AND

A New Paradigm in Linux Debug

September 2008

By

Art Lee, Viosoft Corporation

And

Bruce Ableidinger, MIPS Technologies, Inc.

The Current Paradigm

No one today would argue against the fact that Linux has taken the embedded Real Time Operating System (RTOS) space by storm. More and more applications that historically required either a commercially available RTOS or one that was internally created and maintained are being replaced with a Linux-based platform. The reasons for this movement vary from one company to the next, but some of the most common factors are:

1. The availability of source code to the operating system
2. A wealth of device drivers and communication stacks
3. An increasingly available pool of software engineers that are Linux proficient
4. A perceived cost advantage achieved with removal of the OS royalty component from the products' bill of materials
5. Semiconductor suppliers now provide a Linux port to their SoC based on their hardware reference platform, along with tool chains and a reference distribution

To take full advantage of the Linux operating system, original equipment manufacturers (OEMs) have a choice of engaging with a commercial Linux vendor or adding additional engineering capability in-house. Both models have proven successful, but each carries its own specific costs.

Regardless of the direction the OEM chooses, the typical debug model available for their engineers is the same… a command line-based, client server environment based on GDB (GNU Debugger). This model is illustrated in Figure 1, which depicts an instantiation of GDBSERVER attached and running on each Linux process under debug on the target. Each GDBSERVER is communicating with the host through an Ethernet port.

In addition, it is important to understand that in this approach to Linux debug, the standard Linux kernel is replaced with a "static" version specifically built with debugging code instrumented throughout. With only a few exceptions, all debug communication to the target via KGDB is limited to an RS232 serial link.

This approach provides an additional challenge to the developer that is using the instrumented version of the Linux kernel by actually altering the performance of the target under debug from the "released version" that will eventually ship with the product.

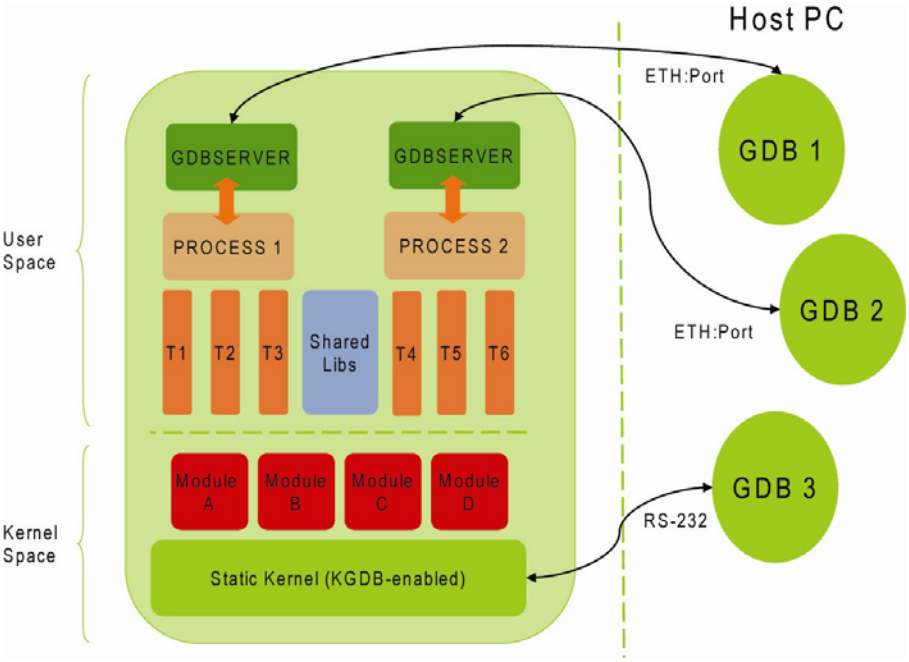


Figure 1: Standard Linux Debug Model

While this is by default the accepted Linux debug environment, there are some well understood limitations to this approach. For example, applications that consist of multiple processes will require multiple copies of GDBSERVER running in the often limited target memory. This can affect the performance of the target under debug. There have been cases of a 50%+ degradation of target performance.

Even in the best case scenario where all kernel instrumentations and communication channels are available, there are still areas of the code that are inherently inaccessible under this debugging paradigm. The illustrated “problem” areas in Figure 2 have presented multiple challenges to kernel and application developers. These areas include the large amount of threads under each process and kernel loadable modules that are code- and data position-independent. While it is possible for skilled developers to put together an environment based on existing technologies to address the debugging needs in these areas, such an environment has been shown to be very user-unfriendly and non-scaleable under load.

