International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 05 Issue: 04 | Apr-2018 www.irjet.net p-ISSN: 2395-0072

Debugging – A way to automate it

Abhilash Yadahalli[1], Dr. C. Vidya Raj[2]

[1]*Student, The National Institute of Engineering, Mysore, Karnataka, India.*
[2] *Professor, Dept. of Computer Science Engineering, The National Institute of Engineering, Mysore, Karnataka.*

Abstract – *Debugging is the routine process of locating and removing computer program bugs, errors or abnormalities, which is methodically handled by software programmers via debugging tools. Debugging checks, detects and corrects errors or bugs to allow proper program operation according to set specifications.*

In the debugging process, complete software programs are regularly compiled and executed to identify and rectify issues. Large software programs, which contain millions of source code lines, are divided into small components. For efficiency, each component is debugged separately at first, followed by the program as a whole. Debugging can be done manually by going through the code and finding and fixing the bugs or can be done using event logs.

The manual way can be tiresome process and involves lot of efforts and is time consuming. Thus, logs are used to obtain data about the system execution which helps in narrowing our view about the root cause of the issue. Thus, these logs help us in reducing both time and human effort. Thus debugging process can be automated to some extent to reduce both time and efforts[1]. A software or a tool can be developed which runs and collect all the system information required for a debugger to find and fix the problem thus, reducing his time and efforts.

Key Words: Debugging, Automated debugging, Debug logs, ETL Trace, Debug Analyzer, ETL Logs.

1. INTRODUCTION

Debugging ranges in complexity from fixing simple errors to performing lengthy and tiresome tasks of data collection, analysis, and scheduling updates. According to Daniel Hansson Verifyter[2], debugging skill of the programmer can be a major factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming language used and the available tools, such as debuggers. Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, restart it, set breakpoints, and change values in memory. Some of these can be automated to reduce both time and effort of the debugger.

In this paper, a concept where an ETL Trace is taken and analyzed for possible failure cases. A very commonly known failure is HPD-Hot Plug Detect. Hot Plugging is connecting the display to the system during or when the system is turned on. Many kinds of failures can in this condition like, Blank-Out: A display will not come up even after it's connected, Corruption: It happens when an image fails to display properly due to wrong programming of the display registers.

These kind of failures can be analyzed and the reasons for these kinds of failures can be approximated to an extent. The Analyzer tool does this by analyzing the trace and finding a possible reason for it.

1.1 Techniques of debugging

There are 2 types of debugging that can be done;

- Interactive Debugging: It includes methods like Print, Remote and Post-mortem Debugging.
- Debugging of Embedded Systems.

1.2 Log Files

Manual debugging involves setting breakpoints and finding the root cause for the issue. This is a tiresome process and involves lot of efforts and is time consuming. Thus, logs are used to obtain data about the system execution which helps in narrowing our view about the root cause of the issue. Thus, these logs help us in reducing both time and human effort. ETL Files, Mem-Dumps, etc. are some examples for log files.

2. ETL Files

Microsoft Windows records application and system-level warnings, errors or other events to a binary **file** called the event trace log, or **ETL**, which can then be used to troubleshoot potential problems. An ETL file is a log file created by Microsoft Trace log[4], a program that creates logs using the events from the kernel in Microsoft operating systems. It contains trace messages that have been generated during trace sessions, such as disk accesses or page faults. ETL files are used to log high-frequency events while tracking the performance of an operating system.

Trace logs are generated by trace providers in trace session buffers and are stored by the operating system. They are then written to a log and stored in a compressed binary format in order to reduce the amount of space

occupied. Reports may be generated from ETL files using the command-line utility Tracerpt. ETL file output may be configured with several options, such as a maximum allowable file size, so that the logs do not cause a computer run out of disk space.

These ETL files can be opened using windows built-in analyzer called **Windows Performance Analyzer**. $Windows Performance Analyzer (WPA)[4] is tool that creates graphs and data tables of Event Tracing for Windows (ETW) events that are recorded by Windows Performance Recorder (WPR), Xperf, or an assessment that is run in the Assessment Platform. WPA can open any event trace log (ETL) file for analysis. WPA is a powerful analysis tool that combines a very flexible UI with extensive graphing capabilities and data tables that can be pivoted and that have full text search capabilities.

