ETFA 2007

12th IEEE International Conference on Emerging Technologies and Factory Automation

September 25-28 2007, Patras, Greece

www.etfa2007.org

Welcome
Committees
Table of Contents
Keynotes
Technical Program
Author Index
Help
Search

IEEE

I.S.I.
Industrial Systems Institute

©2007 IEEE IEEE Catalog Number: 07TH8932C ISBN: 1-4244-0826-1 Library of Congress: 2006937986

TECHNICAL PROGRAM

Session: T1.1 Room: I4 *Wednesday, Sep. 26,* 11:30 - 13:00

Communication in Automation Systems: Possibilities and Limitations

Chairing: Juergen Jasperneite, Alexander Fay

Session: T6.1 Room: I 10 *Wednesday, Sep. 26,* 11:30 - 13:00

Embedded Model Control and Emerging Ttechnologies

Chairing: E. Canuto, J. Ospina

Session: T3.1 Room: I 11 *Wednesday, Sep. 26,* 11:30 - 13:00

Scheduling and Resource Management

Chairing: Bjorn Anderson, Luigi Sassoli

Session: SS2.1 Room: I 12 *Wednesday, Sep. 26,* 11:30 - 13:00

Emerging Technologies to Enable Integrated Manufacturing and Service Systems (IMSS)

