Component for Debugging of Interrupt Based Systems

Alain Tamayo-Fong[1], Luis E. Leyva-del-Foyo[1&2], Pedro Mejia-Alvarez[2]

[1] Dpto. de Computación, Universidad de Oriente, *90500,* Santiago de Cuba, Cuba
[2] Dpto. de Computación, CINVESTAV-IPN, Av. I.P.N. 2508, México, D.F., 07300
alaint@csd.uo.edu.cu, leyvadelfoyo@yahoo.com, pmejia@cs.cinvestav.mx

Abstract. With the aim of developing and implementing a real-time operating system with experimental purpose, the development of a component that allows the debugging and verification of the correct behavior of the interrupt handling mechanism to its lowest level has been needed. This has been achieved through the simulation of interrupt requests at the hardware level. This module has been designed as an independent and reusable component that can be very useful not only for its original purpose, but also for the development of others embedded and real time systems. Besides, the component has a great educational value and as such, it has been used in laboratory practices in the teaching of the hardware interrupt mechanisms of PC systems.

1. Introduction

Hardware interrupts are an important part of computer architectures. Its purpose is to provide an efficient way for external devices to get the processor attention. Debugging interrupt-based systems is a vital subject due to of the inherent complexity of this kind of system. This complexity comes from the fact that interrupts may happen at any non predictable time, and can happen some of them at the same time.

Interrupt hardware assigns to each *Interrupt Request Line* (IRQ) a default static priority order. When an interrupt occurs, the task executed by the processor at that instant is interrupted. Execution control is transferred to the *Interrupt Service Routine* (ISR) assigned to the signaled IRQ line. While this routine is being executed, interrupt request lines with lower priority are disabled. An ISR can be interrupted only by a higher priority IRQ. Once served all IRQs, the interrupted task is resumed.

Computer systems using Intel processors and compliant with the industry standard [3], use an interrupt hardware compound by two *Programmable Interrupt Controller* (PIC) 8259 chips connected in cascade throw the IRQ2 line of the first 8259. This configuration provides 16 IRQ lines (IRQ0...IRQ15).

As part of the development of a *Real-Time Operating System* (RTOS) with special features concerning interrupt handling, the development of a component that allows for simulation of interrupt requests at the hardware level in a controlled way was needed. With this component we were able of debugging, testing, tuning and verification of the correct behavior of the interrupt handling mechanism. In spite of the existence of many low level debuggers [14], we did not find any debugging tool with the required features to accomplish this task.

Control, Virtual Instrumentation and Digital Systems.
Research in Computing Science 24, 2006, pp. 31-41

The main contribution of this work is the design of a module that allows the issue of hardware IRQs in any point of the code controlled by software. This avoids one of the greatest problems of interrupt-based systems: interrupt asynchrony. Furthermore, after the code is properly instrumented, the change of the sequence of interrupt request (test cases) is possible without further code modification. The module is an independent and reusable component that can be very useful not only to its intended purpose, but on the development process of others embedded and real-time systems.

The rest of this paper is organized as follow: Section 2 summarizes the novel features of the experimental RTOS that bring us to the need of this work; as well as, presents a brief survey of the related recent works. Section 3 explains the component design. Section 4 present a test case showing the way our component can be used. Finally, section 5 outlines the conclusion of our work.

2. Motivation for this work

This work is part of the development of the *Portable and Adaptable Real-Time and Embedded Microkernel Operating System* (PARTEMOS) [10]. This is an experimental microkernel being developed for experimenting novel interrupt and exception handling schemes for real-time and embedded systems [7, 9, 8].

2.1. Interrupt Handling in PARTEMOS

PARTEMOS presents some interesting special features, but for the purpose of this work we will only concentrate on those regarding its interrupt handling. The system transforms interrupts in signals operations over a semaphore [1], allowing the tasks to synchronize with interrupts through *wait*() operations on the semaphore associated with an IRQ line. In this way, PARTEMOS hides the (not desirable) interrupts asynchrony in the deepest part of the system. With this approach an IRQ can be seen as a "hardware task" that send a signal through its associated semaphore.