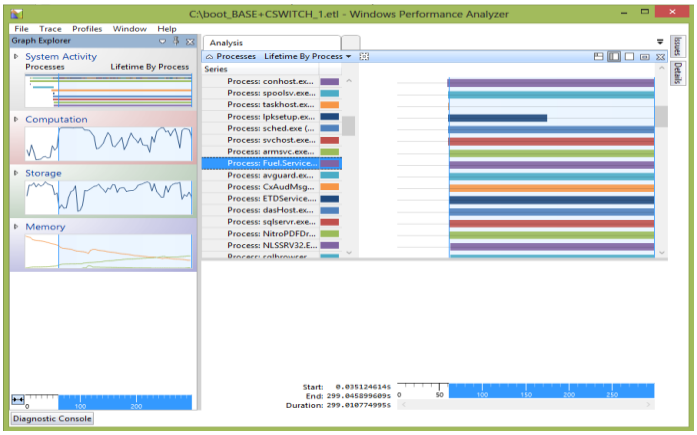

Fig-1: ETL Trace File

A tool called Tracescripts is used to obtain these ETL files[8]. This tool is run and the issue is reproduced, the tool records the system behavior from start to the end of the session which creates .ETL file. This file is viewed using WPA which tells the behavior of the system.

2.1 Trace Sessions

A *trace session* is a period during which a trace provider is generating trace messages. The system maintains a set of buffers for the trace session to store trace messages until they are delivered ("flushed") to a trace log or a trace consumer.

There are three basic types of trace sessions: trace log sessions, real-time trace sessions, and buffered trace sessions. A single trace session can be a trace log session, a real-time trace session, or both. Buffered trace sessions are exclusive.

In a *trace log session*[9], trace messages are written from the trace buffers to a log file in binary format. This is the standard, default type of trace session.

3. Design and Implementation

The main objective was to automate the debug process and to reduce the time and human efforts required. The logs obtained after the trace session are checked for correctness, if found correct the logs are then given as an input to the Analyzer.

The Analyzer takes these logs, processes them and outputs the result. The processing of the log is an iterative process which includes various steps at different stages.

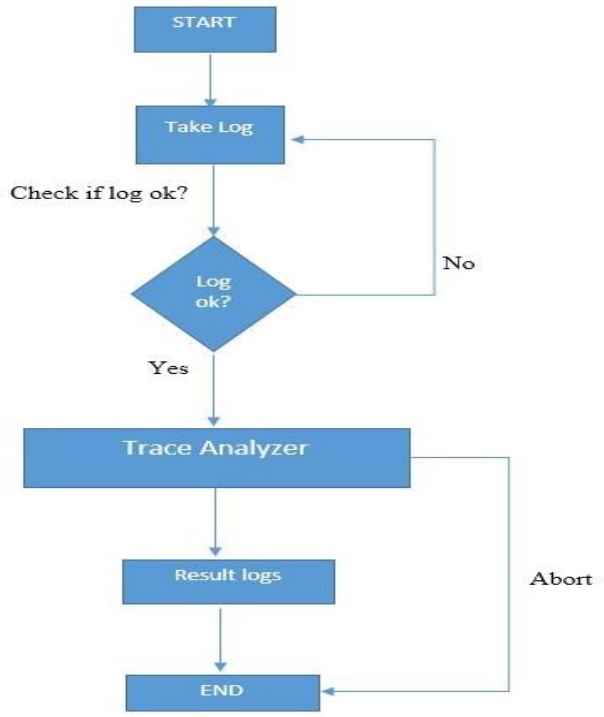

Fig-3.1: Flow Diagram of the Analyzer

The above flow chart slows the flow diagram of the Analyzer. A user can at any point abort the analyzing which will end the processing of the log files.

In this paper the main concentration is on HPD related failures, analyzing them and finding a possible reason for these failures or errors.

Hot-Plugging a display might result in many kinds of failures like, display might fail to show up (Blank-out), corruption on the display, resolution might not show in its timing and many more. All these failures are analyzed and a possible reason for these failures are displayed by the tool. This helps the debuggers to resolve and find a fix to the bug more efficiently and quickly.

Fig-3.2: Corruption seen on Hot-Plug.

