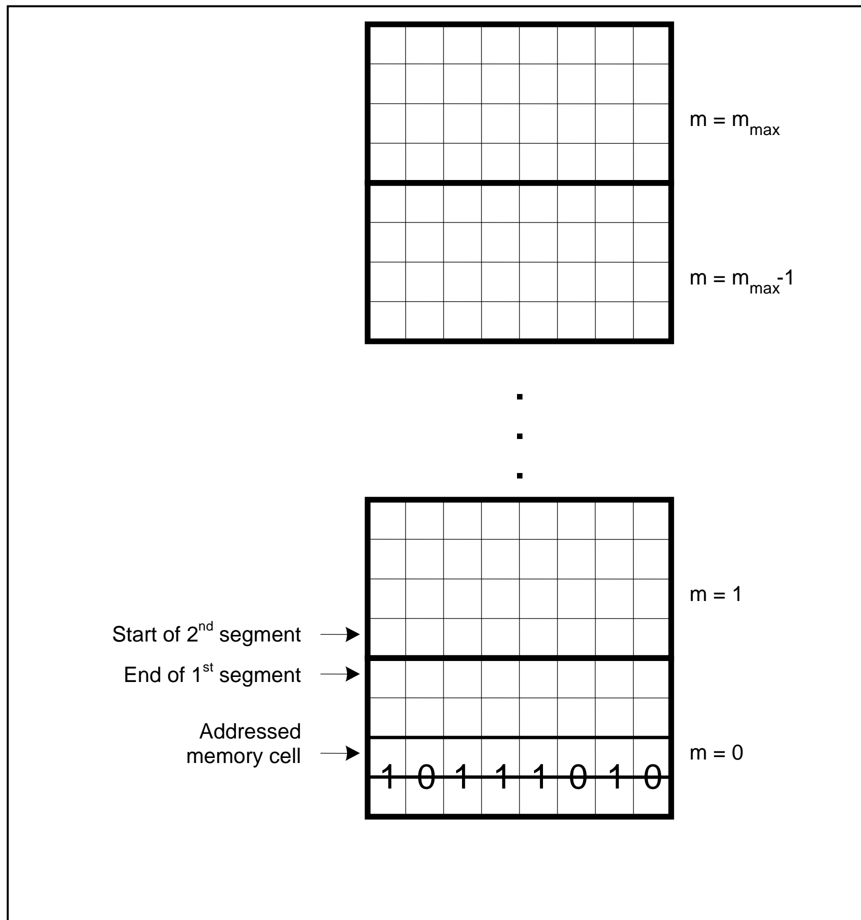


Figure 4:
Execution of the ROM test

ROM_TEST

Set address counter to segment start address

Use old signature as initial signature

Retrieve addressed byte from ROM

Compute new signature

Increment address counter

End of ROM reached?

Yes

No

Signature = desired signature?

No

Yes

End of segment reached?