Chairing: Goh Kiah Mok, Cristian Vasar

Session: T2.1 Room: I4 *Wednesday, Sep. 26,* 14:30 - 16:00

Wireless Industrial Communications

Chairing: Christos Koulamas, Julian Proenza

Session: T8 Room: I 10 *Wednesday, Sep. 26,* 14:30 - 16:00

Computational Intelligence in Automation

Chairing: E. Man, J. Tar

Session: T5.1 **Room: I 11** ***Wednesday, Sep. 26,*** **14:30 - 16:00**

Architectures, Methods and Technologies for Enterprise Integration

Chairing: Rei Itsuki, Jose Lastra

Session: SS2.2 **Room: I 12** ***Wednesday, Sep. 26,*** **14:30 - 16:00**

Planning and Integration Technologies for Manufacturing and Service Systems

Chairing: Angle Goh, He Wei

Session: T1.2 **Room: I4** ***Wednesday, Sep. 26,*** **16:30 - 18:00**

IT in the Design Process of Automation Systems

Chairing: Alexander Fay, Juergen Jasperneite

Session: T6.2 Room: I 10 *Wednesday, Sep. 26,* 16:30 - 18:00

Industry/Bank Automation

Chairing: R. Vilanova, F. Koumboulis

Session: T4 Room: I 11 *Wednesday, Sep. 26,* 16:30 - 18:00

Intelligent Sensors and Sensor Networks

Chairing: Pedro M. Ruiz Martinez, Paul M. Havinga

Session: SS1.1 Room: I 12 *Wednesday, Sep. 26,* 16:30 - 18:00

The IEC 61499 Function Block Model in Control and Automation

Chairing: Kleanthis Thramboulidis, Georg Frey

Session: SS3 Room: I4 *Thursday, Sep. 27,* 11:00 - 12:30

Methods and Instrumentation for Performance Measurement in Real-time Networks

Chairing: Alessandra Flammini, José A. Fonseca

Session: T6.3 Room: I 10 *Thursday, Sep. 27,* 11:00 - 12:30

Control Theory and Applications

Chairing: A. Tzes, L. Massotti

Session: T5.2 Room: I 11 *Thursday, Sep. 27,* 11:00 - 12:30

Emerging Issues and Solutions

Chairing: Masanori Akiyoshi, Jose Lastra

Session: SS5 Room: I 12 *Thursday, Sep. 27,* 11:00 - 12:30

Embedded Systems Security

Chairing: D.N. Serpanos, W.H. Wolf

Session: T7 Room: I 13 *Thursday, Sep. 27,* 11:00 - 12:30

Distributed Intelligent Control for Flexible Manufacturing

Chairing: George Chryssolouris, Nidhal Rezg

Session: WIP 1 Room: I4 *Thursday, Sep. 27,* 16:00 - 17:00

Industrial Networks and Factory Automation

Chairing: Thilo Sauter

Session: WIP 2 Room: I 10 *Thursday, Sep. 27,* 16:00 - 17:00

Embedded Systems and Tools

Chairing: Thilo Sauter

Session: SS9 Room: I 11 *Thursday, Sep. 27,* 16:00 - 17:30

Business Intelligence and its Applications in Industrial Ecosystems

Chairing: Elizabeth Chang, Tharam Dillon

Session: SS6.1 Room: I 12 *Thursday, Sep. 27,* 16:00 - 17:30

Interoperability Issues and Research Work in Progress

Chairing: Vincent Chapurlat, Athanasios Kalogeras

Session: WIP 3 Room: I 13 *Thursday, Sep. 27,* 16:00 - 17:00

Sensors and Actuators

Chairing: Thilo Sauter

Session: T3.2 Room: I 10 *Thursday, Sep. 27,* 17:00 - 18:00

Real-Time and Control

Chairing: José A. Fonseca

Session: T9.1 Room: I4 *Friday, Sep. 28,* 10:00 - 11:30

Intelligent Robots I

Chairing: Josep M. Mirats, Yolanda Bolea

Session: T5.3 Room: I 11 *Friday, Sep. 28,* 10:00 - 11:30

Automated Manufacturing Systems

Chairing: Toshiya Kaihara, Jose Lastra

Session: SS8 Room: I 12 *Friday, Sep. 28,* 10:00 - 11:30

Design and Analysis of Distributed Automation Systems

Chairing: Georg Frey, Alexander Fay

Session: T3.3 Room: I 13 *Friday, Sep. 28,* 10:00 - 11:30

Distributed Real-time Systems

Chairing: Thomas Nolte, Orazio Mirabella

Session: T10 Room: I4 *Friday, Sep. 28,* 12:00 - 13:30

Emerging Issues

Chairing: Gianluca Cena, Dacfey Dzung

Session: SS4 Room: I 10 *Friday, Sep. 28,* 12:00 - 13:00

Innovative E-Learning Experiences

Chairing: Luis Gomes, Orazio Mirabella

Session: SS7 Room: I 10 *Friday, Sep. 28,* 13:00 - 13:30

Grouping and Cooperating of Services

Chairing: Carsten Buschmann, Reinhardt Karnapke

Session: T5.4 Room: I 11 *Friday, Sep. 28,* 12:00 - 13:30

Multi-agent Systems for Manufacturing Control

Chairing: Masanori Akiyoshi, Jose Lastra

Session: T2.2 Room: I 12 *Friday, Sep. 28,* 12:00 - 13:30

Scheduling, Safety and Response Times of Industrial Communication Networks

Chairing: Julian Proenza, Thomas Nolte

Session: SS6.2 Room: I 13 *Friday, Sep. 28,* 12:00 - 13:30

Interoperability Applications

Chairing: Athanasios Kalogeras, Ioannis Gialelis

Session: T9.2 Room: I4 *Friday, Sep. 28,* 13:30 - 15:00

Intelligent Robots II

Chairing: Antoni Grau, Gabriel J. Garcia

Session: WIP 4 Room: I 10 *Friday, Sep. 28,* 13:30 - 14:30

Wireless and Dependable Networks

Chairing: Thilo Sauter

Session: WIP 5 Room: I 11 *Friday, Sep. 28,* 13:30 - 14:30

Control

Chairing: Thilo Sauter

Session: T2.3 Room: I 12 *Friday, Sep. 28,* 13:30 - 14:30

Clock Synchronization and Multimedia Real-time Communications

Chairing: Thomas Nolte, Christos Koulamas

Session: SS1.2 Room: I 13 *Friday, Sep. 28,* 13:30 - 15:00

IEC61499 Implementations

Chairing: Kleanthis Thramboulidis, Georg Frey

Testing Approach for Online Hardware Self Tests in Embedded Safety Related Systems

Thomas Tamandl, Peter Preininger, Thomas Novak, Peter Palensky

Vienna University of Technology,
Institute of Computer Technology
Gusshausstrasse 27-29
1040 Vienna, Austria
{tamandl, preininger, novakt, palensky}@ict.tuwien.ac.at

Abstract

In safety related systems online hardware self tests are integrated to reach a defined level of hardware integrity. These tests comprise testing of the static and volatile memory and the CPU internals. The higher the level of integrity should be, the more efficient tests must be used. Additionally, the effort to verify the correctness of the tests is rising. Hence, designing the test, and the validation and verification of the tests, is a critical part in the development process of safety related systems.

The verification of the correct behavior requires sophisticated methods for stimulating errors that must be detected by the tests. The document describes an environment based on boundary scan technology for testing the online hardware self tests automatically.

1. Introduction

Today's functional safety related systems (short safety related systems) are mostly accomplished with microcontrollers. They are created to reduce the inherent risk of a device or system, for example a node in a fieldbus system, to a tolerable level.

The international standard IEC 61508 [1] defines requirements for designing safety related systems. It specifies a life cycle model including all activities required to avoid systematic failures and to handle stochastic failures.

IEC 61508 defines functional safety as "part of the overall safety that depends on a system or equipment operating correctly in response to its inputs" [1]. A safety related system is a system that executes safety functions and cares for the required safety integrity of the safety functions. A safety function is responsible for reaching or keeping a safe state of a device. The safety integrity of a safety related system includes the hardware integrity and the systematic integrity.

Systematic integrity means applying measures to reduce the risk coming from systematic failure during the design or operation phase of the system. Typical measures are on the one hand management tools and on the other hand functions to monitor the program flow or the supply voltage. To grant a designated level of hardware integrity a defined amount of stochastic hardware failures must be detected.

