OMA DM Based Remote Software Debugging of Mobile Devices*

Joon-Myung Kang[1], Hong-Taek Ju[2], Mi-Jung Choi[1], and James Won-Ki Hong[1]

[1] Dept. of Computer Science and Engineering, POSTECH, Korea
[2] Dept. of Computer Engineering, Keimyung University, Korea
{eliot,mjchoi,jwkhong}@postech.ac.kr,juht@kmu.ac.kr

Abstract. The mobile devices have gained much functionality and intelligence with the growth of network technologies, the abundance of network resources, and the increase of various services. At the same time they are also becoming complicated and related problems to services and resources of mobile devices frequently occur. Since it is not easy for the manufacturers to completely remove the software errors of the mobile devices before they sell them, the users face inconvenience caused by them and the credibility of the manufacturers also decreases. So far, no definitive method has been given to debug software errors of the sold mobile devices. In this paper, we propose a debugging method to find and correct the software errors of the sold mobile devices based on the Open Mobile Alliance (OMA) Device Management (DM) standard. We define the managed objects (MOs) for composing the execution image and design the management operations for collecting MOs at the central server. We present a system that we have developed based on the MOs and the management operations. We also present how to debug software errors with the execution image and JTAG debugger.

Keywords: Device Management, Software Debugging, Mobile Device Diagnostics, OMA DM, OMA DM DiagMon.

1 Introduction

Recently, the growth in ubiquitous and mobile computing systems has led to an early introduction of a wide variety of intelligent wireless and mobile network capable devices [1]. They have gained much functionality and intelligence as the hardware and software technologies are getting advanced. The mobile devices are becoming more complex continuously. The higher the complexity of a device becomes, the higher the possibility of errors in it [15].

* This research was supported by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Assessment)" (IITA-2006-C1090-0603-0045).

S. Ata and C.S. Hong (Eds.): APNOMS 2007, LNCS 4773, pp. 51–61, 2007.

It is very difficult for manufacturers of mobile devices to completely remove software errors before they sell them. Therefore, the sold mobile devices usually still have software bugs, which cause the device to self-reset, freeze, or have system malfunctions. The most common method in solving these problems is to reset the system, which is only a temporary solution. Hence, the users still have to face these on-going problems. Furthermore, the manufacturers cannot find the root cause of these software errors even when they are reported to the service center, since it is difficult to reproduce the same errors in experiments. Therefore, it is difficult to locate the source code's exact error of the software. This leads to a decrease in the credibility of the current mobile devices, and there will not be a solution to fix the software errors in the products that are to be manufactured in future.

In this paper, we propose a debugging method to find and correct the software errors of the sold mobile devices. Through this method, the software errors can be corrected by the manufacturers and updated software versions can be provided to the users via service centers, and the corrected software can be applied to the future mobile devices as well. This will alleviate the users' inconvenience and increase the mobile device's credibility.

One of the methods for debugging the software error is to dump the execution image when an error occurs and to make use of it. The execution image includes registers, stack, key events, and so on. This method is used in common software debugging. For example, we debug using 'core dump' in the UNIX system, and report the gathered errors and logs to the Microsoft server in the Windows system. We also use this debugging method to correct the software errors with the execution image. That is, the software errors of the mobile device can be fixed by using the dumped execution image when the device is automatically or manually reset.

It is difficult to collect the execution image of the sold mobile devices due to the large size of data, and low bandwidth and high error rate of the wireless network environment. Moreover, the process of collecting the images in the central server is very complex. That is, when a reset occurs, the execution image must be produced, and the reset must be reported to the central server, where the image needs to be gathered. When debugging the software error, the system information of the mobile device like CPU type, memory size, etc. is also required along with the execution image.

To satisfy these constraints, the Open Mobile Alliance (OMA) Device Management (DM) framework [2, 3], which is the international standard for the mobile device management, can be used. The DM protocol proposed by the OMA is appropriate for collecting large-scale data in a wireless network environment. In addition, the DM protocol includes the management operations needed for collecting the execution image from the mobile device. In the OMA DM framework, the system information of the mobile device has already been defined as the standardized managed objects (MOs) [4]. We only need to add new MOs to define all of the information for debugging in the OMA DM framework. Therefore, we defined the MOs for creating the execution image and designed the process for collecting it.

