A Debugger RTOS for Embedded Systems

Tankut Akgul, Pramote Kuacharoen, Vincent J. Mooney and Vijay K. Madisetti
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332
U.S.A.
Email: {tankut, pramote, mooney, vkm}@ece.gatech.edu

Abstract

In this paper, a software debugging mechanism for embedded systems is presented. The debugger is a dynamically loadable and linkable module of the operating system. The methodology presented in this paper provides automatic error detection, classification and location capabilities for a set of algorithmic errors. An example implementation of our approach is given for debugging an integer divide-by-zero error.

1. Introduction

Testing and debugging of software constitutes a significant amount of the development time. Debugging time itself can account for up to 50% of the total time required for software development [1]. Experience with software development has shown that software is often delivered late, although it is still functionally incorrect.

Today's debugging is mostly dependent on the programmer's experience. In order to remove a bug from software, the programmer first has to detect the erroneous behavior in the program and classify the error, i.e., he or she has to find out what type of error may be the cause of the abnormal situation. Then, he or she has to locate the exact place of the error in the code. Finally, the programmer has to modify the code piece at the error location to remedy the situation. The problem is that this is an iterative, trial and error process, that is, more than one pass through each step may be needed in order to successfully remove the bug from the program. What is worse is that clearing a bug may create new bugs in the code.

Another challenge arises if we consider debugging embedded software. This is because of the difficulty of testing the code within the embedded environment in which the program will eventually be running. Although the earlier stages of the development can be accomplished on a separate host machine, the final code development has to be done on the target embedded system in order be able to capture the errors related to the target platform, such as an error which only occurs during collecting samples from an attached sensor in the embedded system. It is an unaffordable luxury to run a powerful debugger together with other application(s) and possibly with a complex operating system on the target embedded platform because of the scarceness of the available resources. On the other hand, if the debugger is run on a host machine communicating with the target, it is hard to collect state information with necessary detail from a deeply embedded target system.

In order to ease the debugging process for the programmer, several software-debugging techniques have been researched. One of the efforts is called *relative debugging* [2]. This approach relies on the automatic comparison of the buggy code with a correct reference code. However, in this method the drawback is that the correct version of the code may not be always available.

There are also other approaches where the software developer is forced to avoid errors during coding. This is provided by constraints defined inside the program code in the form of *assertions* [3]. Assertions are linguistic constructions which allow either run-time checking or compile-time checking of constraints defined in the programs. Going one step further, some logic programming languages have been developed depending on the assertion methodology such as Godel [4] and Mercury [5]. Although assertions can get rid of a set of common user errors, they are not fully efficient as they still do not cover a large set of errors that cannot be detected by linguistic constructions. This is also true for logic languages which also require dedicated compilers.

Two other related methods for simplifying the job of locating the error in the program are *divide-and-query* [6] and *program slicing* [7]. The divide-and-query algorithm recursively searches a computation tree representing the target program until a bug is found. Program slicing, on the other hand, is a method which specifies executable statements (*program slices*) that can influence a value of a variable, so that when an incorrect value is attained by that

variable, the error search space is reduced to the relative program slice. However, these methods do not work well for multi-threaded applications running on a Real-Time Operating System (RTOS) where complex interactions between program parts hinder the process of searching computation trees and specifying exact program slices.

There are also different techniques that have been developed for debugging embedded systems. In one technique, a part of the debugger tool that is called the *debug monitor* (which is responsible for providing debugging features and downloading the executable program to the target system) and the application program run on the target system. Since the debug monitor runs on the target platform, it has the ability to collect information about the target system internals. However, a debug monitor is usually burned on a ROM in the target platform, which prevents the monitor from being updated later on. Also, it is not easy to remove the ROM monitor after system testing is done, so that the ROM monitor is usually kept in the final product at an extra cost. The rest of the debugger tool, which is the user interface, runs not on the application platform but instead on a different platform which is usually a computer with a display terminal. Since the platforms are separate, a communication link is needed between the two. A UART and a suitable line driver in the target platform are often needed for this [8].

