Debugging in virtual worlds: *"Checking out" the hypervisor"*

In order to save money, the functions from several electronic devices are consolidated on a common hardware unit. A hypervisor separates the functions on the software side. This results in debugging becoming more challenging but by no means impossible.

Hypervisor – embedded software developers are currently faced with this term all the time. There is almost a hype around this technology (pun intended). For instance, it seems to be a focal point of discussion at the moment in the automotive, aviation and aerospace segments as well as in the field of medical technology. However, what impact does this have on the development cycle and, in particular, in terms of debugging? Debugging tools, particularly those that access the hardware (e.g. JTAG debuggers) need to take so much into consideration when a hypervisor is utilised on the target system. Naturally, the developer wants to have a tool at their disposal that shows them the complete status of the embedded system including all components such as the hypervisor, guest operating systems and guest processes.

Several machines on a single piece of hardware

According to Wikipedia, "Hypervisors permit the simultaneous operation of several guest systems on a single host system". They are used to run various tasks on a single piece of hardware in parallel. In doing so, the tasks are so varied that different operating systems are used in order to implement them. The hypervisor is responsible for allowing these various operating systems to run on a single computer, either by dividing the CPU across the operating systems in a time slice technique or by dynamically assigning the individual cores to different guests in a multicore environment. Everybody is aware of such hypervisors on desktop computers such as VMWare or VirtualBox. For instance, they allow one (or several) complete Linux distribution(s) to run on Windows. Other examples that are also utilised in embedded systems include Xen, KVM, Jailhouse and QEMU.

A concrete application from the embedded systems segment may be structured as follows: The objective is for a car dashboard to work with an industrial Linux distribution, for the infotainment system to operate using Android, for the air conditioning to utilise FreeRTOS and the engine control to work with an AUTOSAR Stack. In the past, four (and more) different hardware platforms were actually required for this purpose. However, all of these functions are now integrated into a single system and, where possible, even on a single CPU.

Why? The first reason can be attributed to costs. Nowadays, embedded systems are so powerful that a single system is able to complete all of these tasks. Furthermore, it is also cheaper to produce and install an integrated hardware module rather than four different systems. This is the primary motivation as every penny counts, especially in the automotive industry. As an "add-on", a hypervisor provides an extra layer of security and protection. The hypervisor is able to monitor all guests and act accordingly in the event of issues, e.g. by restarting a guest. It is also essential to protect the guests from unwanted interaction. A technical prerequisite for this is to ensure that all guests are kept separate from each other in terms of hardware via an independent MMU (Figure 1). We will encounter this feature once again, especially when it comes to debugging.

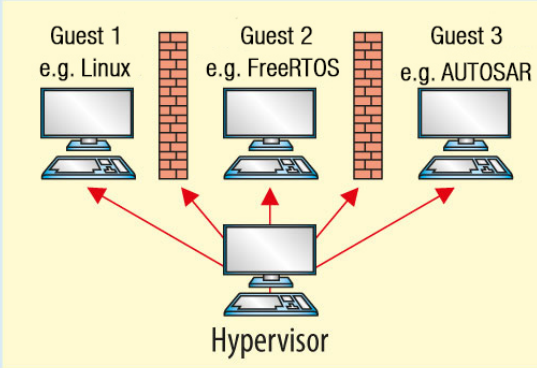

Figure 1: A hypervisor coordinates the operation of several virtual machines on a real machine. In doing so, it ensures a strict separation of the virtual machines.

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Hypervisor functionality

In terms of hardware, the individual guests can be separated from each other if the CPU provides a complete hardware abstraction. In order to do so, three things must be virtualised in principle: The memory, the peripheral equipment and the CPU itself (Figure 2). The guest's operating system should not even know that it is running in a virtualised machine. This means that the operating system manages its own MMU ("Stage 1 MMU") and its own "physical" memory ("guest physical = "intermediate"). However, this is not actually physical. In fact, it is then compiled in a real physical address space in a second MMU ("Stage 2 MMU") of the hypervisor. The peripheral equipment is also virtualised ("virtual I/O") in order to ensure that each guest is able to interact with the environment. In doing so, the hypervisor decides which guest may access which piece of peripheral equipment and which responds to guest interruptions. Finally, each guest receives one or several virtual CPUs that are mapped on the actual cores via a scheduler. In doing so, the number of virtual CPUs of a specific guest can be lower or greater than the number of real cores.
Using the example of the aforementioned automobile system, a chain of events would take place as below:

→ A temperature sensor detects a drop below 3°C and triggers a hardware interrupt. This interrupt is received and processed by the hypervisor. The hypervisor forwards the interrupt to the guest as a virtual interrupt for the dashboard.