In this paper, we present the remote software debugging system for user mobile devices. This system can collect MOs and create the execution image to debug the software errors. The developed system consists of a client and server. The client collects the system data related to the defined MOs, while the server collects such data and uses it to correct the software errors with the JTAG debugger.

The remainder of this paper is organized as follows. Section 2 describes the OMA DM and the OMA DM Diagnostics and Monitoring standard [11, 12]. Section 3 describes the management architecture, management information and management operations. Section 4 presents the system development for validating our proposed solution. Finally, conclusions are drawn and future work is discussed in Section 5.

2 OMA DM and OMA DM DiagMon

In this section, we describe the specification of the OMA DM and the OMA DM DiagMon standard. We present bootstrapping, device description framework, and OMA DM protocol as well as the functions defined by the OMA DM DiagMon Working Group (WG).

2.1 OMA DM

OMA has been established by mobile operators, information technology companies, wireless equipment vendors, and content providers. It has defined the standard for wireless mobile terminals. The OMA DM WG is one of the major WGs in OMA. It has proposed how to define the management information for the mobile devices in the form of DM tree and how to manage the mobile devices remotely using DM protocol [5], which is an SyncML [6] based protocol aimed at providing remote synchronization of mobile devices. The OMA DM standard includes three logical components such as device description framework (DDF) [7], bootstrapping [8], and OMA DM Protocol [5]. DDF provides necessary information about MOs in device for the server. Bootstrapping configures initiative setting of devices. The OMA DM protocol defines the order of communicated packages by the server and client. Each device that supports OMA DM must contain a management tree [9], which organizes all available MOs in the device as a hierarchical tree structure where all nodes can be uniquely addressed with a uniform resource identifier (URI) [10]. The management tree is not completely standardized yet. OMA allows each device manufacturer to

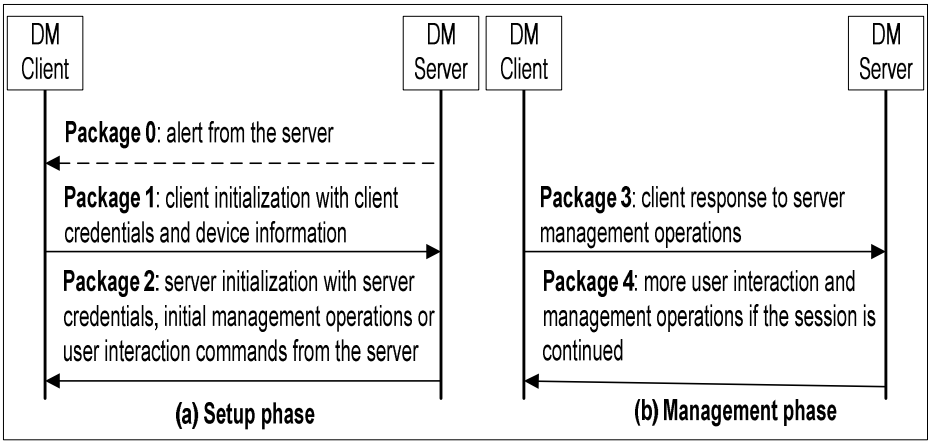

Fig. 1. OMA DM Protocol Packages

easily extend the management tree by adding new management functions in their devices by defining and adding the management nodes to the existing management tree. We show how this can be done in Section 3.

The OMA DM protocol consists of two parts as shown in Fig. 1: setup phase (authentication and device information exchange) and management phase [5]. The management phase can be repeated as many times as the server wishes, and the management session may start with package 0 (the trigger).