Another technique is to use in-circuit emulators (ICEs), which are hardware units containing real-time event detection, real-time tracing and memory emulation, all integrated behind a unified user interface. The problem with ICEs, however, is that they lag behind the processor production time and become useless as the processor version changes. Furthermore, ICEs are usually expensive.

Finally, the last technique, which is becoming increasingly common, is the debugging technique using dedicated processor pins. Most of the modern processors support some dedicated pins by which a debugger program can observe some internal signals of the processor and extract debug information from interpretation of these signals. In this technique, the whole debugger software runs on a host machine and communicates with the target processor via these dedicated pins. In the latest processors this type of debugging is done via JTAG pins [9]. However, for deeply embedded cores, it is almost impossible to reach the internal signals of a processor and extract detailed debugging information.

Therefore, it is desirable to come up with an efficient mechanism which solves the problems related with the debugging of the embedded software. Also, it is highly desirable to develop an easy method which will take the burden away from the software engineer by automating the error detection, classification and location steps not only for single-threaded applications but also for multi-threaded ones.

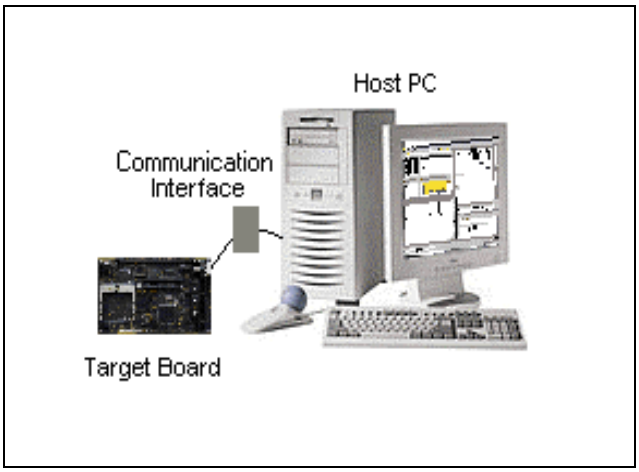

Figure 1. Typical debugging platform for Debugger RTOS applications

In this paper, we explain a new approach which addresses the aforementioned issues by automating the debugging process for multi-threaded applications running on embedded systems. Specifically, we propose a unique and novel RTOS based debugging methodology for embedded systems where the debugger is a dynamically loadable module of the operating system. For this reason we give our methodology the name *Debugger RTOS*. Our Debugger RTOS brings much more control over errors by providing both an automatic error detection, classification and location mechanism as well as complete state information about the system both at the application level and the operating system level. Since the debugger runs on the target system rather than on a host computer, it is possible to collect information about the internals of the target system no matter how deeply embedded it is. On the other hand, since the debugger is dynamically loaded, it does not consume system resources while not in the debugging session. However, instrumentation code – i.e., code inserted into the kernel to detect errors and then load the debugger module – must be present at all times. Thus, during normal operation, the overhead is only due to the instrumentation code. Moreover, there is a flexibility brought by a dynamically loadable debugger module: namely, the debugger module can be improved by additional functionality and integrated into the system at any point in time. In other words, our methodology combines the advantages of ROM based debugging and JTAG based debugging by providing necessary detail together with flexibility and low cost.

2. System model

A typical debugging platform for the Debugger RTOS applications is shown in Figure 1. The Debugger RTOS

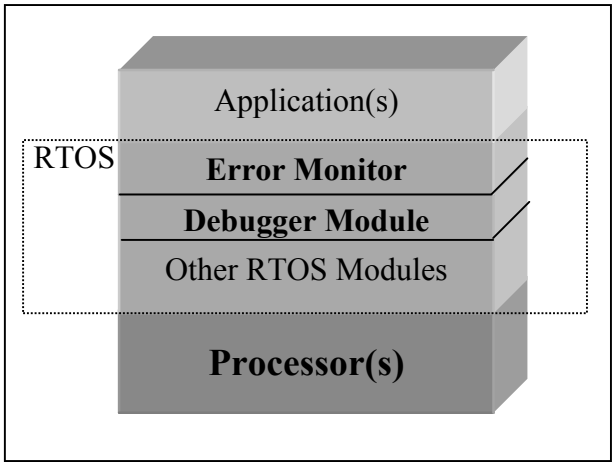

Figure 2. Debugger RTOS structure

and the application(s) reside on the target system. The debugger software is completely inside the RTOS code and runs on the target processor. The host computer, on the other hand, runs the user interface only. The communication between the host and the target is established by any suitable communication interface.