A method to detect stochastic hardware failures is to test the hardware components with online hardware self tests (short hardware tests) implemented in software. Normally hardware tests are periodically inspecting the volatile and non volatile memory as well as the central processing unit (CPU) including its registers, flags and arithmetic logic unit (ALU) for hardware failures.

There are different hardware test algorithms available capable of checking the volatile memory and registers of the CPU. Examples for RAM test methods are the walking pattern test [2], Abraham test [3] or galloping pattern test [2]. Classically static or invariant memory is tested for failures by means of parity bits, checksums or cyclic redundancy checks (CRC). Flags and the ALU are tested by inserting bit patterns and comparing the output with the specification [2]. The algorithms and methods to ensure hardware integrity differ in performance and diagnostic coverage (DC) (= ratio of detected failures to the total amount of failures).

The safety integrity of hardware is categorized by four safety integrity levels (SIL) [1]. Each SIL specifies an error probability per hour, i.e. the tolerable number of dangerous errors per hour. SIL 1 defines the highest value of error probability, whereas SIL 4 specifies the lowest level. The higher the SIL the more rigorous are the safety integrity requirements. A way to meet these requirements is to implement hardware tests with a high diagnostic coverage.

Unfortunately, complete avoidance of human mistakes during implementation of hardware tests is not realistic. For example, even a well-trained and experienced programmer makes an error in every 100 statements [4]. Thus the standard IEC 61508 specifies a great amount of requirements for the testing process too. These requirements shall ensure a high level of software

1-4244-0826-1/07/$20.00 © 2007 IEEE

quality and therefore reduce the probability of malfunctions.

2. Software Testing

Generally, software can be tested code-based or requirements-based. Test cases derived from the source code are white box test cases, others derived from the software requirements are black box test cases. Black box test cases are specified using methods such as equivalence class partitioning or boundary value analysis [5]. On the contrary, white box test cases are defined depending on the code coverage, e.g. statement, branch or path coverage. The coverage is always the fraction of total number of statements, branches or paths having been executed by different test cases.

Test cases are used to perform different types of testing such as function testing (verification of functionality) or performance testing. These test types are carried out at various steps of the software testing process.

The first step of software testing is unit testing. The software unit, e.g. a function, is examined if it can perform its specified functionality properly. This type of testing is described here. The second step is integration testing where the units are integrated to form a subsystem. At this point attention is paid to the interfaces between the units. The final step is system testing where the software has to run on the target platform and interacts with external software.

3. Overview of Test Approach

As already mentioned in section 1 safety integrity is specified by 4 safety integrity levels (SIL). A higher SIL not only results in more rigorous safety integrity requirements, but also in a higher testing effort. The higher the SIL the more comprehensive test cases with white and black box test cases have to be performed on the software. Thus automatic tests tools are desirable to keep the test effort moderate.

The test approach for hardware tests presented in the following sections allows checking the functionality of the hardware tests automatically. It enables the tester to execute white box and black box test cases during the unit, integration and system testing.

The test approach was elaborated during the development process of a safety related system in the project SafetyLon. The European collective research project SafetyLon supported by the European Union within the sixth framework program has the goal to make the LON technology [6] safe. It is foreseen to meet safety requirements as CANopen-safety [7] or PROFISafe [8] does.

Within the project standard LON nodes are enhanced with additional safety related hardware and special embedded safe software. Parts of the safe software comprise the hardware tests for the central processing unit, the volatile- and non volatile memory. The correct behavior of these must be verified. As a result testing techniques to verify the hardware test were developed that are presented in the following.

4. Technical Overview

Hardware tests must be performed to guarantee the correct functionality of the system. The implementation of these tests is described in [9]. In the following the methodology how to verify the behavior of these tests is introduced.

For hardware test verification fault injection technologies are necessary to insert faults that must be detected by the hardware tests. Standard fault injection technologies use direct pin level injection, additional software modules, and external electromagnetic fields or similar to provoke a fault. [10] provides a good overview on some technologies already used.

The microcontrollers used to run the hardware tests are based on a standard ARM7TDMI core with internal RAM and FLASH memory. As all components are located within one package, there are no external data- or address-busses. Due to the lack of accessibility of the internal data busses direct pin level fault injection is impossible. As there was the requirement not to change the software running on the target hardware, software fault injection, regardless of runtime or compile time injection could not be chosen. Hence, approaches chosen from [11], [12] or [13] are not suitable.

The problem, how to manipulate data without any access to internal signals, can be addressed by means of debugging features offered by the ARM core. [14] discusses pin level fault injection in opposite to using fault injection boundary scan. This technology, enhanced with the internal scan chains offered by the ARM7 controller, allows the simulation of internal faults without changing the software running on the target hardware.