Table 1. OMA DM Commands

Feature	Description	OMA DM Command
Reading a MO content or MO list	The server retrieves the content from the DM Client or the list of MOs residing in a management tree.	***Get***
Adding a MO or MO content	A new dynamic MO is inserted	***Add***
Updating MO content	Existing content of an MO is replaced with new content	***Replace***
Removing MO(s)	One or more MOs are removed from a management tree	***Delete***
Management session start	Convey notification of device management session	***Alert***
Executing a process	New process is invoked and return a status code or result	***Exec***

Table 1shows the OMA DM commands, which are similar to SNMP operations [16, 17]. A management session is composed of several commands. The server retrieves the MO content or MO list from the DM client by the '*Get*' command. The server can add a new MO by the '*Add*' command. Moreover, the server can replace or delete by '*Replace*' or '*Delete*' command. The client can notify the management session by '*Alert*' command, while the server can execute a new process to the client by '*Exec*' command. We can design the diagnostic process by a composition of these commands.

2.2 OMA DM DiagMon

The OMA DM WG has introduced device management diagnostics and device monitoring functionality to remotely solve the problems of mobile devices. The overall goal of OMA DM DiagMon [11] is to enable management authorities to proactively detect and repair problems even before the users are impacted, or to determine actual or potential problems with a device when needed [12]. The management authority is an entity that has the rights to perform a specific DM function on a device or manipulate a given data element or parameter. For example, the management authority can be the network operator, handset manufacturer, enterprise, or device owners.

The OMA DM DiagMon includes the following management areas: diagnostics policies management, fault reporting, performance monitoring, device interrogation, remote diagnostics procedure invocation, and remote device repairing. The OMA DM WG publishes the standard documents as the following sequence: WID (Work Item Document), RD (Requirement Document), AD (Architecture Document), TS (Technical Specification), and EP (Enablers Package). The OMA DM DiagMon WG is currently working on TS. DiagMon only defines MOs for common cases of diagnostics and monitoring. We have expanded MOs for reset diagnostics based on MOs defined by DiagMon.

3 Management Architecture

Our goal is to provide an efficient method in order to debug software errors by collecting the reset data from the sold mobile devices. Fig. 2 shows the overall management architecture of our proposed solution, which is composed of the DM Server and DM Client. The DM Server sends the initialization and execution request of the reset diagnostic function to the DM Client. The DM Client, which is equipped in various mobile devices such as PDA, cell phone, lap top etc., replies the result of the request by the DM Server. The analysis server obtains the data related to reset and debugs the software errors by JTAG debugger. The presenter shows the data and the result of the debugging.

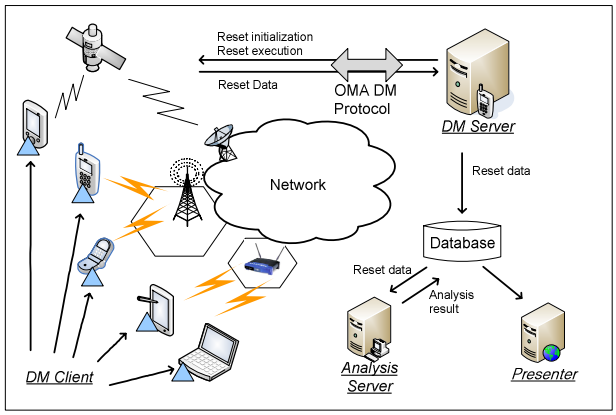

Fig. 2. Overall management architecture

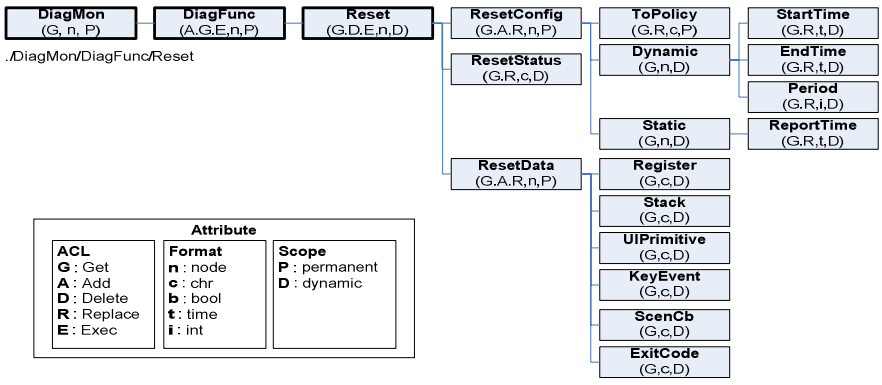

Fig. 3. DM tree for reset diagnostics

3.1 Management Information

We have defined the DM tree for the reset diagnostics (shown in Fig. 3) by expanding the *DiagMon* node and *DiagFunc* node defined by the OMA DM DiagMon WG.