The debugging feature of the Debugger RTOS is composed of two parts: the *monitoring part* (error monitor) and the *debugging part* (debugger module) (Figure 2). The error monitor is responsible for the detection of errors in the program and for the instantiation of the debugging part. The error monitor is always resident inside the RTOS in the form of instrumentation code. The debugger module, on the other hand, is loaded dynamically by the error monitor when an error condition is met. The debugger module is responsible for providing the user with data about the exact place of the error (error location), the type of error (error classification), and the debugging features such as displaying register values, memory dump and giving detailed state information about internal RTOS structures.

In order to get a better view of what composes the error monitor, we should first state some definitions. *Failure* can be defined as a deviation of the system or component from its intended function defined in its specification. An *error* (*bug*), on the other hand, is an unwanted state of program flow which does not necessarily result in a failure. Failures can be caused by a wide variety of errors. In our model, we define two main types of errors which may cause failures in programs.

The first type of error is the error caused by software trying to execute an instruction or a function in a way not supported by the underlying hardware. These errors can be caught by hardware assistance in terms of *hardware exceptions* which are instruction-related interrupts during the execution of the program. One typical example of an exception is an alignment exception which is usually caused by executing a load/store instruction with misaligned operands. The second type of error in our model is an *algorithmic* error which is caused by an incorrect specification or design of the program and which cannot be caught by hardware. A typical example for this type of error is assigning an incorrect value to an intended variable.

In the case of hardware exceptions, the error monitor does not need to provide a detection mechanism as these exceptions themselves can be detected by hardware. Therefore, if there should happen to be a hardware exception, the error monitor is only responsible for the dynamic instantiation of the debugger module. On the other hand, in case of algorithmic errors, the error monitor is responsible for both the detection of the error and the instantiation of the debugger module. Algorithmic error detection is achieved by the probes inserted inside the RTOS code. The programmer can enable or disable these probes according to his needs. This monitoring method is completely different from traditional monitoring methodologies where probes are inserted directly inside the application code. Probes inside the RTOS keep the application code unchanged and brings transparency between the user and the system. Another advantage of RTOS probes comes with the concept of *RTOS-aware debugging* [10]. RTOS-awareness comprises two basic sets of resources: RTOS state knowledge and tracking of code execution on a thread, rather than on a procedural, basis. RTOS probes can have a complete access to RTOS internals such as task control blocks (TCBs), the ready task table, semaphores and mailbox structures. Access to these RTOS internals allows the Debugger RTOS to have wider control over the system, therefore providing much more detailed information in case of a failure condition. Moreover, by RTOS-aware error monitoring, it is possible to differentiate between the threads. Conventional debuggers fail miserably in multi-threaded environments where there is no way to differentiate between threads: thus, for instance, a breakpoint set on one thread affects all instances of the code.

The debugger module is a dynamically loadable and linkable part of the RTOS. In case of error free execution, the debugger module is never loaded; this prevents consumption of extra system resources. When the debugger module is not activated, it is kept in a storage unit external to the target system. This storage unit may be the hard drive of the host computer shown in Figure 1. As soon as an error condition occurs (either a hardware exception or an algorithmic error), the error monitor loads the debugger module from the external storage unit into the memory of the target system and the debugger module is dynamically linked to the RTOS. The dynamic loading and linking mechanism of the debugger module is explained in the following section. This method is similar to the method described by Kuacharoen, et al [13].

