Esto NS-Debugger: The Non-stop Debugger for Embedded Systems

In-geol Chun*, Choon-oh Lee**, Duk-kyun Woo*
*Embedded S/W Research Division,
Electronics and Telecommunications Research Institute (ETRI)
161, Gajeong-dong, Yuseong-Gu, Daejon, 305-700, KOREA, igchun{dkwu}@etri.re.kr
** School of Engineering, Information and Communications University (ICU)
119, Munjiro, Yuseong-gu, Daejeon, 305-732, KOREA, lcol@icu.ac.kr

Abstract **— nowadays there is a lot of embedded software derived from the widespread of embedded systems. The characteristics of the embedded systems are the small size, limited resource and power. Because of that reason, the software in the embedded system is different from general computer software. Especially, finding bugs in the embedded software is the most different and complicate activity out of the embedded system development processes.**

In order to provide the high flexibility and the efficient deployment of various embedded systems, we developed a scalable and user-friendly debugger named Esto NS-Debugger (Esto Non-Stop Debugger). It supports general debugger functions and user-friendly interface like general software debugger. Especially as the time-sensitive and time-consuming application debugging is very difficult and boredom jobs, we also support a non-stop debugging function. In this paper, we focus on the design and implementation of the non-stop debugging function.

Keywords **— Embedded System Debugger, Non-Stop Debugging**

1. Introduction

Embedded systems are increasingly becoming a routine and integral part of modern society, and quality assurance requirements for such systems are quite demanding. Among various quality assurance methods, debugging is a direct and efficient method to remove bugs.

It is more important for developers to debug the software efficiently. In general, for the sake of debugging the software, most developers use a debugger such as GDB (Gnu Project Debugger). GDB supports useful functions to stop and examine a program for correcting it. However in some applications, it is not feasible for the debugger to interrupt the program's execution long enough for the developer to learn anything helpful about its behavior. If the program's correctness depends on its real-time behavior, delays introduced by a debugger might cause the program to change its behavior drastically, or perhaps fail, even when the code itself is correct. It is useful to be able to observe the program's behavior without interrupting it[1][2]. In GDB, the tracepoint facility is currently available only for remote targets. In addition, your remote target must know how to collect trace data. This functionality must be implemented in the remote stub; however, none of the stubs distributed with GDB support tracepoints so far.

In this paper, we developed the non-stop debugger for user convenience. This non-stop debugger consists of the user interface based on eclipse platform, the debugging engine, and the debugging stub.

2. Development of Embedded Software

As the characteristics of the embedded systems such as small memory and computing power, the application development can not be achieved in an embedded system. In general, we make use of a remote development environment to build application and find bugs. It consists of a host system for building and debugging applications and a target system that is distributed to a market as shown figure 1[3].

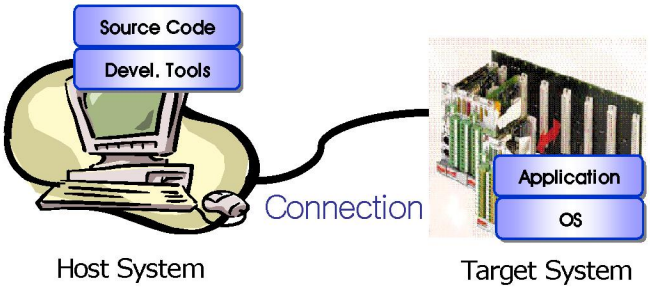


Figure 1. Remote Development Environment

ETRI has developed Esto (Embedded Software Toolkit) for convenient embedded software development. Esto is a visual software development environment based on Eclipse platform[4]. The powerful features of Esto enable software engineers to develop/analyze both general and domain-specific embedded software fast and easily under systematic project management. Esto provides IDE, Debugger, Monitor, Optimization and Analysis tool for accelerating embedded software development speed. With this tool, you can easily develop desirable applications.

3. Architecture of Esto NS-Debugger

With Esto, remote debugging is as simple as local debugging. Esto provides breakpoint debugging as well as tracepoint debugging that is useful for time-sensitive quality assurance.

Tracepoints are alterative breakpoints. The debugger which supports only traditional breakpoints stops the application that a user wants to debug and waits for user commands when the debugger meets breakpoint. If the application has much interaction with a user, these stop-and-resume processes are very boring and time consuming jobs to a user. Also if the application has a time sensitive operation, the suspension of the running application has an effect on the operating results. However tracepoints need no user interventions. A tracepoint is a breakpoint with a custom action associated with it. When the tracepoint is hit, it causes the debugger to perform the specified action instead of (or in addition to) breaking program execution. You can create a tracepoint with the toggle tracepoint command.