Consider the case of Linux kernel loadable modules, which consist of an initialization routine to be invoked at module loading time. Current debug paradigms suggest that such modules be loaded, and their code and data offsets then be adjusted (manually and automatically) within the debugger. However, by this time, the initialization code of the module has already been executed and there is no possible way to debug a problem in this area of the code. Another use scenario involves shared libraries, which are often not well handled by GDBSERVER or equivalents.

Given these obstacles, many engineers still resort to **printf** (user space) and **printk** (kernel space) as their primary debugging aids. Not withstanding the “ugliness” issues and time overhead of recompiling and linking in these messages, it is not uncommon for such debug “instrumentations” to skew the behavior of the target system code to the point that it introduces new software problems or possibly masks existing problems.

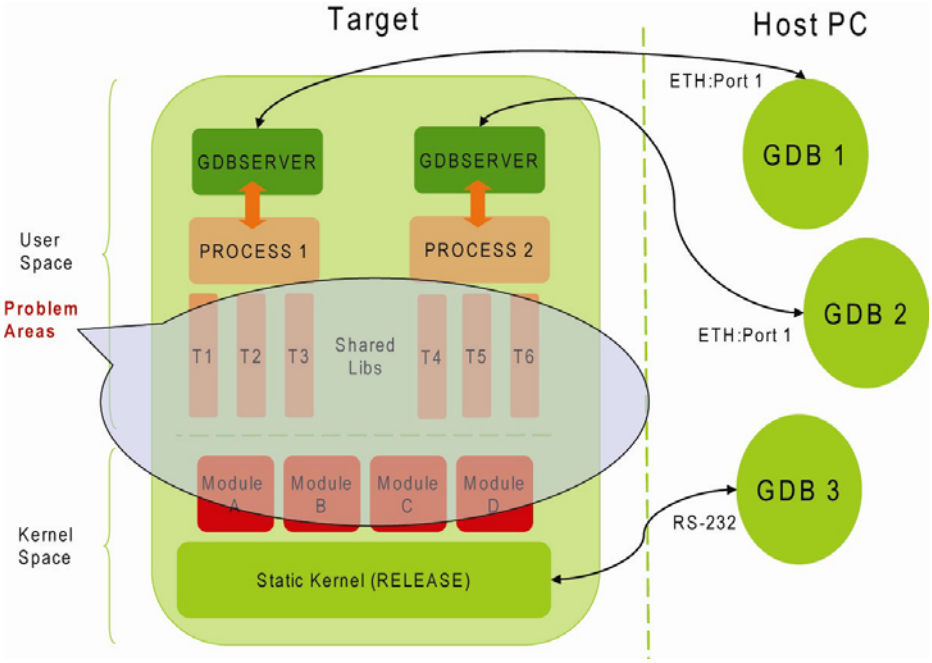


Figure 2: The "Problem" Areas

The Arriba Debugger: A Holistic Approach to Debugging Linux

The Arriba Debugger is designed from the ground up to provide a holistic approach to debugging embedded Linux. In place of GDBSERVER and KGDB, VMON2 is a dynamically loadable, demand-based debug agent that runs on the embedded Linux target. Communicating with the Arriba Debugger on the host, VMON2 provides total visibility of the Linux target, from user-level threads to the static kernel.

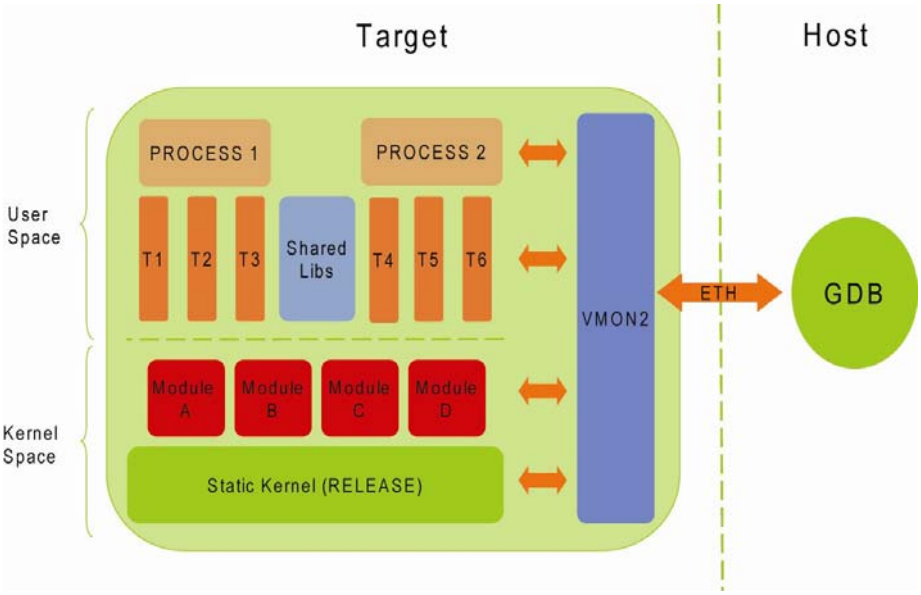


Figure 3: The Arriba Solution

VMON2 has a very small memory footprint and even when loaded has an almost immeasurable performance impact on the running system. At less than 250KB in size on the target, VMON2 is able to provide end-to-end debugging of the target over a single Ethernet connection.