→ The guest system receives the virtual interrupt (or guest driver) and sends a signal to the process within the guest that is responsible for the warning system and displays "Danger: Ice".

Communication between the guests represents a further example: The driver has recognised that it is cold outside and presses the "heating on" button on the dashboard. Thereupon, the guest responsible for the dashboard sends a signal via the hypervisor to the guest responsible for the air conditioning that it should turn the heating on.

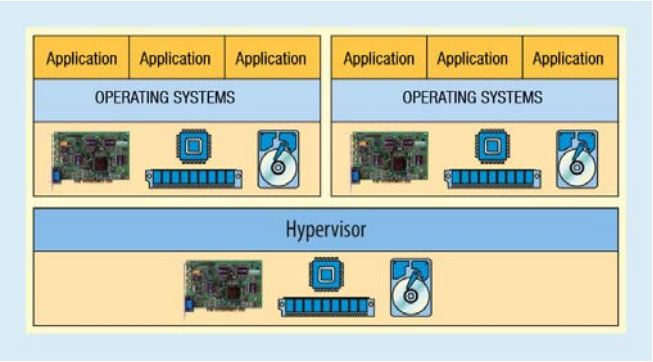

Figure 2: The hypervisor ensures a complete virtualisation of the existing software.

Hypervisor impact on debuggers

So far so good but what happens if the system doesn't respond as expected during the development phase? If the warning signal process is not triggered at all for instance or if the air conditioning has no idea of what the driver wants? The software must be examined using a debugger in order to find the fault.
In principle, there are two debugging methods: The software-controlled run mode debugging and the hardware-controlled stop mode debugging.
The run mode debugging method involves the loading of an additional debug software in the system (e.g. "gdbserver" for Linux processes) that accomplishes the actual debugging. Single-step mode, breakpoints, etc. are all managed by this piece of software (also called the "debug agent"). For this purpose, a debugger on the development computer communicates with the agent, e.g. via a serial interface or Ethernet. In order to ensure that this works, only the components to be debugged are stopped, e.g. a Linux process. The rest of the system will continue to run (this is the reason why it is called "run mode"). The system must continue to run in order to ensure that the communication can be further maintained with the debugger.
Such a debug session only requires an appropriate communication channel. If an underlying hypervisor is present, the channel is simply routed through it (Figure 3). Once this route has been established, neither the debugger nor the agent is aware that a hypervisor is present in-between them, i.e. the debugging is "hypervisor agnostic". This method is perfect if the system needs to continue during the debugging, e.g. because protocols need to be served. This is completely sufficient for the debugging of functions within a process or when it

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

comes to processes within a single machine. However, this method reaches its limits as soon as drivers (Linux modules) are involved. This is particularly the case when the user leaves the VM and other guests or the hypervisor are affected. Therefore, a different debugging method is required if an error is pending for the warning signal outside of the process in the aforementioned chain of events.

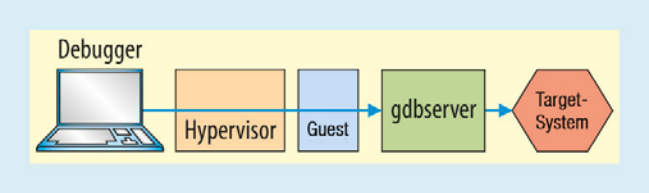

Figure 3: "Run mode" debugging with a gdbserver.