Using GDB, you can specify locations in the program, called *tracepoints*, and arbitrary expressions to evaluate when those tracepoints are reached. Later, you can examine the values those expressions had when the program hit the tracepoints. However none of the stubs interacting with GDB distributed so far. In this paper, we designed and developed not only a stub for remote embedded system debugging but also user friendly interface based on eclipse platform named Esto NS-Debugger.

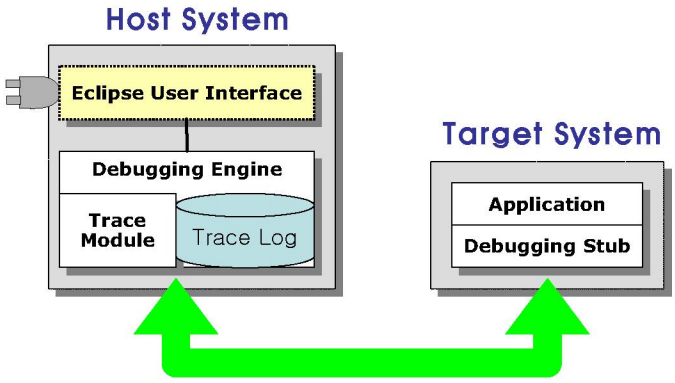


Figure 2. Architecture of Esto NS-Debugger

Esto NS-Debugger consists of 3 parts such as User Interface(UI), Debugging Engine(DE) and Debugging Stub(DS) as shown figure 2.

- User Interface : developed based on eclipse platform for user friendliness and integration with other tools. Especially various functions for tracepoints are implemented as eclipse plug-ins inherited from CDT (C/C++ Development Toolkit). We can use general debugging commands such as setting and unsetting breakpoint or tracepoint at source codes using UI.
- Debugging Engine : The debugging engine interacts with User Interface and Debugging Stub. We define some functions and protocols for tracepoints. For making scalable and flexible debugger, we developed it based on open source debugger named GDB.
- Debugging Stub : Debugging Stub is used for remote debugging as the counterpart for GDB in the host system. The Debugging Stub controls the debugging application of target system such as start, stop, resume the application and record the debugging information.

4. Implementation of Esto NS-Debugger

In this chapter, we focus on a design and implementation of tracepoint debugging. Above mentioned, Esto NS-Debugger has two types of debugging method. In the case of breakpoints, we don't design but use this in GDB while tracepoints is not implemented in GDB. The tracepoint debugging process divided into 3 steps. First user sets/unsets tracepoints and its action. Next user executes a debugger to record the status information of target system. Finally user can examine the recorded information for finding bugs through using replay scheme.

Figure 3 shows the actions of the first step. Especially it describes process for inserting new tracepoints. An user's input from controller module leads to an invocation of 'addTracepoints()' function of 'TracepointManager' class Then, the invocation goes to 'EstoTarget' class, which manages debugging processes for specific target. 'Estotarget' sends messages to the target system using MI protocol, and communication is in charge of 'MISession' class in CDT.

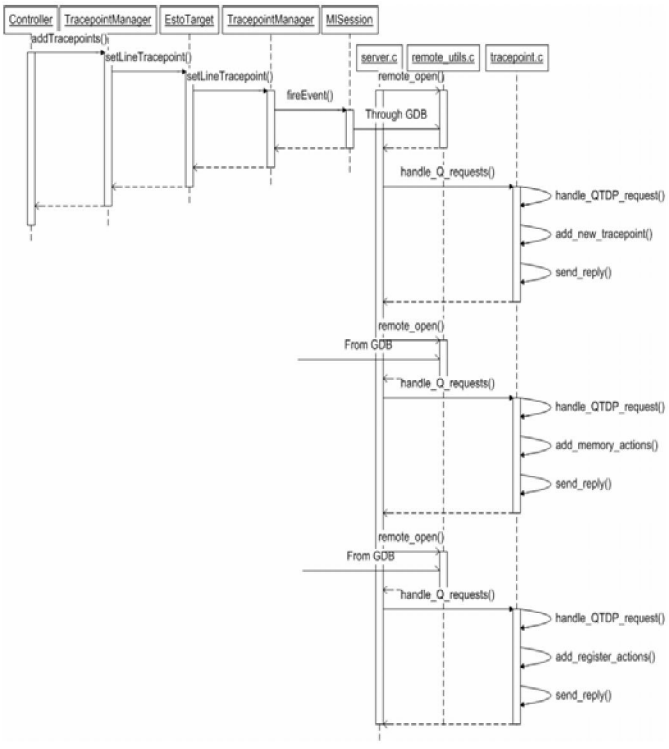


Figure 3. Sequence Diagram of Adding Tracepoints

MI messages are passed through GDB, and eventually the messages are sent to debugging stub derived from

gdbserver[5]. A message receiver of debugging stub is implemented in 'remote_utils.c'. 'server.c', which contains 'main()' function invokes 'remote_open()' function in 'remote_util.c' to take messages from GDB. Then, 'server.c' passes the message to 'tracepoint.c' which has all functions for handling tracepoint's operations.