No

Yes

Set segment start address and signature to zero

Exit with error

Assign new segment start address

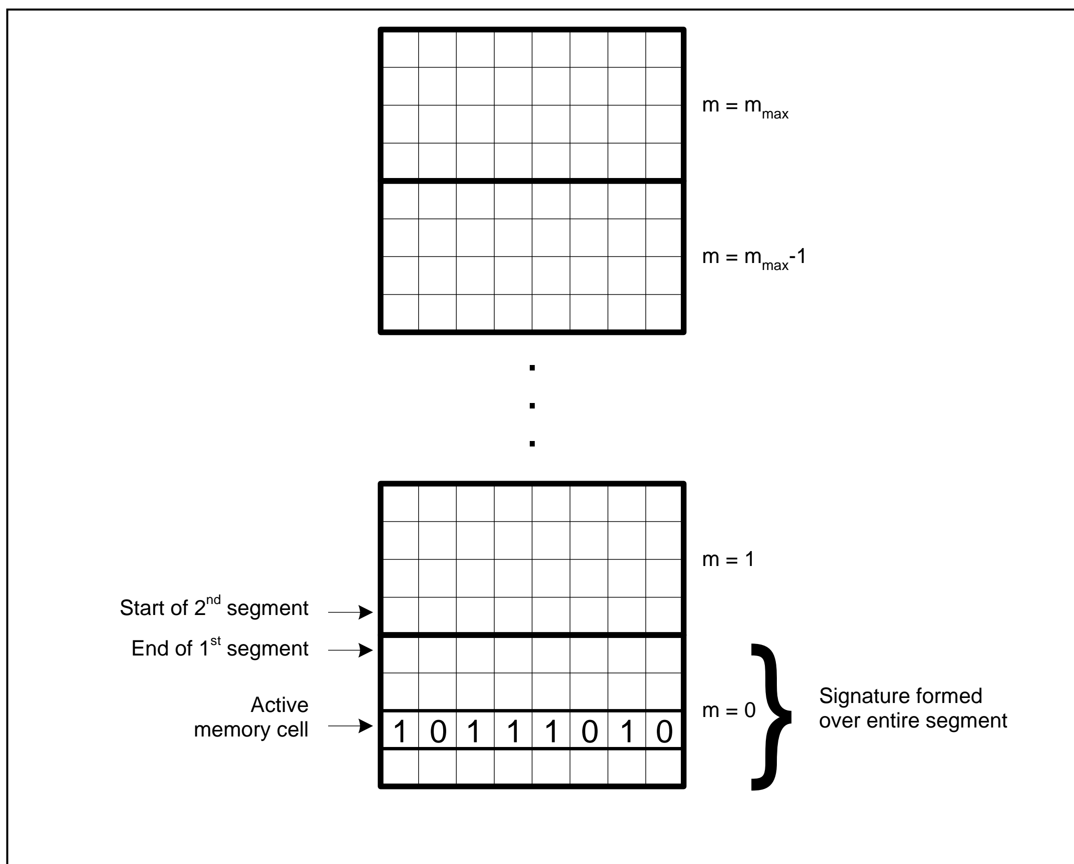
End of function

4.2 Data memory test (XRAMTEST.ASM)

In general, the points described for the program memory also apply to the data memory. Undesired changes to discrete data bits may also lead to an undesired and possibly also hazardous response on the part of the CPU. The data memory must therefore also be tested to ensure that it is operating properly and that, for example, any sticking of bits at a certain value or the short-circuiting of any given bits to form other random bits is detected, and a corresponding error response initiated.

As with the ROM test, segmentation (Figure 5) is performed to divide the test into time slices that are sufficiently small to assure the response time required by the CPU for the control task.

Figure 5:
Segmentation of the RAM test



A further distinction must be drawn here between the XRAMTEST1 function, which tests individual memory cells for internal faults, and the XRAMTEST2 function, which detects short-circuits between any given memory cells (Figures 6 and 7, see Page 34).

Figure 6:
RAM test 1 procedure

XRAM_TEST1

First memory segment selected?

No

Yes

Save content of active cell

Push walking “1” through active cell

Defective value in active cell?

Yes

No

Exit with error

Restore original value in active cell

End of function

This procedure is iterated 8x until the walking “1” has passes through the cell

Figure 7:
RAM test 2 procedure

XRAM_TEST2
Save content of active cell
Form signature over segment
Save test value 00h in active cell
Form signature over segment
Do signatures formed match?
No
Yes
Save test value FFh in active cell
Form signature over segment and select next segment
Do signatures formed match?
No
Yes
Exit with error
Restore original values in active cell
Segment address at end of memory?
No
Yes
Increment address of active cell
Active cell at end of memory?
Yes
No
Set address of active memory cell to start of memory
Set segment address to start of memory
End of function

In order for short-circuits between any memory cells to be detected, each byte of the entire memory to be tested is tested against the rest of the memory. For this purpose, various different values are written to the byte to be tested, and the remaining memory

then checked for changes. This test is performed by generation of a signature: consequently, a difference between the values of the signature before and after writing to the byte to be tested indicates the presence of a short-circuit between memory cells. In order for the time slice required for the RAM test to be kept small, the RAM is divided into m segments as described above, and a signature generated only over one segment at a time. This ensures that the CPU is not occupied for a longer period solely with the test, thereby rendering prompt execution of the user program impossible for lack of processing time.

The file XRAMTEST.ASM contains further information concerning the RAM test.