With the purpose of making the system portable, those aspects depending on the hardware have been located in the lowest layer named *Hardware Abstraction Layer (HAL)*. An important component of this HAL is the *INTerrupt HAL* (INTHAL) which isolates the rest of the kernel from the interrupt hardware, increasing the portability of the system. This layer presents the following specific features:

- Interrupts priorities do not map directly to default interrupt priorities assigned by the hardware. Hardware interrupt priorities can be rearranged as needed.
- Both interrupts and task are within the same priority space. Differently from most know systems where interrupts always have higher priorities than task, PARTEMOS allows tasks and interrupt priorities to be intermixed.

The implementation of these features has made the INTHAL a very complex layer, because the offered behavior is very far from that provided by the hardware. To achieve a correct implementation of this *virtual interrupt controller*, we need a component that eases its debugging. Specifically, we need a component that allows us to generate interrupts in a controlled way. In addition, this component must be flexible enough to build a wide set of test cases.

2.1. Related Works

Bug finding in interrupt driven code is one of the most complex and time consuming task of the overall development process of embedded systems. Recently, the general issue of making bug free interrupt driven code has been faced in different ways. In [15] a method for identify all possible sequences of execution of code in presence of interrupts is proposed, them standard sequential testing techniques can be used. A source code transformation technique presented in [12] turns interrupt-driven code into semantically equivalent thread-based code for checking with standard thread verifier. In [3] a technique for grouping code block for an effective use of model checking to analyze the behavior of interrupt-dependent programs was presented. The work in [4] presents an extension to Petri nets for modeling interrupts. Some techniques for debugging the interrupt driven embedded systems and for coping with the asynchronisms of interrupts were presented in [1]. Any of these techniques can not be used to test low level interrupt management or even the interrupt hardware itself. Existing development environment or debugging tools do not provide means for issue hardware interrupt over software control. The tools presented in this paper can issue hardware interrupt request under software control using a PC hardware trick similar to the used in at-hot way in [13] to test the interrupt behaviors in Windows NT. Our component can be used with any conventional debuggers for avoiding the asynchronisms in interrupt request, letting the repetition of test cases for easy finding of bugs under debugger control.

3. Component for the generation of hardware interrupts.

The designed component contains functions for interrupt generation on computer machines based on Intel Pentium processor or higher with a PCI chipset. Generation of hardware interrupts through software is implemented using an undocumented feature of these chipsets. Manipulating the interrupt hardware edge/level registers produce the activation of some IRQ lines [6]. At the beginning of the ISR serving interrupts generated in this way, the current state of these registers must be restored to its original state; otherwise, the interrupt signal will be generated continuously. It is important to emphasize that not all IRQ lines can be activated using this feature. Furthermore, the interrupts that can be activated this way may vary from machine to machine. For this reason, our component is divided in two parts. The first one is an executable program, named DOSCHK, whose purpose is to determine which interrupts can be generated on the specific machine. The second one is the library *IntrGen.lib* that must be linked with the program that needs to generate the hardware interrupts. This library provides mechanisms to use this feature in a convenient way.

The implementation of the DOSCHK program is simple, and consists in the manipulation of the interrupt hardware edge/level registers. Then, it must check which IRQ lines were activated. The library design is a more complicated issue. A convenient abstraction must be provided to library users, so they can build test cases in a simple and flexible way. This design is presented in the next subsection.

3.1. Library Design

The main objective of the library design is to provide users with a convenient abstraction that eases the building of test cases in a simple and flexible way. This objective includes several aspects as the introduction of minimal modifications to the debugged system, the possibility to change easily from a test case to another and the minimization of details that the person designing the test case must deal with.

A possible solution to this problem is to rely on a function that tries to activate a given IRQ line passed as parameter. This function must be inserted in key points of the debugged system. In this way we can determine how this system reacts to interrupts generated on these points. One advantage of this solution is that it is very easy to implement. Once selected the key points where interrupts must be generated, the interrupt sequence to be generated can be included directly on the source code or it can be stored on variables. The first choice causes that it is necessary to modify and recompile the system code for each test case, so this option is unacceptable. The second choice is more flexible because it allows users to change the sequence of generated interrupts changing the variables values. But, it does not allow users to change the number of interrupts generated on the selected points of the debugged system. The number of interrupts generated on each point must be known when the system is modified. This second choice is superior to the first one but is not flexible enough for our needs. We need to change not only which, but how many interrupts will be generated on each of these points, without changing the system code more than once.