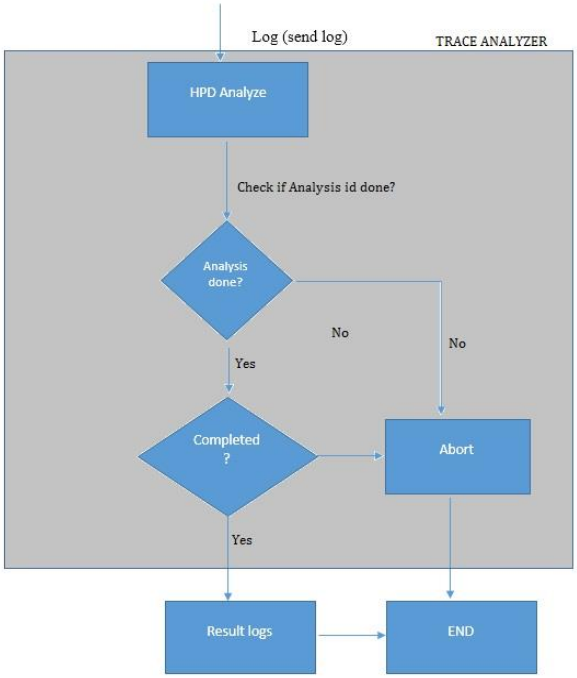

Fig-3.3: Flow Diagram of the Trace Analyzer Tool for HPD.

A detailed expansion the Trace Analyzer tool is shown in fig 3.3. Once the HPD Analysis is exited, the tool checks for the correctness and completeness of the output. If everything is fine, the results are stored in a file which is given as an output to the viewer. If not then the analysis is aborted and ABORT message is displayed.

Once aborted the user has to re-run the whole analysis from the beginning.

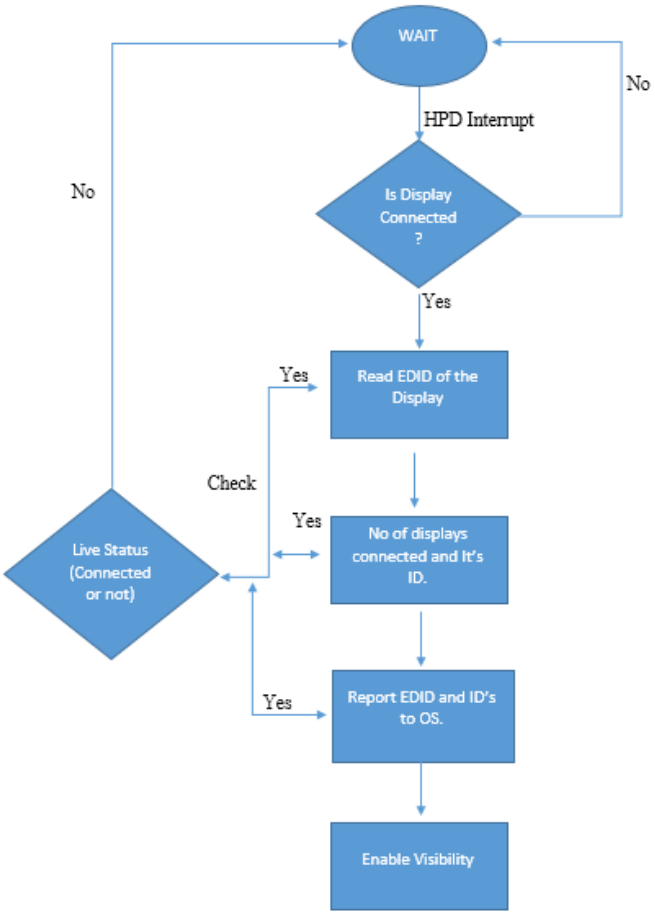

Fig-3.4: Flow Diagram of the Analysis done on Trace files.

The step-by-step analysis done on Trace files is shown in fig 3.4.

Until any display is Hot-Plugged the system remains in wait/unconnected state. Once Hot-Plugged/connected the system runs into a series of checks and captures logs related to connected display and OS. At all point the system checks for the LiveStatus of the display. LiveStatus is whether the display is connected or not. If disconnected at any point during the analysis, the trace stops and displays an error message and return back to wait state.

The main data that is read from the connected display is its Extended Display Identification Data (EDID) [10]. EDID is a metadata format for display devices to describe their capabilities. EDID is a 128-bit data structure contains information like manufactures name, serial number, product type, timing supported, display size, luminance and DisplayID. Every Display Driver uses the EDID structures.