Module Table

Function name	Address
Debugger	0xFF000000

Global Variable Table

Global Variable	Address
Current_Task_ID	0xFF00A000
TCB_List	0xFFF00B00
…	…

Figure 3. Module table and global variable table structures

2.1. Dynamic addition of the debugger module to the Debugger RTOS

The loading and linking process of the debugger module at run-time is composed of three steps as follows:

1. If the debugger module is not already in the memory, it has to be brought into memory first.
2. There may be external references inside the debugger module that must be resolved when it is linked to the kernel executable.
3. The debugger module has to be initialized in order to completely integrate with the kernel.

In order to make the loading and linking processes much more flexible, we decided to implement API functions that take care of the above three steps. These functions are named *load_module_debugger()* and *init_module_debugger()*.

The load_module_debugger() routine handles step 1 of the loading and linking mechanism by allocating memory space and loading the pre-compiled debugger module into this memory location. The first place that is looked for loading the debugger module is a free memory block in the heap section of the memory. If there is no such block available in which to fit the debugger module, another module of the RTOS such as the scheduler is replaced by the debugger module. We assume that after an error is detected, the programmer will not want to continue the execution but will modify the code in order to remove the bug reported by the debugger module. Then, he or she will re-compile the code with the original components and re-execute the program. Therefore, it is completely safe to perform a module replacement as described above.

The second step of the loading and linking mechanism is one of the most important and tricky steps. To achieve this, we maintain a *global variable table* (Figure 3) which holds the addresses of global variables that may be referenced within the debugger module. Furthermore, most of the compilers today generate code so that global variables local to a module are referenced by an indirection from an absolute address. Therefore, these references must be updated according to the target memory space where the module is loaded. However, this updating degrades performance considerably due to further processing of binary executable at run-time [11]. Instead, we compile the debugger module so as to generate position independent code in advance. Consequently, all of the references inside the debugger module become relative addresses and thus independent of the memory address where the module is located.

Finally, the init_module_debugger() function takes care of the third step of the loading and linking mechanism. We construct another table, *module table* (Figure 3), which has an entry for referencing the debugger module inside the RTOS kernel. This entry consists of a function pointer. The init_module_debugger() function updates this pointer in order to make it point to the debugger module. The init_module_debugger() function also handles the initialization necessary for the debugger module to work properly.

3. Experimental setup

We implemented an experimental setup similar to the setup shown in Figure 1 for testing our methodology. The setup consists of the Debugger RTOS which runs on a

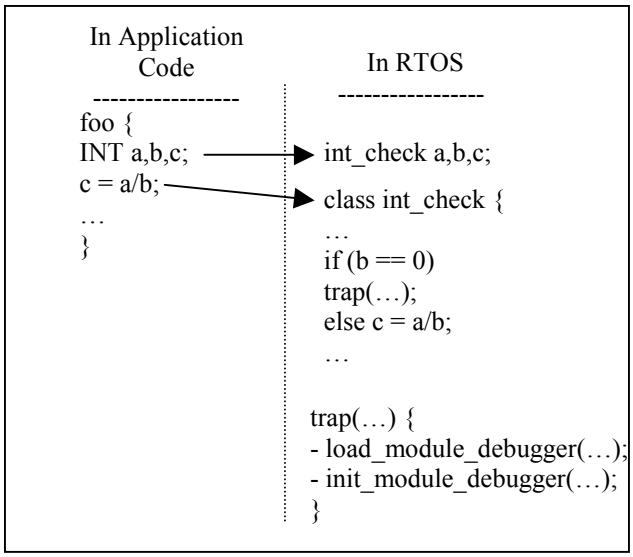

Figure 4. Diagram of the divide-by-zero error detection mechanism

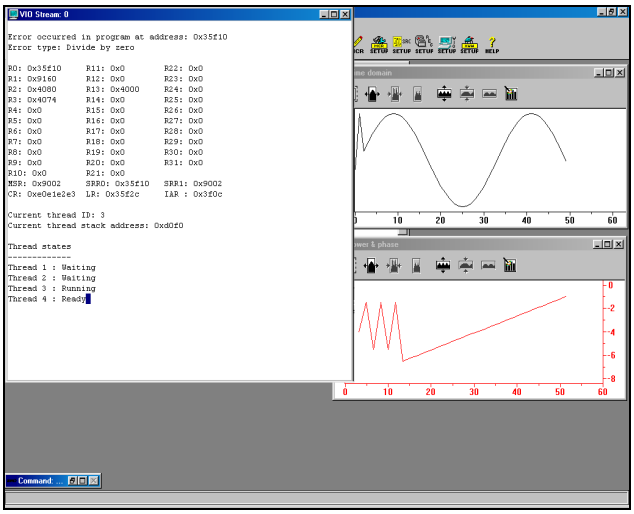

Figure 5. A snapshot of an example application for testing Debugger RTOS with the detection of the divide-by-zero error