These are not the standard nodes but under consideration of the standards. We have created the *Reset* node for the reset diagnostics. Each node has its own *access control list* (*ACL*), *format*, and *scope* attributes, which are denoted in the parenthesis in each node of Fig. 3. For example, the DM server can request *GET*, *DELETE*, and *EXEC* command on *Reset* node because its *ACL* is (G, D, E). The *format* defines the type of the node. The *scope* specifies whether the node is permanent or dynamic.

We have defined three children nodes under the *Reset*: *ResetConfig*, *ResetStatus*, and *ResetData*. The *ResetConfig* node is a placeholder for the reset configuration information. This interior node has following three children nodes:

- *ToPolicy*: the type of reporting schedule (value: *Dynamic*, *Static*)
- *Dynamic*: collect reset data from *StartTime* to *EndTime* and report it periodically. (e.g., if the *Period* is equal to 0, then report it immediately.)
- *Static*: report the reset data at the *ReportTime*.

The *ResetStatus* node specifies the operational state of the reset function. Its value is one of the followings:

- *None:* the collection of reset data is stopped
- *Prepared:* the Exec command of reset data collection is received
- *Active:* the collection of reset data is started
- *Processed:* the reset data is collected
- *Reported:* the collected reset data is sent

The *ResetData* node is a placeholder for the reset data. The child nodes contain the relevant information for analyzing the reset. We can reproduce the reset case on the JTAG debugger using this data. It includes the following children nodes.

- *Register*: register dump when the reset occurred.
- *Stack*: stack dump when the reset occurred
- *UIPrimitive*: last UI Primitives on MMI(Man Machine Interface)
- *KeyEvent*: event value of KEY_EVENT and data on MMI
- *ScenCb*: scenario and call back data dump
- *ExitCode*: pre-defined exit code value of the device

The usage of each node will be described in Section 3.2

3.2 Management Operations

We have designed the management operations based on the DM tree defined in Section 3.1. There are three separate phases in the management operation: initialization, execution, and gathering phase. At the initialization phase, the DM Server checks whether the mobile device can support the reset diagnostics or not. Also, it can create the reset MOs in the mobile device's DM tree if possible. At the execution phase, the DM Server sets the policy information related to the reset diagnostics and executes it. At the gathering phase, the DM Server gathers the reset data from the DM Client when it notifies the reset event.

By dividing the management operation into three phases as shown in Fig. 4, an efficient management operation can be achieved. First, each management phase consists of the same management commands. Hence, a single management command can process an operation of many MO addresses (Target LocURIs), which decreases the size of management package. Second, each phase can be independently used for its purposes. That is, to diagnose a reset, all three phases do not need to be repeated.

Once the initialization is processed, it does not need to be repeated. Also, after the execution phase, there is no need to repeat it to process gathering, as long as the policy for collecting the data remains unchanged. Therefore, it is more efficient than processing all three phases to diagnose a reset.

Fig. 4 (a) shows the initialization phase of the reset diagnostics. When the DM Server wants to initialize the reset diagnostics function, it needs to send the *NOTIFICATION* message [13] to the DM Client. When the DM Client receives the notification message, it sends the server-initiated *ALERT* command to the DM Server with the device name. Next, the DM Server sends the *ADD* command to initialize the reset diagnostics function and the *REPLACE* command to set *ACL* for *Reset as GET, DELETE,* and *EXEC*. For efficiency, as shown in the package *#4* of the sequence 5 and 9 in Fig. 4 (a), we have added many MOs by using one *ADD* command.

If the mobile device supports the reset diagnostics, it can add *Reset* to its DM tree and send a successful *STATUS* command (200) to the DM Server. If the addition of the *Reset* is successful, then the DM Server adds *ResetConfig*, *ResetStatus*, and *ResetData* step by step. Finally, the DM Server sends a completion message to the DM Client to finish this management session. After the initialization phase, the mobile device is ready to execute the reset diagnostics function.