Addressing Well Known Embedded Linux Debug Challenges

Problem 1 - Loadable Modules

Through the Arriba Debugger, VMON2 can be configured to signal the host when a kernel module of a given property is loaded on the target. Upon reception of this signal, the Arriba Debugger will automatically and correctly load the symbol information of the respective module, and place control at the entry point to the module initialization function. The user can now have full debug control of the module in question over a high-speed Ethernet link.

Traditional debug of the Linux kernel or module (when possible) is accomplished with KGDB or JTAG, which completely halts the target under debug. In contrast, an important feature of VMON2 is its ability to provide the same level of debug non-preemptively. In other words, the Linux kernel on the target continues to handle inbound and outbound network traffic, multimedia data, and other time-critical activities that are crucial in maintaining the appearance of normal execution to the outside world. This ability is critical to many data and media-centric applications such as set-top boxes, digital media appliances, and high-speed networking switches and routers.

Problem 2: Debugging of Multiple Processes; Parent/Child Processes

In many instances, Linux application programmers need to create applications that involve multiple processes. Such processes are spawned from a single parent process earlier in the application initialization sequence. A frequent challenge revolves around the need to set breakpoint(s) in the child process and eventually hit such breakpoints when the child process is created and running. Straightforward as this may sound, it is an unsupported use-case with existing Linux debuggers; embedded or otherwise. As a workaround, developers often find themselves manually inserting instrumented code in the child process with an infinite loop that is gated by a variable initially set to ‘true’. This enables debugging tools such as GDBSERVER to attach to the child process in question, change the value of the gating variable to ‘false’ to unblock to loop, and resume debugging.

Because VMON2 has ultimate visibility into the Linux target, events such as process creations result in a notifying signal being sent to the Arriba Debugger on the host. The Arriba Debugger, upon determining that a breakpoint is pending for the child process, transmits the proper run-control sequence to ensure that such a breakpoint is set in the child process code space.

Problem 3: Debugging Kernel Drivers and Shared Libraries… Production Released Kernel

Depending on the scope and breadth of the application, the list of Linux debug “problem areas” can range anywhere from the inability of the programmer to use the debug tools on his or her deployment platforms due to footprint and system performance constraints imposed by debugging techniques, to the tedious and error-prone workarounds that result in much wasted

time and increased frustration. The Arriba Debugger provides an in-depth solution to these problems and beyond.

As a final example, consider the need for programmers and field application engineers to diagnose and fix bugs that occur in products that have already been deployed to the field. Under such conditions, the target platform is subject to severely limited debugging and communication access. VMON2, as a loadable module, can be configured to be launched on already-deployed systems. Thus, VMON2, with its ability to effectively debug and diagnose such systems with minimal intrusion has time and time again proven to be an indispensable tool through all stages of the product lifecycle.

MIPS Technologies' Navigator™ Integrated Component Suite (ICS)

MIPS Technologies recently announced the availability of the MIPS Navigator™ Integrated Component Suite (ICS). This powerful Eclipse-based Integrated Development Environment (IDE) is the cockpit for existing and future tools for developing a MIPS-Based™ design. The Arriba Linux Debugger is now available directly from MIPS Technologies as a plug-in to the MIPS Navigator ICS. This seamless integration is the result of more than four years of collaboration between MIPS Technologies and Viosoft Corporation.

Within the MIPS Navigator ICS is a full-featured Eclipse CDT environment that has been customized specifically for the MIPS® architecture. In addition, MIPS Navigator ICS includes the latest CodeSourcery™ SG++ GNU based toolchains for MIPS and all of the expected features necessary to develop code. The MIPS Navigator ICS also integrates support for all MIPS Technologies' processor IP, including PDTrace™ and EJTAG probe technologies.

In addition to the Arriba Linux Debugger, developers can leverage another new profiling tool called the Arriba Linux Event Analyzer (LEA)—also a plug-in to the Navigator ICS. This new tool provides the ability to see all Linux events occurring on the target by capturing the information and displaying it in a time domain format. The Arriba LEA collects and provides a significant amount of information about the Linux system, including context switches among processes and threads, signals and elapsed execution time.