PowerPC 860 processor. We instrumented the Debugger RTOS with the error monitor code. The debugger module is a custom made small program which reports the cause of an error, the location of the error in the program, and dumps the state information to the screen. The communication between the host computer and the target board is established by a JTAG interface. Note that the JTAG interface is not used for JTAG based debugging in its original way; rather, it is used only for data exchange between the host and the target in our experimental setup. Therefore, we are not subject to the disadvantages of the JTAG based debugging as outlined before. The reason we chose JTAG is that we wanted to show that our methodology can work fine with the latest available technology. In the following paragraphs we show an example of detecting an integer divide-by-zero algorithmic error[1] in the application code.

The probe for the detection of a divide-by-zero error is implemented in the following way. Class types of int_check, float_check and double_check are defined for all data types in the RTOS, which include an overloading of the division operator ("/") using the C++ language. Thus, the ordinary division operator gains a functionality of checking for the divide-by-zero condition. Then, these class types are mapped into the corresponding generic data types such as INT, FLOAT and DOUBLE. Thus, when the application programmer uses a division operator ("/") with operands of these generic data types, he or she actually uses the modified division operator, which detects the divide-by-zero error condition.

[1] This error has hardware support on PowerPC processors in terms of a hardware exception which is generated for floating point division only.

The integer divide-by-zero error detection mechanism is summarized in the diagram in Figure 4. The generic data type INT is converted into int_check class type (which is defined in the RTOS code) during the compilation of the source code. When a divide-by-zero error condition occurs (which is checked by the int_check class type implicitly), a *trap()* function is called, which in turn calls the load_module_debugger() and init_module_debugger() functions.

We ran a multi-threaded graphics application to test the Debugger RTOS. We used the CrossView Pro Tool [12] as our user interface on the host. The experiment application consists of three threads. Two threads read two different sets of floating-point voltage samples from a file record and display the waveforms on the screen. The third thread rounds the voltage values by assigning them to integer variables and computes the ratio of them by dividing one of the rounded samples by the other. When one of the divisor voltage samples is between –1 and +1, it is rounded to zero, so that a divide-by-zero error condition occurs. A snapshot of the user interface running on the host and a sample output from the debugger module are shown in Figure 5 and Figure 6, respectively. Note that the fourth thread, of which status is shown as "ready" in the sample output, is the idle thread created by the operating system in the system initialization phase.

```
Error occurred in program at address: 0x35f10
Error type: Divide by zero

R0: 0x35f10      R11: 0x0         R22: 0x0
R1: 0x9160       R12: 0x0         R23: 0x0
R2: 0x4080       R13: 0x4000      R24: 0x0
R3: 0x4074       R14: 0x0         R25: 0x0
R4: 0x0          R15: 0x0         R26: 0x0
R5: 0x0          R16: 0x0         R27: 0x0
R6: 0x0          R17: 0x0         R28: 0x0
R7: 0x0          R18: 0x0         R29: 0x0
R8: 0x0          R19: 0x0         R30: 0x0
R9: 0x0          R20: 0x0         R31: 0x0
R10: 0x0         R21: 0x0
MSR: 0x9002      SRR0: 0x35f10    SRR1: 0x9002
CR: 0xe0e1e2e3   LR: 0x35f2c      IAR : 0x3f0c

Current thread ID: 3
Current thread stack address: 0xd0f0

Thread states
-------------
Thread 1 : Waiting
Thread 2 : Waiting
Thread 3 : Running
Thread 4 : Ready
```

Figure 6. A sample output from the debugger module

The current version of the debugger module is capable of classifying the error, indicating its location in the program code and dumping state information. As it can be seen from Figure 6, it is very easy to acquire valuable state information about RTOS internals even by a simplistic first version of the debugger module. Table 1 shows the sizes of the error monitor, the debugger module and the other RTOS modules in terms of the number of C lines.