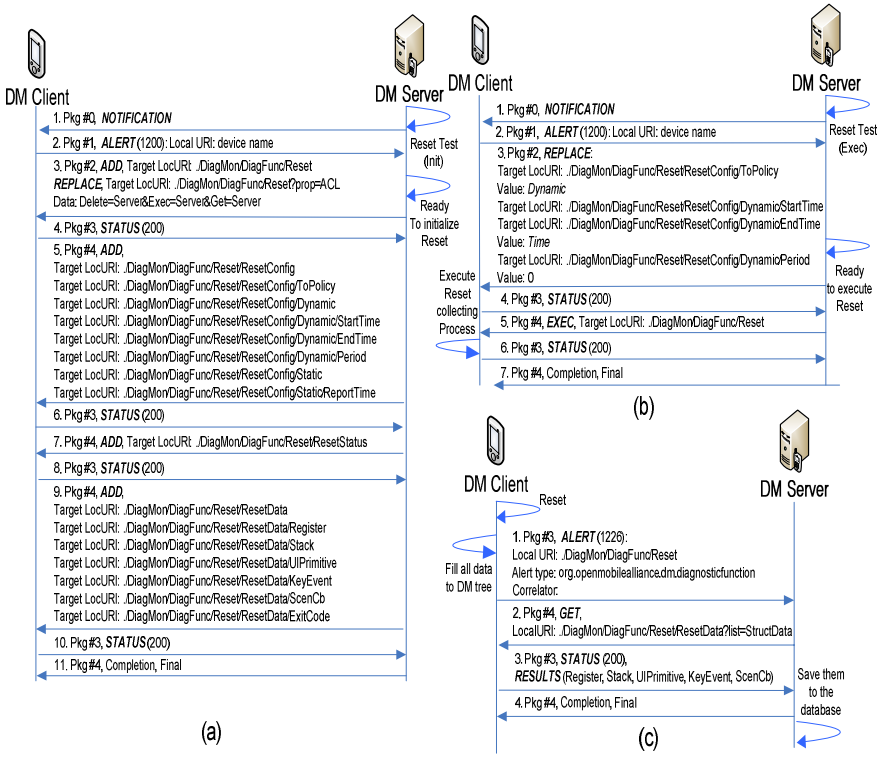

Fig. 4. Three phases of management operation: (a) Initialization phase (b) Execution phase (c) Gathering phase

Fig. 4 (b) shows the execution phase of the reset diagnostics. When the DM Server wants to execute the reset diagnostics function, it sends the *NOTIFICATION* message like the initialization phase. Then the DM Client sends the server-initiated *ALERT* command to the DM Server. The DM Server sends the *REPLACE* command to set the policy information. Since we need the real time data for analyzing the reset, the policy is dynamic in order to get the data from *StartTime* to *EndTime* periodically. The *Period* is 0 in order to receive the reset event immediately. If the mobile device has initialized the reset diagnostics function, then it sends the status code as 200 (success). Otherwise, it sends the status code as error code. Then, the DM Server sends the *EXEC* command to execute the reset diagnostics function. The DM Client executes the reset collecting process. Finally, the DM Server sends the completion message to the DM Client and the management session is finished.

Fig. 4 (c) shows the gathering phase of the reset diagnostics. When the reset occurs, the DM Client stores all relevant information to its DM tree. When it is ready to report, it sends the generic *ALERT* command to the DM Server. The DM Server sends the *GET* command for the *ResetData* node to retrieve all information related to the reset. It saves data to the database. As mentioned earlier, we can reproduce the reset case on the JTAG debugger using this data and find the error in the source code.

4 System Development

We now present the system development based on the DM tree and management operations presented in Section 3.

4.1 Design

Our proposed system is composed of the DM Client and the DM Server as illustrated in Fig. 5. The major components of the DM Client are *DM Tree Handler* and *Reset Detecting Process*. *DM tree Handler* manages the MOs for the reset diagnostics by commands which the manager requests. *Reset Detecting Process* detects the reset in the mobile device, collects the relevant information when the reset occurs, and fills it in the DM Tree. The major component of the DM Server is *Reset Tester*. *Reset Tester* runs initialization phase and the execution phase on the user's request. When the DM Client notifies the reset, it runs the gathering phase to retrieve the reset data and saves it to the data storage.