5 Special function register test (SFR_TEST.ASM)

In order for the internal special function registers of the microcontroller to be tested, they too are written to individually with values after their content has been stored, in the same way as other memory cells. A signature is formed for the remaining cells, i.e. those not being tested. Once the cells under test have been written to, a signature is again generated over the remaining memory cells, in order to detect whether crosstalk with other cells has occurred.

It must be noted here that the writing of some special function registers triggers functions of certain peripheral controller components (such as writing to the send register of the serial interface). Consequently, some of these registers cannot be tested by means of this method, and other test methods may have to be employed.

A particular aspect of the test implemented in this example is that the content of the special function registers under test is first copied into a range of the external data memory; generation of the signature is simpler there, since the special function registers cannot be addressed indirectly.

6 Port tests (IO_TEST.ASM)

Figure 8 contains a schematic representation of a possible port interface by which the CPU can be enabled to read back the instantaneous state of the output peripherals. Correct functioning of the transistor can be tested at any time with the collector-emitter path turned on, by removal of the drive signal from the CPU for between a few hundred microseconds and a few milliseconds (as a function of the filter action of the external circuit) and reading back, via the port input, of proper disconnection. This tests whether the processor would still be able to de-activate the relay should this be necessary. The duration of removal of the drive signal must be selected such that the relay is unable to drop out and the machine control would be influenced by the test measure.

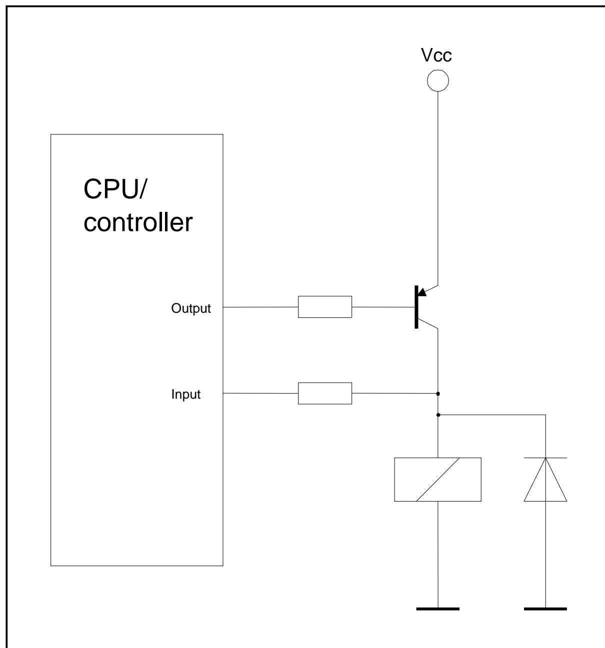


Figure 8:
Port interface arrangement
with readback

In order to make the port output used to drive the relay partly independent of the port input used for readback, different ports should be used (e.g. port 1.5 for the output bit, port 3.5 for the input bit).

Besides the test described here, fault detection must be performed for the actuator itself, for example the relay shown in Figure 8. For this purpose, the switching state of a break contact on the mechanically linked output relay is generally read back in through an input port. In this test, it is essential that a request for the relay to switch be executed by the process. Should the relay be unable to drop out owing to internal

faults, this is detected by the feedback of the mechanically linked contact, and the second de-energization procedure involving the watchdog must initiate the safe state. Should the fault remain present, re-energization of the control system is then reliably prevented.

In order for these diagnostics features to be available, both the hardware and the software must satisfy the necessary criteria. The measures for testing of the output ports on the hardware side are shown in Figure 8. On the software side, a function must be created which tests whether the controller output is still capable of switching. The function implemented here (Figure 9) queries and stores the status of the outputs.