For debugging purposes the ARM7 core provides a Joint Test Action Group (JTAG) conform debugging interface. The international standard IEEE 1149.1 [15] Standard Test Access Port and Boundary-Scan Architecture describes this interface that was initially intended for testing printed circuits. In case of the ARM7 core it grants the access to the core internals via the test access port (TAP) controller. Hence mostly all processor internals are accessible from the outside. The JTAG interface allows controlling the ARM7 core from outside, Figure 1 shows a typical debugging environment.

A debug host computer using a high level debugging language is communicating with the debug target. Communication is handled a protocol converter which

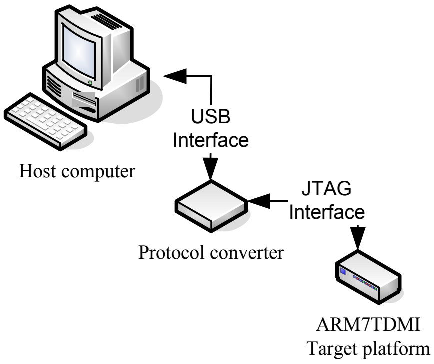

Figure 1. Typical debug environment

converts the high level language into the JTAG compliant commands.

As already mentioned the debugger on the host computer is communicating with the TAP controller. This device is controlling three internal scan chains connected to the main processor logic and the EmbeddedICE-RT logic described later, see Figure 2 for details. Scan chain 0 and 1 interfaces the processor core and the data bus and scan chain 2 grants the access to the EmbeddedICE-RT registers. Using this scan chains all processor internals can be accessed, examined and modified.

The ARM7 core itself offers the following debug features. The program execution can be halted, the state of the system can be examined, as described before, and the program execution can be resumed.

The EmbeddedICE-RT logic provides additional features. It can be accessed and configured by means of scan chain 2.

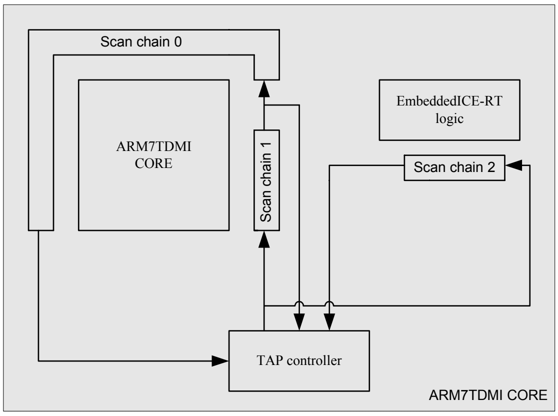

Figure 2. ARM7TDMI scan chains [16]

The EmbeddedICE-RT logic monitors processor internal signals such as data- and address-bus. Beside others this logic provides two configurable break or watchpoints. The breakpoints can be configured to force a halt of the processor if the appropriate address is reached. Watchpoints can be configured to halt the processor when specified memory cells are addressed. Refer to [16] for details on the ARM7TDMI internals.

The aforementioned features permit to halt the processor without having direct access to the signals of the core logic when accessing specified memory addresses. In halt mode modifications can be done and the program can be resumed. These features allow automatic testing of the hardware tests presented in [9].

For setting up an automatic test environment a high level language is useful for the communication with the TAP controller. Therefore a programming interface was developed by the company Segger Microcontroller Systems [17]. Its purpose is to provide third party applications the whole functionality of the J-Link interface.

Segger Microcontroller Systeme GmbH offers a set of tools for developing applications running on ARM7 cores. A protocol converter named J-Link – see Figure 1 – is used to connect the host PC via USB to the JTAG port of the processor. Normally a debugger is communicating with the J-Link. For automatic fault injection a debugger is not suitable so an adequate application was developed. Therefore direct control – without using a debugger – over the J-Link interface is required.

The J-Link Software Development Kit, including the J-Link application program interface, is satisfying these requirements. The communication itself is performed like a standard communication resource. After opening and configuring the JTAG connection the ARM7 core can be controlled by running the appropriate commands. The automatic test tool described in the next chapter is based on the J-Link application program interface.

5. Test Tool Design Basics

Based on the information provided in the previous chapter the J-Link application program interface can be used to set up an automated test tool. In the following the principles the automated test tool is based on are explained.

5.1. Testing Principles

A test tool for safety related systems has to be able to perform a huge number of different test cases. The best solution is a tool which can perform the tests automatically. Such a tool needs information on how to perform a test and how a test is considered as successful or failed. Therefore it must be able to accept defined test cases which are stored in an input file. Details on the input file are presented in section 6.3. The necessary functions of a software test tool are on the whole similar to debugging functions. The most important representatives are:

- Reset ARM7 core
- Start and halt ARM7 core

- Set and reset breakpoint
- Single stepping
- Manipulation of memory contents
- Read and store memory contents
- Manipulation of registers
- Read registers contents
- Manipulation of register flags

In order to set up an automatic test environment additionally the following functions are necessary:

- Read and compare memory contents to specifiable values
- Read and compare register contents to specifiable values
- Read and compare flag values to specifiable contents
- Specification of test case fail/success depending on the result of the comparisons

Beside the functionality of an automated test tool, the ability to generate reports must be taken into account. In order to detect erroneous memory contents, functions for comparing memory cells or registers to specific values are available. The result of the comparison can be used to define the failing or passing of a test case.