4.2 Implementation

Fig. 6 shows the screenshot of the Reset Diagnostic Client and the Reset Diagnostic Server. We have developed it based on the open source project called SyncML Conformance Test Suite [14]. Fig. 6 (a) shows the client system which initialized the reset diagnostic function. Fig. 6 (b) shows the server system which gathered the device information and the reset data from the mobile device 1.

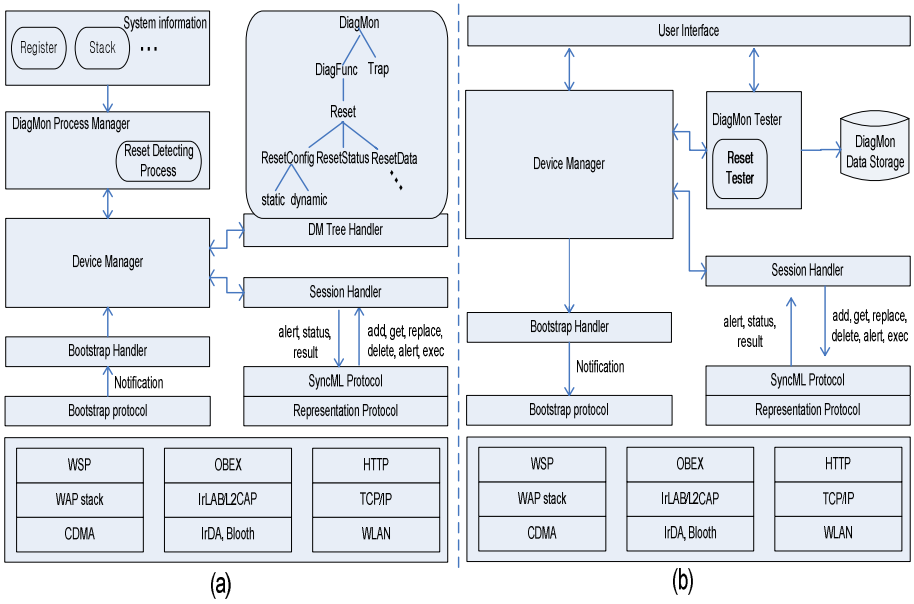

Fig. 5. System Architecture Design: (a) DM Client and (b) DM Server

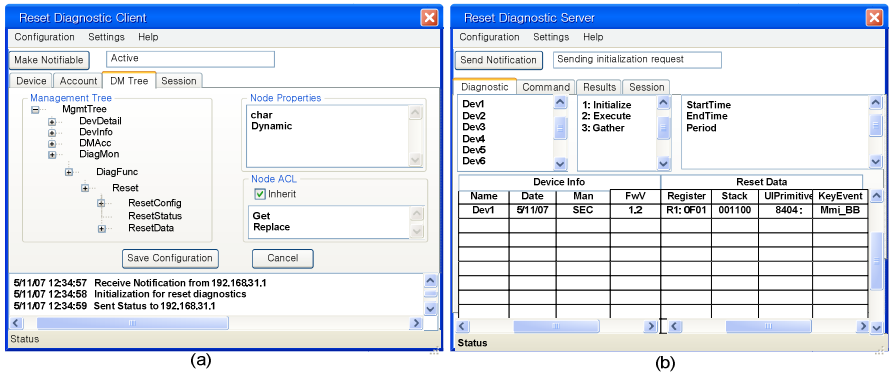

Fig. 6. (a) Reset Diagnostic Client system (b) Reset Diagnostic Server system

The server gathers the reset data from the client and the data is used to debug the reset error using JTAG debugger. Fig. 7 shows the screenshot of debugging using Trace32 debugger [18] as JTAG debugger. First, it loads the CMM file in accordance to the mobile device model and configures the debugging environment. The mobile device model information is recorded in the standard object *DevInfo*. This debugger can set the LR address in Register14 and we can find the source location of Register 14 address in the Data.list window. The name of source file and function can be found in the symbol.info window.

