Auxiliary IP Blocks for Early Dependability Analysis of Small Processor based Systems

J. Barboza, J. Basualdo, J. Perez Acle
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

***Abstract*—Fault injection experiments are a powerful aid to identify and fix problems in the design of fault tolerance mechanisms, particularly when performed at early development phases. For this purpose, it is important not only to classify the faults, but also to understand the different faulty behaviors. When an embedded system is considered, a common approach for analyzing the faulty behavior is to exploit the execution trace features often available in medium to high size processors. This paper proposes two IP modules intended to facilitate fault injection experiments in small processor systems: a memory saboteur and a bus event recorder. The former allows the injection of SEU and stuck-at faults, both at a specific memory location and at the address or data bus level. The latter provides an alternative to the use of a full execution trace solution, which is often not available in small processors. The IP blocks were used to inject the faults and to analyze the behavior of a submodule of an implantable pulse generator running on an FPGA-hosted openMSP430 processor system. The IP blocks, the fault injection environment and the results of the fault injection campaigns are presented. The event traces captured by the event recorder IP played a fundamental role to understand the faulty behavior.**

Keywords—dependability analysis; fault injection; saboteur; openMSP430; MSP430; memory saboteur; bus event recorder

I. INTRODUCTION

Fault injection has been extensively used for the dependability validation of embedded systems, both for fault removal and fault forecasting [1]. When the objective is fault forecasting, the expected outcome of the injection campaign is to classify the faults and obtain the fault coverage provided by the detection and tolerance mechanisms.

On the other hand, when fault injection is performed at the earlier development phases, usually the predominant objective is fault removal. In this case, fault injection is the first step of a verification-diagnosis-correction process. The faulty behavior must be understood in order to identify and fix what is wrong in the detection and tolerance mechanisms under development. This raises the need for more observability on the fault injection experiment.

When the system under analysis includes a microprocessor, an inexpensive solution to analyze the faulty behavior is the use of the debug and execution trace resources offered by medium to high size modern processors. These tools, particularly the second one, provide a deep insight of the behavior of the faulty system without sacrificing execution speed. However, execution trace tools are usually not available in small processors and slower solutions must be used.

Here we present a bus event recorder instrumentation IP designed to facilitate emulation-based fault injection experiments in small processor systems. With a proper selection of the event matching conditions, the resulting log enables a thorough understanding of the faulty behavior. We also present a memory saboteur that allows for the injection of SEU and stuck-at faults, both at a specific memory location and at the address or data bus lines.

Both IP blocks were used to inject faults and to analyze the resulting behavior of a module of an implantable pulse generator under development. The module runs on an FPGA-hosted openMSP430 processor system. The event traces captured by the event recorder IP played a fundamental role to understand the faulty behavior. The event traces also allowed to evaluate if the faulty behavior could be harmful for the patient using the implantable device.

The following Section briefly presents some of the most commonly used instrumentation IPs for fault injection and fault effect observation. In Section III we describe the proposed instrumentation IPs. Section IV details the experimental setup and presents the preliminary results of a fault injection campaign. Finally in Section V some conclusions are presented.

II. BACKGROUND

Different fault injection techniques are commonly used to assess the effectiveness of fault detection and fault tolerance mechanisms [2]. Here we concentrate in the emulation-based technique [3], i.e., a prototype of the system under evaluation is synthesized, usually inside an FPGA, together with the instrumentation needed to inject the faults and to observe its effects.

Some of the most usual solutions for this instrumentation are briefly described in the following subsections.

A. Injecting faults

Saboteurs are often used as the means to inject a fault at some point in a circuit. They were initially introduced in [4] for simulation-based fault injection, and are extensively used

This work was partially supported by grant POS_NAC_2013_1_11657 from the Agencia Nacional de Investigación e Innovación, Uruguay .

also in emulation-based fault injection. A *serial saboteur* is a block inserted in the path between a driver output and its corresponding receivers. Additional block inputs are needed to control if the output is modified according to some fault model, or if it is left equal to the input for normal behavior

This scheme can be directly exploited, for example, to implement permanent *stuck-at* faults on a bus line.