5.2. White Box Testing

The presented functions allow performing white box tests for hardware tests. Obviously performing white box test cases is only possible on assembler level. If C-code has to be tested, this can only be done with the underlying compiler generated assembler code.

Figure 3 shows the standard program flow of the test tool. The flow does not include the more complicated black box testing implementations. Line per line of the input file is interpreted by the JLink interface. According to the interpretation the appropriate JLink operation is executed and logged to a report file. If the operation sets the ARM7 core into a run mode, the program waits until the ARM7 core is halted due to reaching a breakpoint or watchpoint. As soon as the ARM7 core is halted a log line is written and the results of the operation are evaluated and logged (if applicable). If the operation is a test operation, the tool evaluates whether the test is considered to be successful. If the test is successful, the complete process is executed again until either a failed test occurred or a test is considered as failed. If a test is considered as failed the program is stopped.

White box tests require the user to alter register and memory contents in order to guide the program into the different program paths. For doing that, the user can run the program to a certain point specified by a breakpoint and modify the memory or register contents (and therefore change the path of program execution). A second breakpoint can be defined for testing if the path had been executed properly. E.g., the user can query if a memory cell or a register contains the expected results. A wrong value can cause the test to fail.

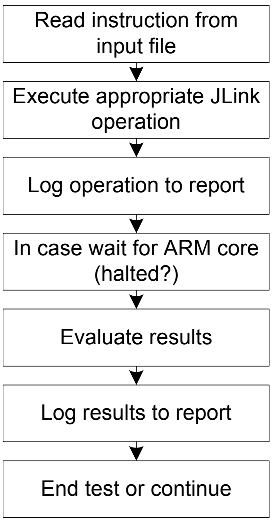
Read instruction from input file
Execute appropriate JLink operation
Log operation to report
In case wait for ARM core (halted?)
Evaluate results
Log results to report
End test or continue

Figure 3. Standard program flow of test tool

All these actions are logged to the report in order to keep track of the program proceedings. In case of a failure, the logged data can be used to check if the test itself had been set up wrongly or the program produced a failure.

It is the main task of the tester to define tests which cover all possible program paths in order to detect possible errors in the program flow.

5.3. Black Box Testing

For testing the hardware tests of volatile or non volatile memory black box testing methods are required. In opposite to white box testing these kinds of tests are independent from the programming language used for designing the hardware test.

The test of the hardware tests are based on simulating stuck at faults and coupling faults. A stuck-at fault means that a memory cell delivers only a 1 or 0 when accessed. The value is independent of the content that the cell should have. A coupling fault on the other hand concerns two different memory cells. One memory cell, which is accessed, causes the other memory cell to be influenced. This behavior is called a coupling fault. It can depend on the access mode (read or write) of the influencing cell as well as on the contents of the memory cells.

For the simulation it is necessary to specify which bits of a memory cell are stuck-at 1 and which ones are stuck-at 0 (or coupled to 1 or 0).

The simulation of both faults uses the watchpoint functionality offered by the EmbeddedICE-RT unit of the ARM7 core. This feature halts the ARM7 core as soon as the specified memory address is accessed. The possibilities for the watchpoint configuration are numerous. Read/write access can be specified as well as the halt on one specific memory address or a masked memory address. In order to keep the tool user-friendly only one specific memory address can be specified for the watchpoints.

The difficulty of using watchpoints is that the ARM7 core is halted after an instruction occurred. This means that in case of a write access the ARM7 core is halted after the memory cell has been written – in case of a read access the registers already contain the appropriate values.

In order to be able to simulate a stuck-at or coupling failure the previously executed instruction has to be identified and evaluated. For doing this the program counter is read via the JLink and the instruction at the appropriate address is read.

In case of a write to memory instruction the simulation is straight forward. The appropriate memory cell is modified. After performing a read instruction the register which contains the memory data has to be identified. This can be more complicated in the case of a multiple read access.

In fact the difficulty is that the executed instruction which led to the halt has to be identified, translated and the effects of the instruction on the memory cells and CPU registers has to be made undone.

As an example Figure 4 shows the implementation of the stuck-at simulation. The implementation is only shown for a single run, after the execution of the run, the ARM7 core is restarted. As shown in the diagram the OP_Code of the instruction which led to the watchpoint is evaluated using a case instruction. The appropriate concerned register is extracted out of the opcode.