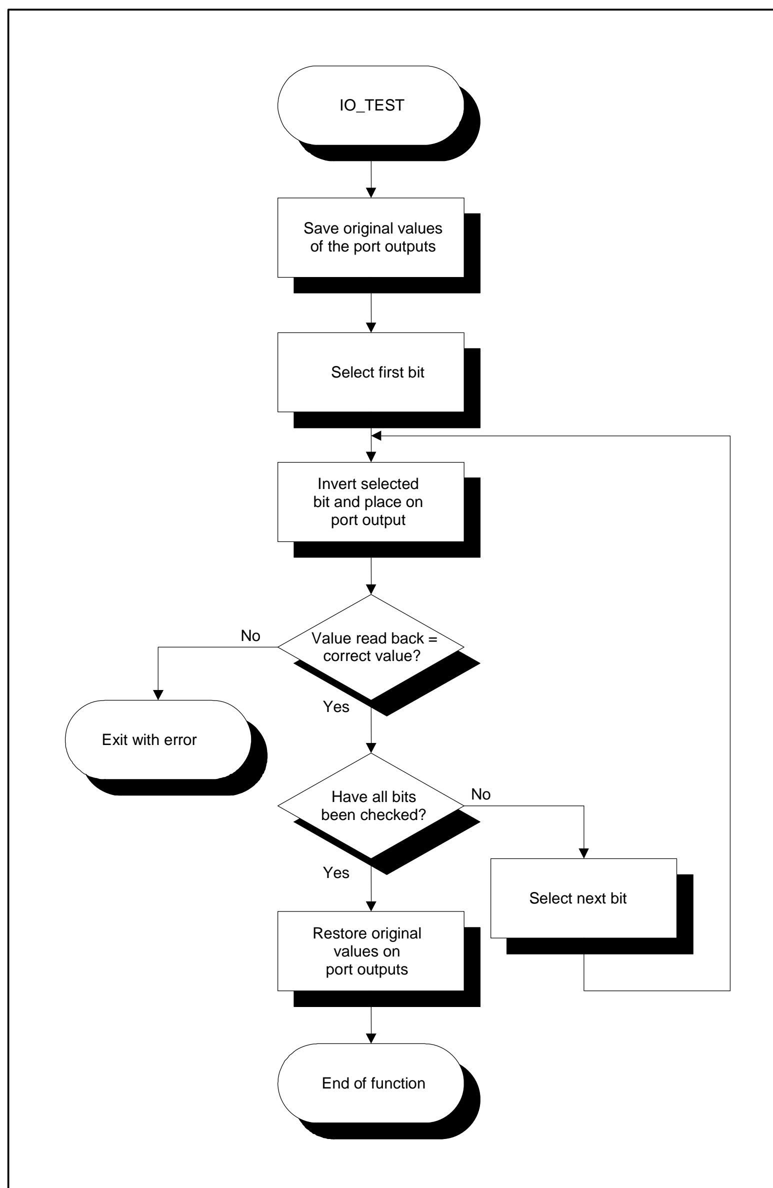


Figure 9:
Port test procedure

Each bit of the port is then inverted in turn[1], and the expected bit pattern read in and compared with the output bit pattern. Any static states of outputs and inputs on the ports concerned are thus detected.

[1] Note: in this procedure, it is essential that a "1", even if only transient, should not be allowed to cause actuation of the external peripherals.

7 Main program

Figure 10 shows a schematic possible arrangement of the main program for incorporation of both the self-tests and the user program.