Stop mode debugging: Stop everything

When it comes to hardware-controlled debugging, the debugger is connected directly to the CPU via special pins (Figure 4). The debugger uses these pins, typically JTAG, to control the CPU itself, e.g. stop it, trigger individual program steps, read the registry or memory. However, this also means that the entire system, including all processes, guests and – of course – the hypervisor, is stopped in the event of a breakpoint. In such a case, no more interrupts are operated, no communication protocols run and no VM, process or task changes take place. The CPU is effectively "frozen", which is why it is called "stop mode".
When in this state, the CPU only "sees" the components that are currently released by the MMU, i.e. only one guest (the one currently running on the CPU) and only one process (the one where the guest is currently active). All registries and memory accesses relate to this context. The CPU does not have any access to other VMs or processes. A hardware debugger also accesses the system via the CPU and is therefore initially subjected to the same restrictions: It can only "see" the current situation. However, it is able to do slightly more than that: Thanks to a temporary minimal manipulation to the MMU registry, it can also directly read the physical address space and the current "intermediate" (= "physical guest") address space. However, all debug symbols belonging to the processes and guests are stored on virtual addresses, meaning that this additional view is not particularly useful to begin with. However, developers want to see everything: The hypervisor, all guests as well as all guest processes all at once and all at the same time! However, this is, in principle, not possible in run mode for the aforementioned reasons. But it is possible in stop mode. , which is the main strength of this option.
For the debugger to be able to see everything beyond the current status, it needs to know about the system, i.e. demonstrate "awareness". It requires a "hypervisor awareness", an "OS awareness" for each guest and an "MMU awareness" for both the hypervisor as well as for each guest, which can vary considerably. As the debugger is now aware of the system layout, it can read the list of guests and processes as well as their MMU tables from the system. Equipped with this knowledge, the debugger performs the MMU table walk (translation of the virtual into physical addresses) for each virtual address of a guest or process itself, i.e. past the hardware MMU, and reads the respective data from the physical memory directly. By implementing this method, the debugger accesses all addresses belonging to all guests and all processes, irrespective of whether they are virtual, intermediate or physical. And all of this is done at the same time!

Figure 4: JTAG debugger with the target system: This is how a hardware debugger is connected to the target system.

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Debugger needs to have "awareness"

Consequently, the debugger accesses a target system that consists of a hypervisor and several guest operating systems. Each machine has its own set of registries, MMU translations, processes, symbols, breakpoints, etc. The debugger must be able to work with each of these machines. This applies to both the "real" machine (i.e. the hypervisor) as well as all virtualised guest systems as if they were real machines. Of course, they need to work with all machines at the same time.
The aforementioned "awareness" is used for this purpose. A hypervisor awareness must be dedicatedly adapted to each hypervisor and determine the list of the virtual machines, their IDs, virtual CPUs and the MMU settings for this. The awareness uses the hypervisor debug symbol information (ELF/DWARF) in order to read the necessary information from the system. The illustration of the virtual machines and their resources provides an excellent system overview (Figure 5). The hypervisor awareness is also responsible for managing the layout of the stage 2 MMU translation so that the debugger has access to all VMs.

An "OS awareness" is required in order to analyse the content of a guest operating system and each guest requires its own. The awareness is also developed specifically for each OS in use here. This awareness then determines the processes of the operating system and the MMU settings within the VM as well as the MMU table layout (stage 1 translation). For this purpose, the awareness then uses the debug symbol information belonging to the respective operating system. When using Linux, this is the "vmlinux" file for instance. As a result, processes, threads and other resources can be illustrated.

Using these awarenesses, the debugger is ultimately aware of the machines, operating systems and processes running on the target system. Consequently, the TRACE32 debugger developed by Lauterbach is able to illustrate a hierarchical tree of the entire system. Various commands and windows can be specially used on a certain machine or certain process. For instance, the process taking place on a Linux machine and the task being performed by a FreeRTOS device can be shown at the same time (Figure 6). The TRACE32 symbol management has been changed accordingly so that the developer is able to assign the loaded symbols to a certain machine or a certain process. It was also expanded with a machine ID and a process ID in order to ensure that the developer is also able to access every virtual address at all times. Therefore, each virtual address is unambiguous.
If the software runs on a breakpoint, the entire system will be stopped as described above. The debugger then automatically switches to the (real) core and displays the machine and process on which hit the breakpoint. This allows the user to immediately see the conditions that led to this break. The current VM is called the "current machine". Naturally, it is possible to manually switch to other cores and their "current machines".