The same problem arises when we need to check the state of the PIC. We could provide a function for returning its state. Calls to this function must be inserted at points of the debugged system where this information is needed. The problem results from the fact that this point might change from one test case to another. The choices are: modify the systems for each test case (which is not acceptable), or modify the systems just once and gather this information in all the places that might sometimes need it. Once again, the inflexibility of this solution is evident.

Another design solution is the definition of objects that encapsulate several interrupt generation requests. These objects must include a mechanism (methods) to be notified of which and how many interrupts must be generated in a given moment. This solution allows users to determine which points of the systems must be modified just once. Then, build a test case consist just of determining the values used to initialize this objects. Each object acts as a queue where interrupt generation requests are stored and provides methods to generate one or more interrupts.

To generate more than one interrupt at a given point we could rely on a method receiving as parameter how many interrupts must be generated from the queue. This solution is not flexible because it requires the number of interrupts to be specified on the source code directly or through variables. None of these choices is effective because one implies the modification of the system code for each test case, and the other, that we must control separately the interrupts to generate and the number of them, complicating the design of the test cases. The best solution is to rely on an integrated mechanism where an object could determine just from its state which and how many interrupts generate in each point. Following this reasoning we could include in the queue some marks to indicate how many interrupts must be generated

at each point. This solution simplifies the design of test cases because all the aspects regarding interrupt generation are part of a single object. In the same way, the request of information about the interrupt controller state can be included in this queue too. By including these requests on the queue object we gain total control about what action to execute at each important point of the system: generate a variable number of interrupts or obtain state information from the interrupt controller.

From this analysis we decide to implement a class representing a *command queue.* This queue will contain commands to generate interrupts, show state information of the interrupt controller and special markers to group commands so they could be executed in a single function call. This class will also contain methods to generate one or more interrupts. To generate a single interrupt a command is taken from the front of the queue and a function for activating the corresponding IRQ line is called. To generate more than one interrupt, commands are taken from the front of the queue and executed until a marker or the end of the queue is found.

3.2. Library interface

The *IntrGen.lib* exports the class *commmandQueue* declared in the header file *IntrGen.h.* This class implements the command queue using a structure containing an array of configurable size where commands are stored. The structure also contains the indexes of the first and last command on the queue. *commmandQueue* exports methods to: (1) Initialize the queue (*initCQ()*), (2) insert commands in the queue (*postCQ()*), (3) execute a single command from the queue (*execCQ()*), (4) execute several commands from the queue (*execAllCQ()*)

The initialization function *initCQ()* takes the maximum number of commands that the queue can hold and allocate the memory needed for holding the commands array.

The function *postCQ()* is used to insert commands in the queue. This function receives a variable number of parameters. The first parameter is the number of commands to insert and is followed by the command set.

The commands can be executed individually, calling the function *execCQ()*; or by groups, calling the function *execAllCQ()* that executes commands inside a loop. The command types that can be used are:

Activate an IRQ line
Request information from the PIC
Marker

The first command activates an IRQ line from the PIC; this is exactly what an external device does when it needs processor attention. These commands are named adding the interrupt line number to the identifier IRQ. For example: IRQ3, IRQ4, etc. The second command allows the users to read the PIC state. The 8259 PIC has three state registers, each one with a bit associated to each IRQ line: the *Interrupt Request Register (IRR)* indicates which IRQ lines are requesting the CPU attention; the *Interrupt Service Register (ISR)* shows which interrupts are being serviced; and the *Interrupt Mask Register (IMR)* indicate which interrupt lines are disabled. The PIC information can be read partially (one register at a time) using the CS_IRR, CS_ISR and CS_IMR commands; or totally using the CS_ALL command.

Markers provide a mechanism for grouping of commands that could be executed on a single function call. When the function *execAllCQ()* is invoked commands from the queue are executed until a marker or the end of the queue is found. If a marker is found when the function *execCQ()* is called, no action is taken.

Besides the commandQueue class, the library exports a function named *disableFLIRQ()*. This function must be called inside the ISR associated to interrupts generated through the edge/level register manipulation, for restoring these registers to its normal state. If this function were not called the IRQ line would continue generating this interrupt indefinitely.

4. Test Case using the Interrupt Generation Component

As an example, this section presents a test case for studying the interrupt controller behavior when nested interrupts occurs on the MS-DOS operating system. This operating system was chosen because it provides unlimited access to hardware. On the design of the test case, special care has been taken in avoiding the used of interrupts that might lead to system malfunctioning.