The simulation of the coupling failures is very similar to the implementation presented in Figure 4. The only difference is that the watchpoint is set to the original memory cell (which changes the coupled cell if accessed). The main difficulty is the behavior of the multiple load and store commands. If both – the original and the coupled cell – are accessed within a multiple load or store command this issue has to be taken into account.

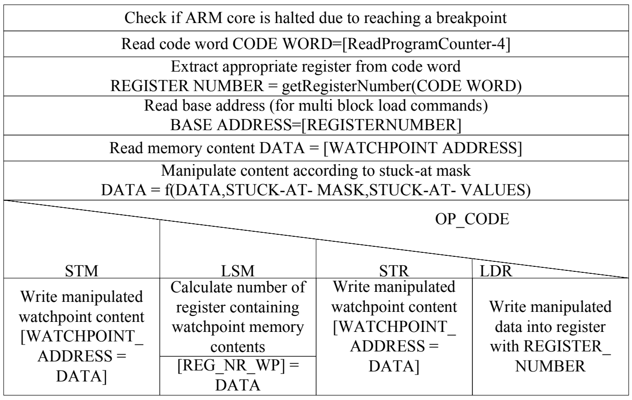
Check if ARM core is halted due to reaching a breakpoint
Read code word CODE WORD=[ReadProgramCounter-4]
Extract appropriate register from code word
REGISTER NUMBER = getRegisterNumber(CODE WORD)
Read base address (for multi block load commands)
BASE ADDRESS=[REGISTERNUMBER]
Read memory content DATA = [WATCHPOINT ADDRESS]
Manipulate content according to stuck-at mask
DATA = f(DATA,STUCK-AT- MASK,STUCK-AT- VALUES)
OP_CODE
STM
LSM
STR
LDR
Write manipulated watchpoint content [WATCHPOINT_ ADDRESS = DATA]
Calculate number of register containing watchpoint memory contents
[REG_NR_WP] = DATA
Write manipulated watchpoint content [WATCHPOINT_ ADDRESS = DATA]
Write manipulated data into register with REGISTER_ NUMBER

Figure 4. Nassi-Schneidermann diagram of simulation of a stuck at failure

Based on the two failure models, the stuck-at and the coupling fault model, the correct behavior of the hardware tests for volatile and non volatile memory can be verified. Also the performance of the tests, regarding their capability of detecting different failures can be tested.

6. Performing Tests

As presented in section 5 there are a lot of possible configurations for the automated test tool. For performing tests on hardware tests only a few instructions are necessary to configure the test tool according to the specified test. In the following the test tool is presented on a more detailed level. Later, on behalf of a white box test case for a test of the arithmetic logic unit, it is shown exemplarily how to set up a concrete test. Following this model all test mentioned above can be implemented.

6.1. Tool Configuration

At the beginning an input file, designed according to the comma-separated value (csv) file format, is defined. The csv-format is used for storing tabular data, where the columns are separated by a delimiter. It allows editing the file with a simple text editor or a spreadsheet calculator. Hence, simple test cases can be designed very quickly.

In order to gain the required breakpoints the map files generated by the linker have to be examined. These map files contain the information on the addresses which are used for setting the breakpoints. Every time the source code or the placement of the object file changes, it is likely that the addresses of the breakpoints are changed too. Depending on the changes done, a spreadsheet calculator may help to address the problem of changing addresses. E.g. the addresses of the breakpoints then can be defined as the sum of the module address and an offset. Hence, as long as the module internals do not change, only the module addresses must be updated. This does not help if the tested module itself is changed. Of course the specification of the test procedure and its breakpoints has to be done very carefully, as the number of tests usually is rather numerous (for example, there are specified more than 270 test cases to test the online RAM test).

To avoid failures during the creation process of the input file a very clearly arranged format is used. First a command is given specifying the appropriate action. The other values are depending on the specified command. The test tool simply interprets the command together with the specified data and performs the appropriate action of the Segger interface. Also comments can be specified in the input file in order to make it maintainable more easily. Lines within the input file are considered as comments which are logged directly to the report file. Additionally a line number referring to the input line number is written into the output file.

Table 1 shows the instructions which can be used within the csv input file.