Address (Decimal)	Data	General Description
0-7	Header	Constant fixed pattern
8-9	Manufacturer ID	Display product identification
10-11	Product ID Code	
12-15	Serial Number	
16-17	Manufacture Date	
18	EDID Version #	EDID version information
19	EDID Revision #	
20	Video Input Type	Basic display parameters. Video input type (analog or digital), display size, power management, sync, color space, and timing capabilities and preferences are reported here.
21	Horizontal Size (cm)	
22	Vertical Size (cm)	
23	Display Gamma	
24	Supported Features	
25-34	Color Characteristics	Color space definition
35-36	Established Supported Timings	Timing information for all resolutions supported by the display are reported here
37	Manufacturer's Reserved Timing	
38-53	EDID Standard Timings Supported	
54-71	Detailed Timing Descriptor Block 1	
72-89	Detailed Timing Descriptor Block 2	
90-107	Detailed Timing Descriptor Block 3	
108-125	Detailed Timing Descriptor Block 4	
126	Extension Flag	Number of (optional) 128-byte extension blocks to follow
127	Checksum	

Fig-3.4: EDID 128-bit table.

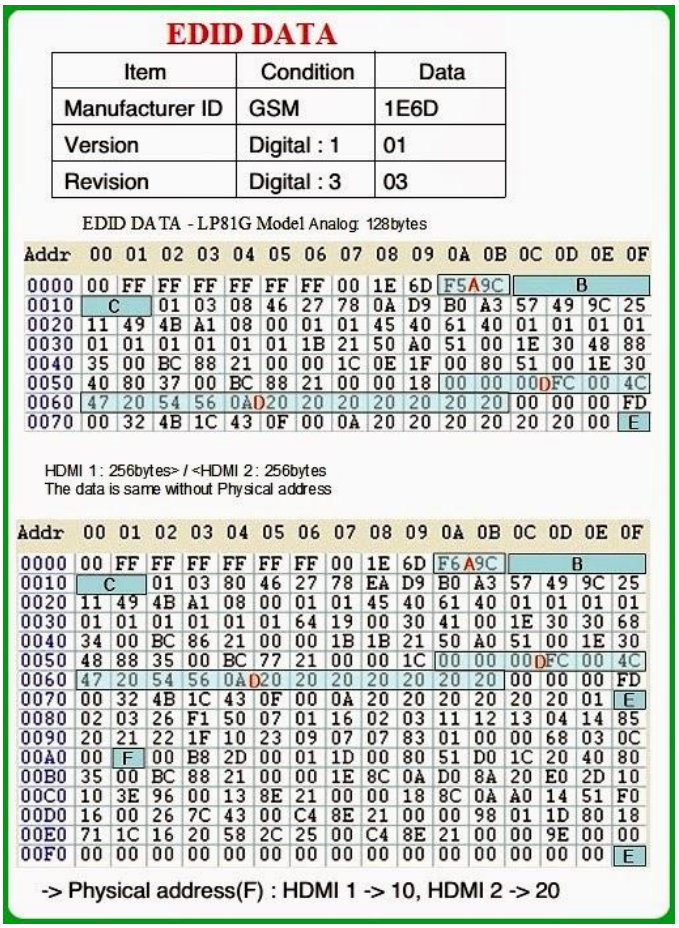

Fig-3.5: EDID 128-bit Raw Data.

4. CONCLUSIONS

In the debugging process, complete software programs are regularly compiled and executed to identify and rectify issues. Debugging when done manually can be very tiresome and time consuming as in involves going through the code, finding and fixing the bugs. This process to some extent can be automated using event logs in order to reduce both time and human efforts. In this paper there is a suggested working idea on how to automate a part of debug process.

HPD is just one of the checklist. Similarly analyses can also be added for different checklists like, Blank-out, Buffer Under-run, memory allocation and de-allocation etc.

The trace analyzer does this work by analyzing the logs of different checklist and finding the most nearest and accurate solution to fix the bug. This tool can only automate the process up to some extent and not all of it. The debug process still needs a human hand in looking through the output data from the analyzers and trying the best solution to fix the bug.

REFERENCES:

[1] Measuring the Gain of Automatic Debug http://ieeexplore.ieee.org/document/6926095/?reload=true

[2] Automatic Bug Fixing by Daniel Hansson Verifyter AB, Lund, Sweden http://ieeexplore.ieee.org/document/7548934/

[3] https://docs.microsoft.com/en-us/windows-hardware/test/wpt/

[4] https://en.wikipedia.org/wiki/Tracing_(software)

[5] https://msdn.microsoft.com/en-us/library/windows/hardware/hh448170.aspx

[6] https://fileinfo.com/extension/etl

[7] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/trace-message-control-file

[8] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/etl-trace-log

[9] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/trace-session

[10] Wikipedia-Extended Display Identification Data