More elaboration is needed when trying to inject faults in individual memory locations. The direct solution of adding a saboteur for each location is either unfeasible because of the excessive area cost involved, or directly impossible if the memory is implemented using the memory blocks embedded inside the FPGA. In this situation a global saboteur can be inserted at the memory output bus. The controller that drives the saboteur control inputs must evaluate the memory address and control signals in order to activate the fault only when the location under attack is being accessed.

SEU faults can be faced by triggering a *read-modify-write* cycle at the injection time. This approach can be used to inject faults in individual flip-flops [5] or at memory locations [6]. Another approach consists in providing additional logic to the saboteur controller described above in order to activate the fault at the injection time and de-activate it if a new write operation to the location under attack is detected. This latter approach is the one we used, as described below.

B. *Observing faulty behavior*

The simplest solution for observing fault injection experiments of a processor based system would be waiting until the program reaches the exit point and observing the memory to check if the results produced by the program were affected by the fault. This is not enough because the normal execution flow is modified by effect of the injected fault; consequently, additional program exit points must be identified. Moreover, a timeout mechanism must be used because the faulty execution flow, or even the internal processor logic, can remain in an endless loop.

A low cost solution to implement an environment like the one described above is to use a watchdog or timer as a timeout mechanism, and the debug features offered by almost all the processors to capture the exit points and observe the final memory contents.

However, if I/O activity is involved, and especially if there are safety requirements associated to it, I/O operations must be observed to decide if the faulty behavior is safe or not. Also, when the main goal is to fix possible detection or tolerance problems, a more detailed view of the execution flow is needed. In these situations the execution and/or bus activity trace tools available in most processors are a valuable tool.

Examples of the execution/bus activity trace features available in modern processors are the ARM CoreSight On-chip Trace and Debug Architecture [7] available in ARM based System on Chip designs, and the AHB Trace buffer peripheral available for the Leon3 processor [8].

Smaller processors usually do not provide full trace capabilities, or in the best case they have a very limited capacity. For example, the Enhanced Emulation Module [9] available on some MSP430 processors from Texas provides advanced debugging including breakpoint conditions triggered by program or data access conditions, but the trace depth is limited to the 8 instructions that precede a breakpoint.

III. PROPOSED IP MODULES DESCRIPTION

A. *Saboteur*

A saboteur, like the one described at the end of Section *II.A. Injecting faults*, was developed targeting an openMSP430 processor system [12]. The openMSP430 processor core [10] is a processor compatible with the Texas Instruments MSP430 family and is available at the Opencores repository [11].

The saboteur can inject faults at the address and data bus lines, and also at a specific memory location.

For the faults injected at a bus line, the supported fault models are permanent *stuck-at-0* and *stuck-at-1* and are configured by a mask specifying the affected bit positions. For the memory location faults the same fault models plus SEU (bit-flip) faults are supported. Additional configuration is needed to specify the affected memory address. In the case of SEU faults, the time at which the fault must be injected should also be specified.

The SEU faults remain active since the configurable injection time until the module detects a write operation on the same memory location.

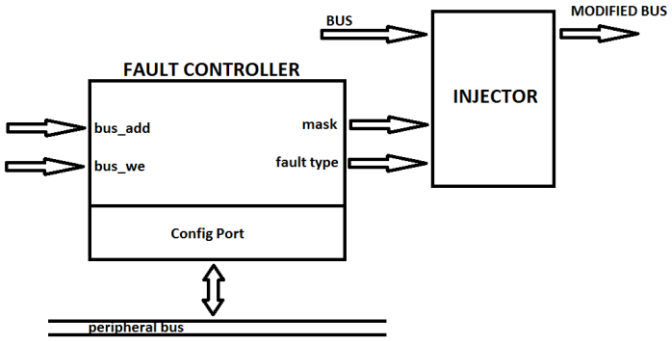

Fig. 1. Connection and main structure of the Saboteur.

The saboteur architecture consists of a fault injection controller and two saboteur blocks to be inserted in the memory address and data output buses. Fig. 1 is a block diagram, showing only one of the saboteurs for clarity. For each saboteur, the fault injection controller generates a mask with value "1" at the positions where a fault must be injected or all zeroes at the times no fault must be injected. An additional control signal indicates the type of fault that must be injected. The saboteur is purely combinatorial and produces the proper modification according to the fault type indicated by the controller.