The first step is to determine which interrupts lines can be activated by software using the DOSCHK program. The generated interrupts might vary from machine to machine. A 166 MHz Pentium PC running MS-DOS 6.11 produced the output:

```
Checking interrupts activated through the edge/level registers...

activated IRQs: IRQ3 IRQ4 IRQ5 IRQ7 IRQ15
masked IRQs: IRQ3 IRQ4 IRQ5 IRQ7
```

This output shows that IRQ lines IRQ3, IRQ4, IRQ5, IRQ7 and IRQ15 can be activated by software. It also shows that IRQ lines IRQ3, IRQ4, IRQ5 and IRQ7 are masked. Next, we will use lines IRQ3, IRQ4 and IRQ5 to build an example test case.

4.1. Test Case Design

We will design a test case that shows the interrupt controller behavior when nested interrupts are generated using IRQ lines IRQ3, IRQ4 and IRQ5. This test case will illustrate what happens when an interrupt is being serviced and another one with higher priority is generated. It also illustrates what happens if a lower priority interrupt is generated. The test must reflect how the interrupt controller keeps track of the interrupts being serviced and the interrupt requesting the CPU attention.

The test case that will be implemented is the following: *Activate IRQ5; while its ISR is being executed, activate IRQ3. Finally, while the ISR that is serving IRQ3 is in action, activate IRQ4.*

The first thing to do is modifying the ISR associated to IRQ5 and IRQ3 because while they are being executed lines IRQ3 and IRQ4 must be activated respectively. The ISR associated to IRQ4 also needs to be modified to show interrupt controller information. The first interrupt is generated from the main program using the function *execAllCQ()*. This function is called inside the mentioned ISR to execute commands from the queue. Listing 1 shows how the modified ISR for IRQ3 looks like.

```
void interrupt irq3Handler()
{
   disableFLIRQ(3); // Reset edge/level register
   cprintf("Serving Interrupt 3...\r\n");
   execAllCQ(&queue);

   cprintf("Leaving Interrupt 3...\r\n");
   outportb(0x20,0x20); /* Send EOI*/
}
```

List. 1. Source code for a modified ISR.

The ISR calls the function *disableFLIRQ()* on its entry to reset the edge/level registers to its original state. This function takes as parameter the corresponding IRQ number. Then, the ISR tries to execute commands from the *commandQueue* object *queue* (declared previously in the program) using the function *execAllCQ()*. Once executed the commands the End Of Interrupt (EOI) signal must be sent to the interrupt controller. On entry and exit of the ISR, some messages are printed to help keep track of the execution flow.

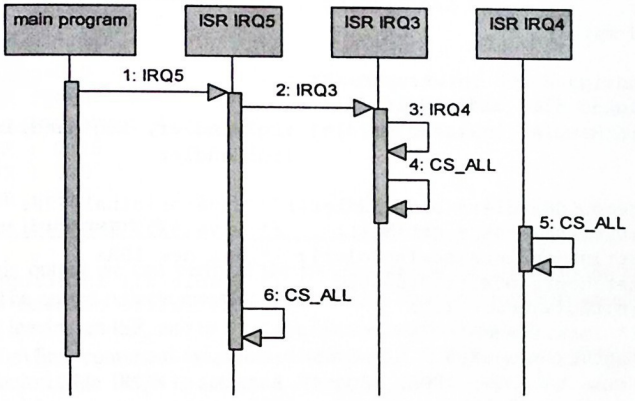

Fig. 1. Sequence diagram showing the expected execution flow. Arrows indicates execution control transfers when commands are executed. Markers are not shown in the diagram.

The hardest part of the process is choosing the right commands to achieve the expected interrupt sequence (Figure 1). The first action is activating line IRQ5, so this must the first command in the queue. This command is executed from the main() function. The next command activates IRQ3, in this way we can observe how the interrupt hardware behaves when a higher priority interrupt line is activated while a lower priority line is being served. In this case the ISR associated with IRQ3 must be executed immediately. The third command is IRQ4, so we can observe what happens when a lower priority interrupt line is activated while a higher priority one is being served. This request must wait until all interrupts with higher priority have been served. At this point the state of the interrupt controller is shown (CS_ALL), to take a look at the way the controller knows which lines are being served and which are requesting attention. The next command in the queue is a marker. This marker ends the execution of the function execAllCQ() inside the IRQ3 ISR. Once finished the

execution of this ISR the line IRQ4 must be served because its priority is higher than that of IRQ5. In this point we read the interrupt controller state once again (CS_ALL) and insert another marker to end up IRQ4 ISR execution. Then, we are back in IRQ5 ISR, now we can insert a marker to end up execution or just insert none because we are already at the end of the queue, so we will finish execution anyway. Once finished the IRQ5 ISR we return to the main() function.