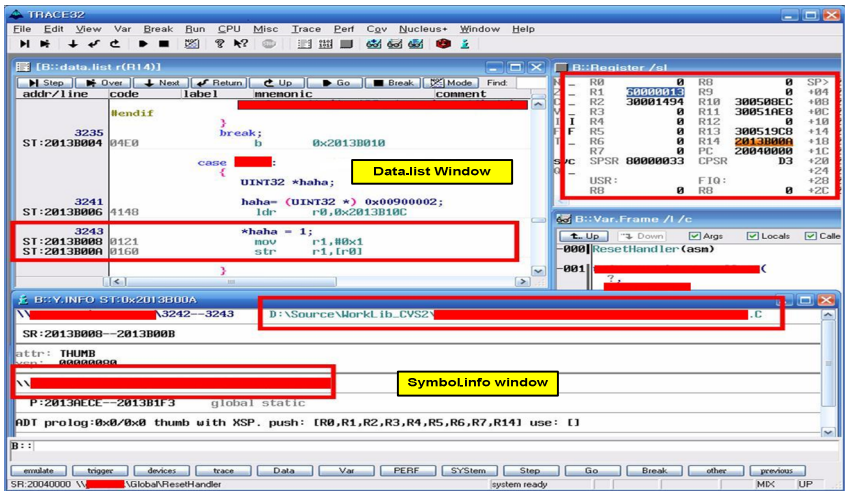

Fig. 7. Screenshot of debugging software error using JTAG debugger

5 Concluding Remarks

The software errors of the sold mobile devices cause inconvenience for the end-users. So far, we have used temporary solutions such as the system reset because it is not easy to completely remove them. In this paper, we proposed a debugging method to find and correct the software errors of the sold mobile devices based on OMA DM. We defined the MOs for the reset diagnostics and designed the management operation as three phases. We also developed the client and the server system for validating our proposed solution. Finally, we presented how to debug software errors with the reset data and JTAG debugger.

For future work, we need to evaluate the performance of the management operations and develop a system for integrating the firmware update system with our proposed system to solve the software errors and apply the solution to the mobile devices.

References

1. Chakravorty, R., Ottevanger, H.: Architecture and Implementation of a Remote Management Framework for Dynamically Reconfigurable Devices. In: ICON 2002, Singapore, pp. 375–381 (August 2002)
2. OMA (Open Mobile Alliance), http://www.openmobilealliance.org/
3. OMA DM (Device Management) Working Group, http://www.openmobilealliance.org/tech/wg_committees/dm.html
4. OMA, OMA Device Management Standardized Objects (2007)
5. OMA, OMA Device Management Protocol (2007)
6. SyncML Forum, SyncML Device Management Protocol, http://www.syncml.org/
7. OMA, OMA DM Device Description Framework, Version 1.2 (2007)
8. OMA, OMA Device Management Bootstrap (2007)

9. OMA, OMA Device Management Tree and Description (2007)
10. IETF, Uniform Resource Identifiers (URI) RFC 2396 (1998)
11. OMA, DiagMon (Diagnostics and Monitoring) Working Group.
12. OMA, DiagMon Requirement draft version 1.0 (June 2006)
13. OMA, OMA Device Management Notification Initiated Session (June 2006)
14. SyncML Forum, SyncML Conformation Test Suite, http://sourceforge.net/projects/oma-scts/
15. Adwankar, S., Mohan, S., Vasudevan, V.: Universal Manager: Seamless Management of Enterprise Mobile and Non-Mobile Devices. In: MDM 2004, Berkeley, CA, USA, pp. 320–331 (January 2004)
16. Stallings, W.: SNMP, SNMPv2, SNMPv3 and RMON 1 and 2, 3rd edn. Addison-Wesley, Reading, MA, USA (1999)
17. A Simple Network Management Protocol (SNMP), RFC 1157 http://www.ietf.org/rfc/rfc1157.txt
18. MDS Technology, Trace32 Debugger, http://www.mdstec.com/