The faults corresponding to specific memory locations are injected at the output data bus. The fault injection controller monitors the memory address and control signals: when a read access to the configured address is detected, the mask is activated so that the faulty value is produced by the saboteur

block; when a write operation is detected and there is an active SEU type fault, the fault is de-activated and remains inactive until reconfiguration.

The saboteur block is shown in Fig. 2 and Table I. It receives the mask and fault type from the controller and applies the proper bitwise operation according to the fault type. Note that when no fault is being applied the controller generates an "all zeroes" mask so that all the gates leave the input unmodified.

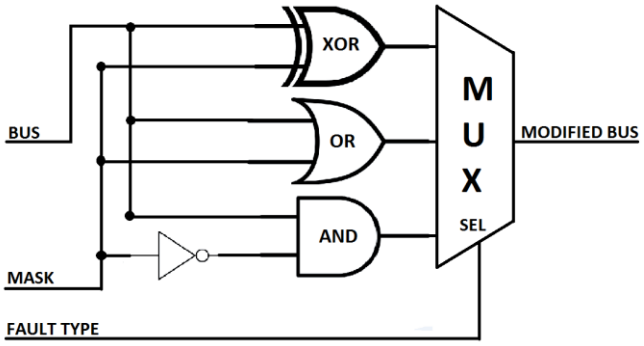

Fig. 2. Structure of the Injector block.

TABLE I. LOGIC USED FOR EACH TYPE OF FAULT

Fault type	Logic operation	Input	Output
SEU	XOR	1	0
		0	1
Stuck-at '1'	OR	1	1
		0	1
Stuck-at '0'	AND	1	0
		0	0

B. Event recorder

For the fault injection experiment to be representative, it is desirable to maintain the system under study with a minimum of changes. For this reason, in order to be able to capture a log of the program behavior during each run, a new peripheral was developed –the Event Recorder. This peripheral allows obtaining such a log while running the system at normal speed and without modifying the firmware code.

The Event Recorder captures the occurrence of pre-configured events in the OpenMSP430 buses and then stores them in an internal memory (FIFO). The events that can be configured are read or write transfers with matching values in address or data bus. Once a capture takes place, signals of Program memory and Data memory buses corresponding to the last bus transfer are stored. Additionally, together with these signals, a time-stamp word and a bit mask (indicating the event that triggered the capture) are also stored. The time-stamp must be generated by an external timer.

This new peripheral is managed completely through the peripheral bus. This allows the user to configure and read the Event Recorder either from the program executed by the microcontroller or through the debugging unit using GDB debugger.

The hardware description of this block was written using VHDL. After synthesizing for an Altera Cyclone III chip, the following summary was obtained:

- Total logic elements: 1.626 / 15.408 (11 %).
- Total memory bits: 26.624 / 516.096 (5 %). This can be adjusted by modifying the size of the FIFO.
- Maximum operation frequency: 93.46 MHz.

1) Internal structure

The peripheral is basically composed by 5 main blocks as shown in Fig. 3.

Buses interface: This block connects the peripheral bus to the Control registers and Data output registers blocks. Besides that, it uses latches for storing the last transaction carried out on the program memory and data memory bus.

Control registers: Besides implementing the registers which are mapped in peripheral space, this block contains the set of registers that stores the events to be captured. The number of events is configured at compile time.

Comparator: This is a combinatory block which compares the data of the last bus transaction (coming from the Buses interface) with the content of the registers implemented in the Control registers block. When a coincidence occurs, the data of the buses is sent to the FIFO block, along with a bit mask identifying the event that triggered the capture and a write enable signal connected to the FIFO through an edge detector.

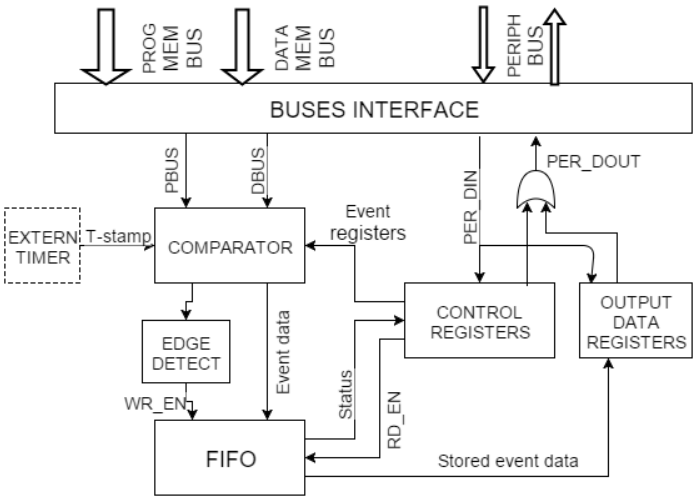

Fig. 3. Main structure of the Event Recorder.