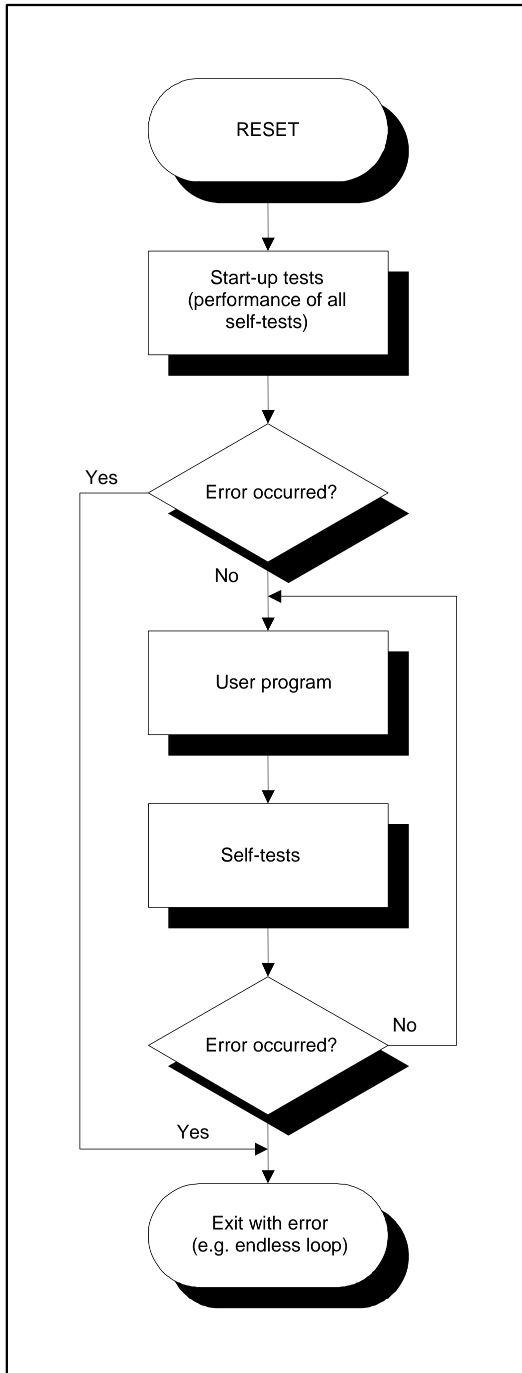


Figure 10:
Procedure for the main program

It is important here that following energization of the microprocessor system and before the user program is executed for the first time, all tests are executed completely

once. This start-up test ensures that only a correctly functioning microprocessor system is placed in service. Following execution of all tests (CPU_TEST.ASM), the continually executed self-tests can be divided into time slices for execution.

The main program will be programmed completely differently according to the intended application of the microcontroller. Both procedure-driven and event-driven programming are conceivable for this purpose; no specification can therefore be made at this point of a suitable test manager.

8 Concluding remarks

This report is intended to illustrate that there is no single “right way”. The examples shown are intended as suggestions only and to facilitate access to the subject-matter. For further considerations, up to and including a complete safe system, it must be appreciated that the basis for such a system lies in the safety-related structure. The microprocessor tests provided in this report are intended to complement such a system and to prevent the accumulation of faults. The response to detected faults may be adapted to the circumstances of a particular system. Should issues require clarification during implementation, the BGIA can provide appropriate advice.

“Quo vadis, fault?” was the question posed in the sub-title. All faults detected by the tests are passed to the watchdog, as a result of which – indirectly, by the creation of an endless loop in the test program being executed – they fail to supply trigger signals to the watchdog, and thus bring about the safe state by initiation of the second de-energization procedure. This is one of the possible structures which, within a package of further measures, is suitable for controlling the impact of random failures in machine safeguarding.

9 References

[1] *Hauke, M.; Schaefer, M.; Apfeld, R.; Boemer, T.; Huelke, M.* et al.: Functional safety of machine controls – Application of EN ISO 13849. BGIA Report 2/2008e. Ed.: BGIA – Institute for Occupational Safety and Health of the German Social Accident Insurance, Sankt Augustin 2009. www.dguv.de/bgia, Webcode e91335 (in preparation)

[2] IEC 61508/VDE 0803: Functional safety of electrical/electronic/programmable electronic safety-related systems. Beuth, Berlin 2002

[3] *Klug, J.; Schaefer, M.*: Fehlererkennende Maßnahmen in Mikroprozessoren. Erich Schmidt, Berlin 1997

[4] *Leisengang, D.*: Klassifikation und Einsatz von Signaturregistern zur Fehler-erkennung in digitalen Schaltungen. Dissertation, Technische Universität München, 1982

More detailed literature:

Halang, W. A.; Konakovsky, R.: Sicherheitsgerichtete Echtzeitsysteme. Oldenbourg, Munich 1999

Leisengang, D.: Signaturanalyse in der Datenverarbeitung. Elektronik 21 (1983), pp. 67-72

Hölscher, H.; Rader, J.: Mikrocomputer in der Sicherheitstechnik. Verlag TÜV Rheinland, 1984

Schaefer, M.; Gnedina, A.; Bömer, T.; Büllesbach, K.-H.; Grigulewitsch, W.; Reuss, G.; Reinert, D.: Programmierregeln für die Erstellung von Software für Steuerungen mit Sicherheitsaufgaben. Verlag für neue Wissenschaft, Bremerhaven 1998

Siemens Microcomputer Components, SAB 80C517/80C537, 8-Bit CMOS Single-Chip Microcontroller. Siemens, Munich 1983