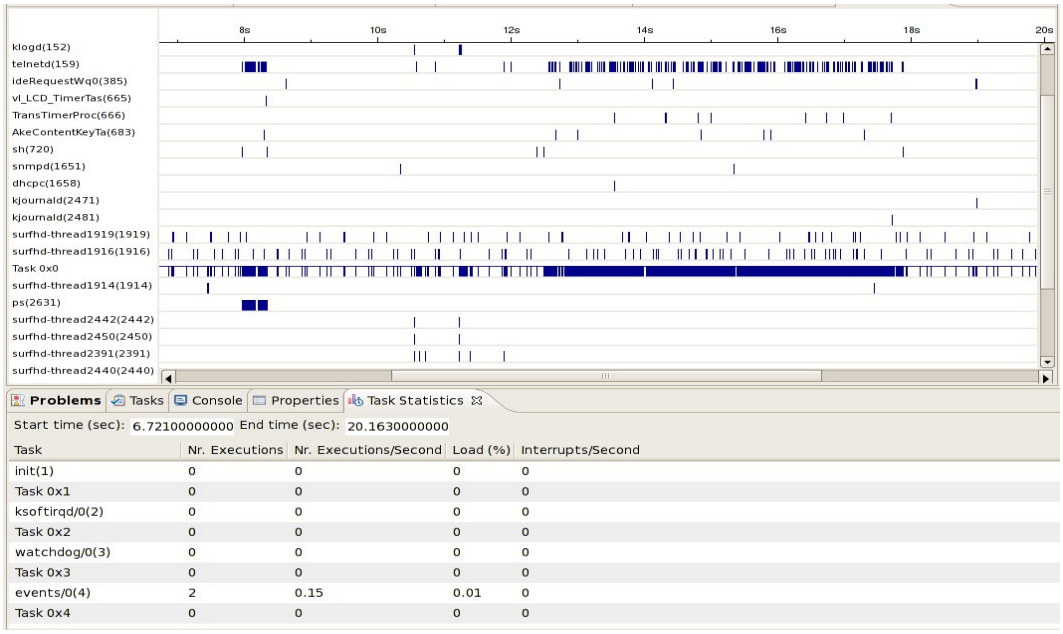
The LEA has a small memory footprint and a minimal impact on CPU cycles. Because the LEA is light-weight and able to dynamically add and remove instrumentation points on a production-ready system running Linux, it is an ideal performance analysis and debugging tool for both in-house development and field service.

An example LEA screen display is shown below in Figure 4. Within this view, the user can zoom in and out to gain a detailed understanding of how their code behaves and how the tasks execute in both the kernel and user space areas. The LEA provides the ability to measure latencies, response times to external events and even the load that each event represents on the running system. This information is also available in a "raw" format that can easily be imported to Microsoft Excel for additional post-processing and analysis.

Because no two end-user applications are alike, each developer or team of developers within an organization is likely to be interested in collecting and visualizing different aspects of the system with the LEA. The need for an open-ended analysis tool led to a highly-customizable design. By creating and deploying their own kernel module plug-in to the LEA, developers can easily and rapidly gain a level of visibility into their applications and system that is not possible with other close-ended tools.

The LEA uses the same instrumentation technology employed by VMON2 in the Arriba Linux Debugger, which means that no debug patches or special compilation of the Linux kernel is required. This capability makes the LEA an ideal choice for deployment on production systems.

The combination of the Arriba Linux Debugger, Arriba LEA and MIPS Navigator ICS provides MIPS developers with a comprehensive and powerful Linux development environment. The solution was designed to shorten customers' time to market while providing developers the ability to ensure a level of code quality that until now was not obtainable.



Task	Nr. Executions	Nr. Executions/Second	Load (%)	Interrupts/Second
init(1)	0	0	0	0
Task 0x1	0	0	0	0
ksoftirqd/0(2)	0	0	0	0
Task 0x2	0	0	0	0
watchdog/0(3)	0	0	0	0
Task 0x3	0	0	0	0
events/0(4)	2	0.15	0.01	0
Task 0x4	0	0	0	0

Figure 4: The Linux Event Analyzer (LEA) ICS View

Seeing is Believing!

As with any new technology solution to well known debug challenges, it is reasonable for prudent readers to cast doubt as to whether it will work in their embedded Linux environment. MIPS Technologies welcomes you to contact us for an in-depth product demonstration. We will not bore you with "Hello World" debug examples, but instead real world applications that involve very large amounts of code. Contact MIPS Technologies at sales@mips.com and see for yourself.