To add a new tracepoint, three messages are passed from GDB to debugging stub. First is a message for the tracepoint initialization. It contains the location and the pass_count of the user defined tracepoint and it is handled by 'add_new_tracepoint()' function in 'tracepoint.c'. Second, the memory actions are added to initialized tracepoint through the next message, and 'add_memory_actions()' function is invoked. Finally 'add_register_actions()' function handles third message which contains register actions of the new tracepoint.

After adding all tracepoints, a user sends start commands to the debugger. At that time, application is downloaded to the target system automatically, and then the commands and action information for tracepoints are sent to the debugging stub. Finally the debugging stub invokes an application to record tracepoint information.

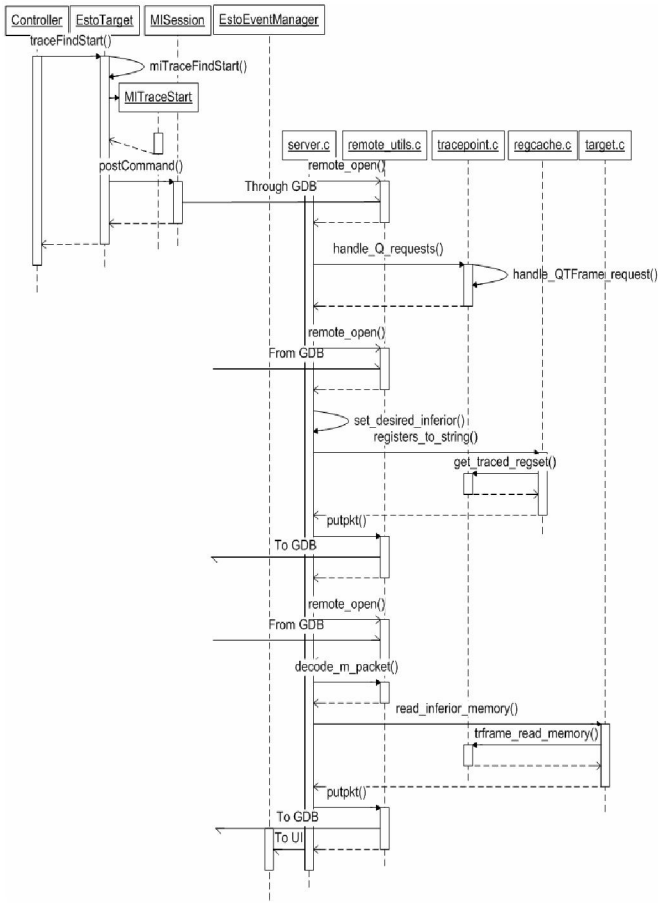


Figure 4. The Sequence Diagram of Replaying Tracepoints

Figure 4 shows the actions of the final step of tracepoint debugging. Following the trace, a user can examine the trace information through replay process for finding bugs. The sequence diagram for replaying is in Figure 4. When a user starts replaying, controller invokes 'traceFindStart()' function in 'EstoTarget' class, and MI message is sent to GDB. The message goes to the debugging stub and 'handle_QTFrame_request()' function in 'tracepoint.c' starts the replaying process on the target system parsing the message. 'handle_QTFrame_request()' function will be invoked when a user starts the replaying, and moves the current frame to be referred. After the current frame is set, memory and register results of current frame are gathered immediately. Results of registers are given by 'get_trace_regset()' function in 'regcache.c', and 'trframe_read_memory()' function in 'tracepoint.c' gathers memory results and returned them to 'server.c'. Returning messages are generated by 'putpkt()' function in 'remote_util.c' and passed to GDB.

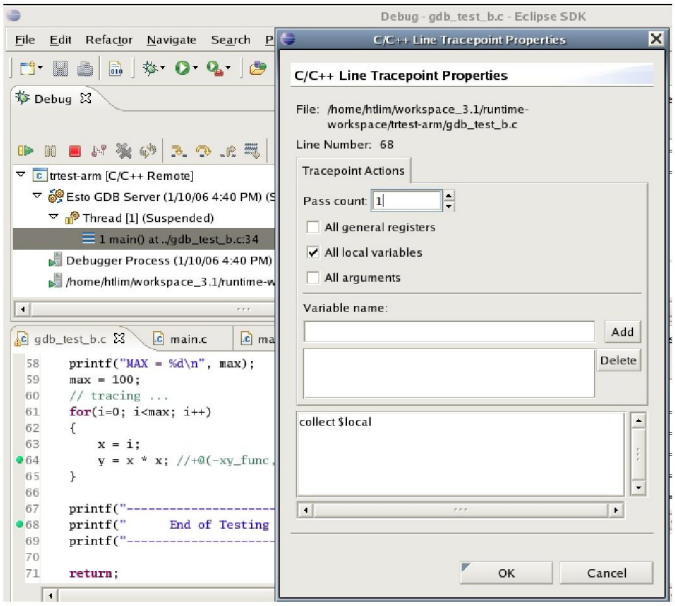


Figure 4. The screenshot of Esto NS-Debugger

Figure 4 shows the screenshot of Esto NS-Debugger. It is developed based on Eclipse CDT. A user can set/unset tracepoints and define the action for tracepoints as following steps.