4.2. Test Case Implementation

First, the source code of the program must be written (Listing 2). This source code must be compiled and linked with the *IntrGen.lib* library obtaining an executable file.

```
/* IntrTest.c: Test case for the interrupt hardware */
#include "IntrGen.h"
commandQueue queue;        /* command queue declaration*/
/* ISRs for IRQ3, IRQ4 and IRQ5 */
  ...          /* See Listing 1 */
void main()
{
   unsigned int interruptMask;
   intHandler oldIntHdler[3];
   intHandler newIntHdler[3]={ irq3Handler, irq4Handler,
                               irq5Handler              };

   readOldHandlers(oldIntHdler);/* Save original ISRs      */
   interruptMask = getIMR();    /* Save 8259 IMR register  */
   setIntHandlers(newIntHdler); /* Set new ISRs            */
   setInterruptMask(0x0000);    /* Enable all interrupts   */
   initCQ(&queue, 10);          /* Initialize command queue*/
   /* Insert command in the queue                          */
   postCQ( &queue,8    /* Number of commands               */
          ,IRQ5, IRQ3, IRQ4 /* Activate IRQ5, IRQ3 & IRQ4*/
          ,CS_ALL      /* Show 8259 state                 */
          ,C_BRK       /* Marker                          */
          ,CS_ALL      /* Show 8259 state                 */
          ,C_BRK       /* Marker                          */
          ,CS_ALL      /* Show 8259 state                 */ )
   execAllCQ(&queue);  /* Start commands execution        */
   setInterruptMask(interruptMask);/* Restore original IMR */
   setIntHandlers(oldIntHdler);    /* Restore original ISRs */
```

List. 2. Test case source code.

The program firstly includes the IntrGen.h header file. Next, a commandQueue object named *queue* is declared and the modified ISRs for the used interrupt lines are defined. On entry, *main*() save the addresses of the original ISR that will be replaced (IRQ3, IRQ4 e IRQ5) and the interrupt controller IMR register. These values are needed to restore the interrupt mechanism to its original state when the test is over. Then, we modify the *Interrupt Vector Table* (IVT) to point to the new ISRs, enable all

interrupt lines and construct the commandQueue object using the function *initCQ()*. At this point, we insert the commands for getting the desired sequence (*postCQ()*) and the execution is started calling the function *execAllCQ()*. Once finished the execution of commands we proceed to restore the IVT and the IMR register to its original state.

4.3. Test Case Execution Analysis.

The program execution produced the following output:

```
(0) Activating Interrupt 5
Serving Interrupt 5...
(1) Activating Interrupt 3
Serving Interrupt 3...
(2) Activating Interrupt 4
(3) ISR : 0x28
    IRR : 0x10
    IMR : 0x0
Leaving Interrupt 3...
Serving Interrupt 4...
(5) ISR : 0x30
    IRR : 0x0
    IMR : 0x0
Leaving Interrupt 4...
(7) ISR : 0x20
    IRR : 0x0
    IMR : 0x0
Leaving Interrupt 5...
```

From this output we can verify if the program execution adjusts to the expected behavior. The output reflects messages printed each time an interrupt is generated, on entering or leaving an ISR, and to show the interrupt controller state.

When the first command is executed (command 0) calling *execAllCQ()* at the *main()* function, line IRQ5 is activated. Then, its corresponding ISR takes control showing a message and calling the function *execAllCQ()* to activate the IRQ3 line (1). As we can see from the output, this interrupt is served immediately because of its higher priority. Inside the IRQ3 ISR line IRQ4 is activated (2), this interrupt is not served at this point because of its lower priority. This fact can be seen from the output. The next command executed while IRQ3 line is being served shows the interrupt controller state (3). The ISR register presents bits 3 and 5 set, reflecting that interrupt IRQ3 and IRQ5 are being served, which is correct. Register IRR presents bit 4 set, indicating that line IRQ4 is requesting the CPU attention. The IMR register value is not important in this test case; it keeps the same value all the time. After this, the next command, a marker (4), finish the execution of function *execAllCQ()* in IRQ3. The marker does not produce any output, but a message is printed when the ISR end up its execution.