FIFO: This FIFO memory stores the comparator output when the write enable signal is activated. The size of the FIFO is configurable at compile time.

Output data registers: This block is intended to make the content of the FIFO accessible from the peripheral bus; it divides the FIFO output into 16 bits words which are mapped in peripheral space.

2) General operation of the peripheral

To correctly configure and read the Bus Event Recorder, the following steps must be performed:

a) Enabling peripheral writing

Since this peripheral is aimed to be part of a fault injection system, it is necessary to avoid possibly undesirable reconfigurations during the injection experiments. This is due to the fact that, in presence of a fault, the program being executed could behave in an unexpected way, performing writing operations in the Bus Event Recorder addresses.

In order to reduce the probability of this occurring, a mechanism that allows enabling and disabling writing operations was implemented. A password has to be written in a certain peripheral address so as to enable writing in the peripheral, and similarly, a different password can be written to disable writing. The address, as well as the passwords, is configurable before compiling the project.

In particular, for the application described in this paper, the address was set to 0x01C6 (base address + 0x0006), the enabling password to 0x5555 and the disabling password to 0xAAAA.

b) Events to capture

The types of event that the Bus Event Recorder is able to capture are listed in Table II. For example, the first row type of event is activated when writing to the Memory Data Bus at an address matching a pre-configured value.

TABLE II. TYPES OF EVENT

Transfer type	Matching condition	Bus
WRITE	Address	Data Memory Bus
WRITE	Data	Data Memory Bus
READ	Address	Data Memory Bus
READ	Data	Data Memory Bus
READ	Address	Program Memory Bus
READ	Data	Program Memory Bus

For each event that the peripheral must capture, two write operations must be done to configuration registers. First, CTRL 1 register (base address) must be written with the event identifier, a flag indicating whether the event associated with this identifier is active and the type of event to be captured. Second, CTRL 2 register (base address + 0x0002) must be written with the matching value for address or data.

These steps must be repeated for each event that must be configured. Once the event configuration is finished, it is recommended to disable the peripheral writing as explained above.

c) Reading captured events

Before reading a captured event, it is necessary to perform a writing operation on base address + 0x0008. This action extracts the previous event from the FIFO allowing for the reading of a new event. The event can be read through the peripheral bus in a range of 18 addresses mapped on peripheral space.

In addition, the system has a status flag register which indicates the amount of events in the FIFO. Empty and full flags are also available.

IV. EXPERIMENTAL SETUP

The device under study was a sub-module of a system based on the microcontroller MSP430 which is intended to be used in an implantable pulse generator (IPG). The IPGs are a class of biomedical equipment which best-known example is the cardiac pacemaker. The chosen sub-module was that responsible for managing the therapy delivered to the patient, for which its dependability features are critical. This sub-module was implemented in an FPGA-hosted OpenMSP430 microcontroller and the IP modules described above were integrated to the system. Finally, the FARM model [1] was used to perform the fault injections campaigns.

A. System emulation and IP modules integration

As the system to be evaluated is based on a MSP430 microcontroller, the OpenMSP430 was used to emulate it. This project is compatible with the family of the microcontroller used by the system

The program memory size of the OpenMSP430 was configured to 24 Kbytes and the data memory size to 10 Kbytes in order to adapt the system to the firmware requirements. A ROM was used for storing the program code so as to better emulate the actual system, which uses flash memory for this purpose. The clock frequency was set to 20 MHz so as to work just like the actual system.

Two Saboteur modules were used, one for each memory (program and data). The Saboteur modules as well as the Event Recorder were connected to the peripheral bus so as to be able to configure and read them. The Saboteur modules were connected to the corresponding address bus and between the output data of the memory and the data bus of the OpenMSP430. On the other hand, the Event Recorder was connected to the program and data memory buses of the microcontroller.