① In a source window, right-click a line where you want to set a tracepoint and chooses toggle tracepoint in the shortcut menu.

② When Hit dialog box appears, it contains tracepoint actions tab. If you want debugger to break at that source line, specify the number of the pass count. The pass count means that the debugger stops when meet at the designated time.

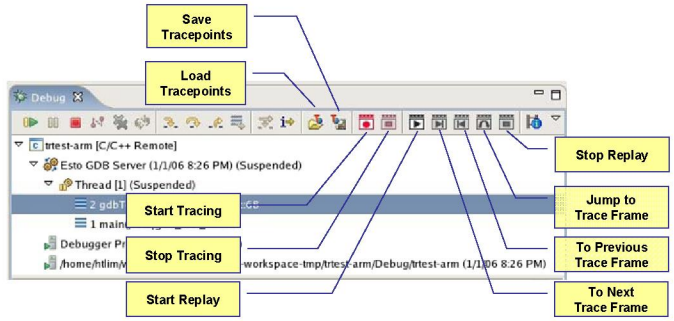
③ At this point, you can select items recorded using check box: all general registers, all local variables, and all arguments. If you want to record specific variable, you can type its name into variable name field. Then click add button.

④ Repeat step 3 until the entire variable recorded is specified.

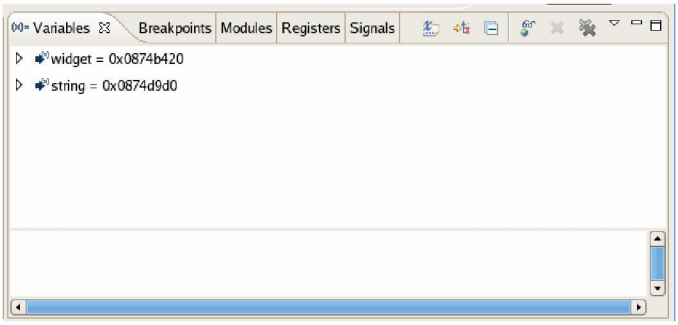
⑤ Click OK.

After completion of setting tracepoints, click "start tracing" icon. It makes the debugger start tracing. Then a user can

replay the tracepoint debugging and examine the gathering information of each debugging stack frame using forward and backward frame button. Figure 5 depicts tracepoint commands and the recorded results using replay.



(a) Tracepoint Commands



(b) Recorded Information

Figure 5. Screenshot of Tracepoint Command and Recorded Results

5. Conclusions

As embedded systems become more popular and widespread, various and complex software are used in them more and more. The software for embedded systems are similar to the legacy software, but embedded systems – such as PDA, MP3 player, set-top box and so on - have the limitation on resources and time so a general debugging method is not applicable. Especially, it is very difficult and time consuming jobs to debug embedded software.

In this paper, we proposed Esto NS-Debugger for embedded systems. The debugger is extended from open source (such as GNU debugger, Eclipse platform). GDB support a breakpoint based debugging but it is not applicable to time-sensitive and time-consuming embedded software exactly. In order to solve the problem, we propose a tracepoint based debugging method.

In the case of tracepoint, a debugger doesn't stop at tracepoint but collects selected information, such as variables, registers, memory, stack, etc. Then, you replay the debugging session to inspect collected information at any tracepoint. At this time, the prototype system is developed and we go on additional tests for stabilizing the debugger. In the future works, we focus on the efficiency and the reliability of the non-stop debugger.

REFERENCES

[1] Debugging with GDB : The GNU Source-Level Debugger, Richard M. Stallman, Roland H. Pesch, Stan Shebs, et al., GNU Press

[2] In-Geol Chun, Che-duk Lim, ES-Debugger : the flexible Embedded System Debugger based on JTAG technology , The 7th International Conference on Advanced Communication technology (ICACT2005), Feb. 2005

[3] Yaghmour K., Building Embedded Linux Systems, O'Reilly & Associate, 2003

[4] Myeong-Chul Park, Young-Joo Kim, In-Geol Chun, Seok-Wun Ha, Yong-Kee Jun, A GDB-Based Real-Time Tracing Tool for Remote Debugging of SoC Programs, 2006 International Conference on Hybrid Information Technology, Nov. 2006

[5] William Gatliff, Implementing a Remote Debugging Agent Using the GNU Debugger, 2001