Table 1. Instruction set of the test interface

Instruction	Description
RESET	Reset ARM7 core
HALT	Stop ARM7 core
RUN	Start program execution
STEP	Singlestep ARM7 core
STEPN	Execute n instructions
RUNS	RUN Specific - Run ARM7core to a specific address using single stepping, no breakpoints are used in this case.
BPS/BPD	Breakpoint Set/Delete - Set/delete a breakpoint
WPS/WPD	Watchpoint Set/Delete - Set/delete a watchpoint
SIMSTUCKAT	SIMulate STUCKAT - Simulate stuck-at fault at a specified memory address. Both possible stuck-at faults can be simulated.
SIMCROSSTALK	SIMulate CROSSTALK Simulate a coupling fault for two specified addresses (uni-directional). The crosstalk simulation is similar to the stuck-at simulation. The only difference is that the stuck-at is injected into the influenced memory cell.
RM	Read Memory cell - 1, 2 or 4 byte can be read
CM	Compare Memory cell to given data. It can be specified that the test has to be aborted, if the data do or do not match.
WM	Write Memory content - 1, 2 or 4 bytes can be written.
READMEMIMG	READ MEMory IMaGe Read the complete memory and write it to a file.
RR	Read Register
CR	Compare Register content to a given value, if the data do or do not match, the test can be aborted.
WR	Write Register content – a value is written to the specified register
RAR	Read All processor Registers
RCF	Read Condition Flags
WCF	Write Condition Flags
OCF/ACF/XCF	Or Condition Flags And Condition Flags Xor Condition Flags Use mask to modify the flags, using OR/AND/XOR
Other phrases	Other phrases are interpreted as comments which are also written into the test report.

As soon as the simulation is executed it receives its tasks from the input file, a message is logged to the report file and the core is set into run mode until either a watchpoint or a breakpoint is reached. In case of a watchpoint, they are used to simulate a coupling- or stuck-at-fault, the processor performs the necessary data manipulation and is set to run mode again.

After stopping the program execution at a breakpoint the test tool refers to the next line of the input file in order to get new instructions.

6.2. Standard Workflow

The standard workflow for the execution of a test case is shown in Figure 5. First the program has to be compiled, linked and downloaded into the microcontrollers FLASH storage using the development tools of the project (e.g. an integrated development environment). The second important step is to specify the possible and necessary breakpoints for the test respectively. In order to recognize misbehaving parts of a program, one breakpoint is set where the memory contents can be manipulated. A second breakpoint is set to the return instruction of a function which returns an OK value or a failure value. This return value (which is usually stored in register R0) is compared to the expected value. The position of the breakpoints within the program flow can be found in the MAP file, which is generated by the linker.

After downloading the hardware test to the microcontroller, the test tool is started gaining its commands from the file. The first instructions within the csv-file must contain the specification of the breakpoints, followed by a reset and a run instruction. At some point within the program execution the configured break- or watchpoint is hit. Once the processor stops, the actions defined in the input file are performed. Afterwards a breakpoint is set to an instruction, e.g. to a return command, where the effects of the performed actions can be evaluated. Based on this evaluation the test of the hardware test can be considered as failed or passed.

Using this methodology allows to perform various tests. Within one input file several test cases can be placed consecutively. For starting a new test case the former must be performed successfully.

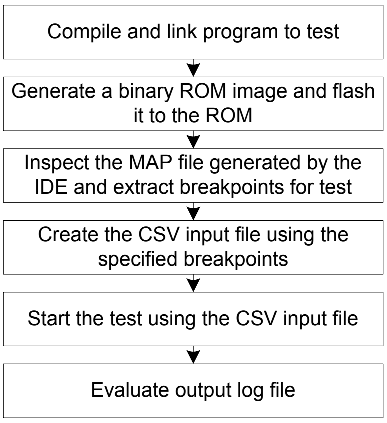
Compile and link program to test
Generate a binary ROM image and flash it to the ROM
Inspect the MAP file generated by the IDE and extract breakpoints for test
Create the CSV input file using the specified breakpoints
Start the test using the CSV input file
Evaluate output log file

Figure 5. Work flow of a test case execution

6.3. Sample Test Input File

The following lines show an example of an input file written in csv-format:

```
TEST30-1;;;;
HALT;;;;;
BPD;0;;;;
BPS;0;14476;;;
BPS;1;15252;;;
RESET;;;;;
RUN;;;;;
WR;2;0xFFFFFFFE;;;
RUN;;;;;
CR;0;0;NORES;PONM;Error -> test failed
```

First the test is named. The name of the test case is written to the output log file. At the beginning the ARM7 core is halted and the breakpoints 0 and 1 of previous test cases are deleted. Afterwards the breakpoints which are necessary for the test are set. Breakpoint 0 is set to a position within the code where a calculation result is compared to a special value. The second breakpoint – breakpoint 1 is set to the return of the function in order to be able to check the return value of the function. The ARM7 core is then reset and started. After the RUN statement the test tool waits until the ARM7 core is halted. This happens at breakpoint 0. At this position the register R3 which contains the calculation result, is manipulated in order to contain a wrong value. The ARM7 core is set into run state again and is halted at breakpoint 1. At this position in the source code, register R0 should contain a value (return value) different to 0. No reset has to be done after the comparison – the test is passed, if the value of R0 is not equal to 0 (PONM – pass on no match). If failed, the message “Error -> test failed” is written to the log file.

```
0  Automatic Test log using JLINK
   Start of logging at Mon Jun 26 21:28:51 2006
   DLL Version: 3.20a
   JLINK Compilation Date: Apr 27 2006 23:07:04
   Firmware: J-Link compiled Apr 27 2006
   12:55:19 ARM Rev.5
   J-Link speed = 30
   ARM core ID: 0x3F0F0F0F
***********************************
   TEST30-1
***********************************
81 ARM Core halted
82 Breakpoint number 0 deleted
83 Breakpoint number 0 set at address 14476
84 ARM Core reset
85 ARM core started
   ARM core halted
   Current position: 0x0000388C
86 Value 0xFFFFFFFE (-2) written to ARM_REG_R2
87 ARM core started
   ARM core halted
   Current position: 0x00003B94
88 ARM_REG_R0 = FFFFFFFF
        Register    content    0xFFFFFFFF    of
        ARM_REG_R0  does  not  match  to  value
        0x00000000 (0)
        (null)
*********** T E S T   P A S S E D **********
```