In addition, a new timer A was implemented so as to provide the Event Recorder with a time-stamp. This timer is also used to establish a timeout for the fault injection runs. Taking this into account, the interrupt request output of the timer was connected to the NMI input of the OpenMSP430. In this way, it is possible to interrupt the program execution and lead the program counter to a known address (NMI ISR) in case the program execution takes an unpredictable behavior.

The hardware description was synthesized in an Altera Cyclone III FPGA. The firmware of the system under study was compiled with GCC tools and it was loaded and debugged into the platform using GDB. The OpenMSP430 UART serial debug interface was used to send the GDB commands to the on-chip debug unit.

Even though the IP modules were used with a particular system based on the OpenMSP430 and they were configured through the debugging interface of this processor, it is worth to point out that their configuration interfaces can be easily adapted to any other external communication interface and therefore used in systems based on other microprocessors. In case of the *Event Recorder*, for using it with another processor, it is also necessary to modify the *Comparator* block so as to adapt it to the structure of the new processor buses.

As mentioned in Section III.A. *Saboteur*, the *Saboteur* is intended to inject faults on memory and also on the address/data buses lines. This module can also be used for injecting faults in the register file of the processor, provided register file buses are accessible.

B. Fault injection campaigns

1) F (Faults)

The fault space assigned to the experiments performed in this work includes only stuck-at faults (either at '0' or '1') in program memory space.

Each stuck-at fault is determined by three elements: the memory address to be attacked, the bit number within this address and the type of fault (stuck at '0' or '1').

Matlab was used for generating a vector of 5.984 faults. The elements that characterize each fault were generated as follows:

- Addresses: The address of each fault was generated using a uniform distribution between 0xA000 and 0xF230 which is the memory space filled by the firmware code. There are no repeated addresses.
- Bit number: This element was generated by a uniform distribution between 0x0 and 0xF.
- Type of fault: The program memory content was disassembled and saved in a Matlab file (.m). Once the address of the fault and the bit number were generated, the memory content was used to get the bit value at that location. If the memory contained a '0', the fault type set to stuck at '1' and if the memory contained a '1', the fault type was set to stuck at '0'.

The fault vector obtained with this procedure was used to configure the Saboteur during each run.

2) A (Activation)

The fault activation stage consists in executing a therapy routine after the fault was set. This routine includes a system integrity check which is performed before delivering therapy pulses. When this integrity check fails, the program goes into a safe mode, which implies aborting the therapy.

The therapy was configured with standard parameters before the firmware compilation. This therapy has a periodic behavior and as a consequence, only the first cycle is evaluated during the experiments.

Four possible exit points of the program were taken into account for each run: *Normal execution*, the program ends after the first therapy cycle; *Reset*, the program gets back to the start point; *Safe mode*, the program goes into safe mode; *Timeout*, the program has an unexpected behavior and the timeout expires.

The hardware breakpoint capability of the OpenMSP430 was used to set breakpoints at the locations mentioned above. This was necessary due to the fact that a ROM was used for implementing the program memory, for which software breakpoints are not allowed.

The Event Recorder was used to obtain a trace of the program behavior. It was configured to capture the following groups of events which are enough for the evaluation of system misbehaviors:

- Writing on the variable that stores the error status of the system.
- End of the first program cycle.
- Output terminals management.
- Pulses amplitude setting.
- Program reset.

The fault injection experiments are managed through the OpenMSP430 debugging interface (by setting the fault to be injected, configuring the Event Recorder, starting the program execution and getting the results). For this reason, and with the aim of automating the experiment, a GDB script was developed. The orders of the script are the following:

1. Connection with the target.
2. Hardware breakpoints setting.
3. Event Recorder configuration.
4. Processor reset (provoked by writing an invalid value to the watchdog control register).
5. Timeout setting.
6. Saboteur setting.
7. Start of program execution.
8. Reading of the events captured by the Event Recorder.
9. Repeat steps 4 to 8 with a new fault.

The program execution evaluated in each run includes the integrity check routine and the first cycle of the therapy pulses. This has a duration of 654.860 clock cycles (32,743 ms).

3) R (Readouts)

The results obtained from the experiment are based in the data extracted from the Event Recorder after each run. This information is compared with that obtained with a golden run (run without faults) in order to determine whether there was a misbehavior.