Table 1. Debugger RTOS modules

	Number of C lines
Error monitor	273
Debugger module	168
Other RTOS modules	2686

4. Conclusion

Debugging software is an inevitable and arduous task. Embedded systems provide the additional challenges of limited visibility of the system through a small number of inputs and outputs. Today's debugging methodologies for embedded systems can be inadequate for overcoming this problem with a low cost and flexible solution. The Debugger RTOS brings an efficient and flexible software mechanism for debugging embedded systems. The capability of automatic detection, classification and location of an extensible set of algorithmic errors adds intelligence to the debugger, including errors thought of after writing the application code. Our Debugger RTOS migrates the debugger from the application level to the operating system level. This eases the job of gathering program state information for the debugger in terms of RTOS level entities such as threads or interrupts. Since the error monitor resides inside the RTOS, the error detection mechanism is completely transparent to the application programmer. Furthermore, by the help of dynamic loading and linking mechanism of the debugger module, more efficient use of memory can be achieved. This also helps to update the debugger module separate from the other parts of the RTOS and integrate it with the system easily in the future.

The first step as a future work is to improve the debugger module with more sophisticated debugging features such as program tracing, breakpoint insertion and variable query. In order to be able to do that, the debugger module has to be run concurrently with the application(s) and other RTOS modules during the debugging session. Therefore, an intelligent selection scheme is necessary to determine which other RTOS module(s) can be replaced by the debugger module if there is not enough memory available on the embedded system until the debugger session completes.

5. Acknowledgements

This research is funded by the State of Georgia under the Yamacraw initiative and by NSF under INT-9973120, CCR-9984808 and CCR-0082164.

We also acknowledge software donations from Mentor Graphics and Synopsys as well as hardware donations from Sun and Intel.

6. References

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Efficient Debugging With Slicing and Backtracking", *Software Practice & Experience*, June 1993, 23(6), pp. 589-616.

[2] D. Abramson, I. Foster, J. Michalakes, and R. Sosic, "Relative Debugging: A new paradigm for debugging scientific applications", *The Communications of the Association for Computing Machinery*, November 1996, 39(11), pp. 67-77.

[3] G. Puebla, F. Bueno, and M. Hermenegildo, "A Framework for Assertion-based Debugging in Constraint Logic Programming", *Proceedings of the International Conference on Principles and Practice of Constraint Programming,* number 1520 in LNCS, June 1993, pp. 472-473.

[4] P. Hill, and J. Lloyd, *The Godel Programming Language,* MIT Press, Cambridge MA, 1994.

[5] Z. Somogyi, F. Henderson, and T. Conway, "The Execution Algorithm of Mercury: An efficient purely declarative logic programming language", JLP 29, October 1996, pp. 1-3.

[6] Shapiro, E. Y., *Algorithmic Program Debugging,* MIT Press, Cambridge MA, 1983, (Ph.D Thesis).

[7] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Dynamic Slicing in the Presence of Unconstrained Pointers", *Proceedings of the 4th ACM Symposium on Testing Analysis and Verification*, October 1991, pp. 60-73.

[8] Diab-SDS web page, http://www.sdsi.com/developers/debugging.php3

[9] Joint Test Action Group (JTAG) web page. http://www.jtag.com/

[10] D. Shear, "Real-time operating systems", *EDN (European Edition)*, April 1994, 39(8), pp. 84-96.

[11] Levine, J. R., *Linkers and Loaders*, Morgan Kaufmann Publishers, October 1999.

[12] Tasking, Inc. web page. http://www.tasking.com

[13] P. Kuacharoen, T. Akgul, V. J. Mooney, V. K. Madisetti, "Adaptability, Extensibility, and Flexibility in Real-Time Operating Systems", *Euromicro Symposium on Digital System Design*, September 2001.