The sample above shows the output of the automated test presented at the beginning of the section. The header of the output file presents information on the hardware and software versions used and the test start time. Afterwards the test identifier is shown and the information on the test results is logged. In the end of the test case the test condition is verified and the test case is logged as passed.

7. Conclusion

Using boundary scan technology paired with the debugging features offered from the ARM7 core allows the simulation of faults without having direct access to the internal components. For the presented method no expensive hardware or complex software is required.

It was shown that a relatively limited tool is able to perform the necessary fault injection in order to verify the correct behavior of hardware tests. Due to the high degree of adaptation the test tool may not be suitable for performing tests within a comprehensive project. The advantage of the presented method is that no adapted software running on the device under test is required. Instead of adapted software extensive study of the linker generated map files is necessary to set the appropriate breakpoints and parameters. An additional drawback is the changed timing behavior. Every time a hardware fault is simulated the target processor is halted and the appropriate tasks to simulate a hardware fault take place.

By means of the J-Link application program it is possible to controll the ARM7 core internals via a high level programming language. Hence the test tool can easily be adapted to new requirements without knowing details about the underlying communication from the development host and the target hardware.

Results after performing tests show that white box tests as well as black box tests can be performed very efficiently. To enhance the usability of the test tool a more convenient method to extract and set the required break points might be useful.

References

[1] "IEC 61508 – Functional safety of electric/electronic/programmable electronic safety-related systems", 1999.

[2] H. Hölscher, J. Rader, *Microcomputers in Safety Technique*, TÜV Rheinland, 1986.

[3] Ravindra Nair, Satish M. Thatte, Jacob A. Abraham, "Efficient Algorithms for Testing Semiconductor Random-Access Memories", *IEEE Transactions on Computers*, Volume 27, pp. 572-576, 1978.

[4] L. Wang, K. Ch. Tan, "Software Testing for Safety-Critical Applications", *IEEE Instrumentation & Measurement Magazine*, pp.38-47, 2005.

[5] W. Zuser, S. Biffl, T. Grechnig, M. Köhle, *Software-Engineering mit UML und dem Unified Process,* Pearson Studium, München, Germany, 2001.

[6] "EN 14908 – Open data communication in building automation, controls and building management – control network protocol", 2006.

[7] "CANopen Framework for Safety-Related Communication", *CiA work Draft 304*, CAN in Automation e.V., 2000.

[8] "PROFISafe, Profile for Failsafe Technology," *PROFIBUS-Nutzorganisation Karlsruhe*, v1.0, 1999.

[9] P. Preininger, "Hardware Selftests For Safety Critical Fieldbus Nodes", *M.S.thesis,* Institute of Computer Technology, Vienna University of Technology, Austria, 2006.

[10] M.C. Hsueh, T.K. Tsai, R.K. Iyer, "Fault Injection Techniques and Tools", *Computer*, Volume 30, pp. 75-82, 1997.

[11] G.A. Kanawati, N. A. Kanawati. J.A. Abraham, "FERRARI: A Tool for The Validation of System Dependability Properties", *Twenty-Second International Symposium on fault-Tolerant Computing, FTCS-22. Digest of Papers*, pp.336-344, 1992.

[12] J. Carreira, H. Madeira, J.G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Transactions on Software Engineering*, Volume 24, pp. 125-136, 1998.

[13] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R.Dancey, A. Robinson, T. Lin, "FIAT-Fault Injection Based Automated Testing Environment", *Eighteenth International Symposium on fault-Tolerant Computing, FTCS-18. Digest of Papers*, pp. 102-107, 1998.

[14] S. Chau, "Fault Injection Boundary Scan Design for Verification of Fault Tolerant Systems", *Proceedings of International Test Conference*, pp. 677-682, 1994.

[15] "IEEE Standard Test Access Port and Boundary-Scan Architecture", IEEE Std 1149.1-2001.

[16] ARM7TDMI (Rev 4) Technical Reference Manual, ARM.

[17] Segger, J-Link ARM API, "Users guide of the J-Link application program interface (API)", Version 3.20, Manual Rev. 1.