It is considered that there was a misbehavior if some of the following occurrences took place:

- The exit point of the program was different from that of a normal execution.
- The order of the captured events was different from that of the golden run.
- Any pulse amplitude was set incorrectly.
- The output terminals are set incorrectly.
- The amount of therapy pulses or their duration was wrong.

The faults are classified according the kind of misbehavior that they produce:

- Dormant faults: They had not produced an error in the system up to the end of the experiment.
- Active Detected faults: The system detected the fault and consequently had a safe behavior.
- Active Undetected Safe for the patient: no therapy is delivered, it is aborted prematurely, or it is incorrect but safe for the patient.
- Active Undetected Potentially Harmful: the energy of the delivered pulses is increased (for instance, modifying pulse amplitude or pulse width).

4) M (Measures)

Results of the 5.984 experiments were analyzed and compared with a golden run. A summary is shown in Table III.

TABLE III. PRELIMINARY EXPERIMENT RESULTS

Dormant faults			5.680
Active faults	Detected		32
	Not Detected	Safe	268
		Potentially harmful	4

The faults classified as potentially harmful produced an unexpected pulse amplitude setting while the therapy was being delivered. If the new amplitude is greater than the correct one, there could be adverse effects.

For that reason, these 4 faults were individually analyzed in order to check the amplitude that was being set. The analysis showed that even though the function that set the amplitude was called improperly, the amplitude was finally set with the correct value.

Based on the obtained results, 5,1 % of the injected faults produced misbehaviors in the system. Only a 0,07 % provoked a potentially harmful misbehavior; however, after analyzing them in detail, it was verified that they are not risky.

The set of faults taken into account during the experiment totalizes 3,56 % of the fault space (considering only the stuck-at faults in program memory).

V. CONCLUSIONS

Two IP modules, a saboteur and an event recorder, were developed for low cost fault injection on small processor-based embedded systems.

Both modules were successfully used to perform fault injection experiments on a prototype of an implantable pulse generator under development.

Preliminary results of a fault injection campaign for permanent faults in program memory were presented.

The saboteur provides a flexible solution for fault injection in memory and interconnection buses, both for SEUs and permanent stuck-at faults.

The event traces captured by the event recorder IP played a fundamental role to understand the faulty behavior for several faults and to determine if the faulty behavior can be harmful for the patient using the implantable device.

References

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," IEEE Trans. Softw. Eng., vol. 16, no. 2, pp. 166–182, 1990.

[2] Y. Yu and B. W. Johnson, "Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation," A. Benso and P. Prinetto, Eds. Kluwer, 2003, pp. 7–39.

[3] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante, "FPGA-based fault injection for microprocessor systems," in Proceedings 10th Asian Test Symposium, 2001, pp. 304–309.

[4] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: the MEFISTO tool," in Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing, 1994, pp. 66–75.

[5] J. Perez Acle, M. S. Reorda, and M. Violante, "Early, accurate dependability analysis of CAN-based networked systems," IEEE Des. Test Comput., vol. 23, no. 1, pp. 38–45, Jan. 2006.

[6] J.-M. Daveau, A. Blampey, G. Gasiot, J. Bulone, and P. Roche, "An industrial fault injection platform for soft-error dependability analysis and hardening of complex system-on-a-chip," in 2009 IEEE International Reliability Physics Symposium, 2009, pp. 212–220.

[7] ARM CoreSight On-chip Trace and Debug Architecture, http://www.arm.com/products/system-ip/debug-trace/

[8] J. Gaisler, E. Catovic, M. Isomaki, K. Glembo, and S. Habinc, "GRLIB IP core user's manual. Version 1.3.7 - B4144," 2014.

[9] "SLAA263F–September 2005–Revised July 2015 - Application Report - Advanced Debugging Using the Enhanced Emulation Module (EEM) With IAR Embedded Workbench 5.60," 2005.

[10] O. Girard, "OpenMSP430," 2009.

[11] OpenCores community for development of hardware IP cores as open source, http://opencores.org/

[12] J. Basualdo, M. Vazquez, and F. Viera, "SATELITEST: Test de inyeccion de fallas en satelite ANTEL-SAT." Montevideo, p. 140, 2014.