The highest priority interrupt line waiting for attention is IRQ4, so its associated ISR is executed. This ISR executes the next command that shows the interrupt controller state once again (5). ISR register reflects that lines IRQ4 and IRQ5 are being served. The IRR register shows no line requesting attention. Right here, a

marker command ends up the command execution inside this ISR (6). At this point we are back to the ISR serving IRQ5. The next command to execute asks the interrupt controller for information about its state (7). We can see at the output that only IRQ5 is being served. There are no commands left in the queue so after showing a message execution control is back to the *main()* function. The program output proves that the interrupt hardware is working as expected when nested interrupts occur.

5. Conclusion

With this work we have obtained a very useful software component for the debugging and tuning of interrupt based systems. The main feature of this component is that it allows the generation of interrupts at the hardware level, but in a synchronous way under the software control. It is important to highlight that generated interrupts are not "simulated", but real hardware interrupts and go through all the PIC logic. As result, our component can be very useful for debugging and verification of the hardware interrupt handling mechanism itself, something that can not be done with the current tools. These tools use to "simulate" hardware interrupts using software interrupts and therefore, ignoring the interrupt hardware. As an example of the possibilities our component provides, impossible to achieve with other existent debugging modules, we can mention its utilization for getting the major temporal characteristics of a real-time system like the measurement of the interrupt latency without needing additional hardware. Additionally, the component has been used for teaching the interrupts topic at the computer architecture undergraduate courses on several computing majors.

References

1. Ball, S.: "*Embedded Microprocessor Systems*, Real World Design" Newnes 1998.
2. Dijkstra E. W., "*Cooperating Sequential Processes,*" Technical Report EWD-123, Technological University, Eindhoven, The Netherlands, (1965).
3. Fidge, C and Cook, P.:"*Model Checking Interrupt-Dependent Software*", In Proceedings 20th Asia-Pacific Software Engineering Conference (APSEC 2005), 2005.
4. Guangming, C., Minghong, L.: "*The Definition of an Interruptible Petri Nets*", Proc. of the 2nd International Conference on Embedded Software and Systems (ICESS'05), 2005.
5. Intel: "*IA-32 Intel Architecture: Software Developers Manual Vol. 3 System Programming Guide,*" 2002.
6. Intel: "*82371FB (PIIX) and 82371SB (PIIX3) PCI ISA IDE Accelerator*", Intel Developer's Insight CD-ROM, 1997.
7. Leyva-del-Foyo, L. E., Mejia-Alvarez, P., de Niz, D.: "*Real-Time Scheduling of Interrupt Request over Conventional PC Hardware*", Proc. of the Seven Mexican International Conference on Computer Science (ENC 2006), IEEE Computer Society 2006.
8. Leyva-del-Foyo, L. E.; Mejia-Alvarez, P.; de-Niz D.: "*Aligning Exception handling with design by contract in embedded real-time systems development*", Proceedings of the IEEE ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems", 2005.
9. Leyva-del-Foyo, L. E, Mejia-Alvarez P.: "*Custom Interrupt Management for Real-Time and Embedded System Kernels,*" Proceedings of ERTSI: Embedded Real-Time Systems Implementation Workshop in the IEEE RTSS04, December 5-8, 2004 Lisbon, Portugal.

10. Leyva-del-Foyo, L. E.: "*Introducción a PARTEMOS.*" I Seminario Cientifico del Departamento de Computación. Universidad de Oriente, Julio de 2003.
11. McGivern, Joseph: "*Interrupt Driven PC System Design*", Annabooks/Rtc Books, 1998.
12. Regehr, John: "*Thread Verification vs. Interrupt Verification*", in Proceedings of the 2006 Federated Logic Conference (FLoC 2006), Seattle, August 10 - 22, 2006.
13. Roberts, D. : "*Interrupt Behavior in Windows NT,*" Dr. Dobb's Journal, April 1998.
14. Rosenberg, J. B.: "*How Debuggers Work: Algorithms, Data Structures, and Architecture*", Wiley, 1996
15. Thane, H., Hansson H.: "*Handling Interrupt in Testing of Distributed Real-Time Systems*", 6th Int. Conf. on Real Time Computing Systems and Applications (RTCSA99), 1999.