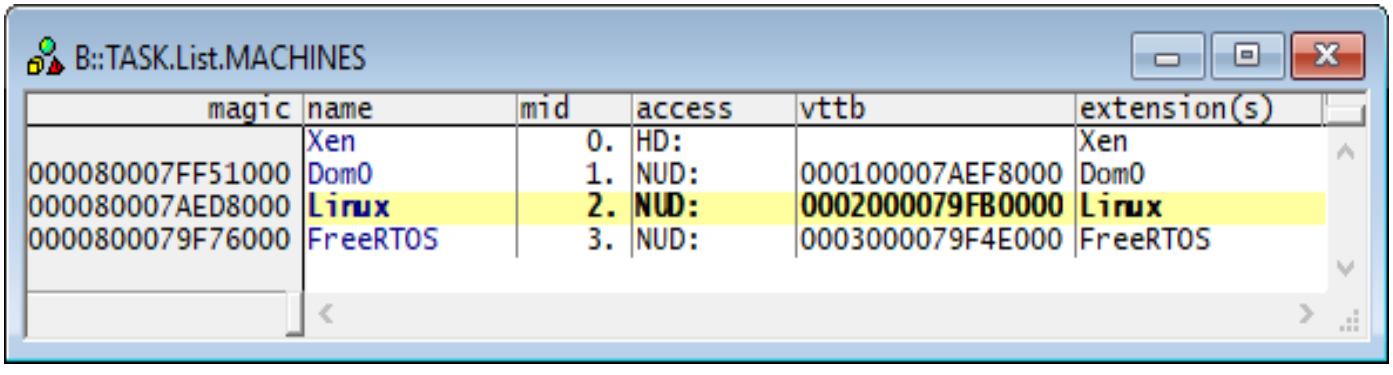

Figure 5: The debugger "knows" the hypervisor and the guest machines.

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Access to inactive guest systems

With this concept, the user is not only able to switch the view to other hardware cores; they can also switch to other, currently inactive guest systems. As a result, a symbolic access to all of the functions and variable of other machines is possible at all times. The symbols were loaded for a specific machine. The debugger translates the virtual address of the symbols into a physical address (it performs the MMU table walk itself) and, for instance, then reads the variable value from the physical RAM. In doing so, it is important that the CPU state is not changed at any time and that everything is performed within the debugger. By accessing the symbols belonging to all of the machines, it is also possible to set breakpoints to any function on any machine at all times. Of course, the debugger can also be switched to the registry set of a certain machine or process. If the registries are not loaded in a real core at this moment in time, the debugger reads the values for this from the hypervisor or guest system memory. Using these values, the debugger determines the current stack frame in order to; for instance, display the current call hierarchy of a task's functions. Straightaway, the developer sees the current progress of the task and why it may be potentially waiting.

How can these functions now be used in order to debug for the application mentioned at the start? As a reminder: The guest process did not receive a hardware interrupt. In fact, it is now easy to analyse this problem. Using the hardware debugger, a breakpoint is applied directly to the interrupt vector. The system will stop as soon as the interrupt is triggered. As the debugger is aware of all components, the developer is now able to follow the chain of events, i.e. the progress from the point of interrupt by the hypervisor, via the guest operating system and to the individual process, either in single step mode or through breakpoints in the respective steps. As a result, any fault that may have crept in is easy to find. However, deadlocks are more difficult to find if two communication processes mutually block each other for instance. In such a case, the system view helps as the states of all participating components can be illustrated next to each other. Consequently, it is easy to see which task of which guest – or even the hypervisor – is in an forbidden state or is potentially waiting for mutual resources. The post-mortem analysis option should not be underestimated, if the entire system gets into a state and no longer responds. Due to the fact that a hardware debugger does not require any active software on the target system, it is now able to investigate the state of every component. As TRACE32 from Lauterbach also contains an instruction set simulator, the developer can obtain complete memory dump from such a system and comfortably analyse it at a later date without true target hardware, in a similar way to a core dump analysis. However, this time across the entire system, including the hypervisor, all guests and all processes.

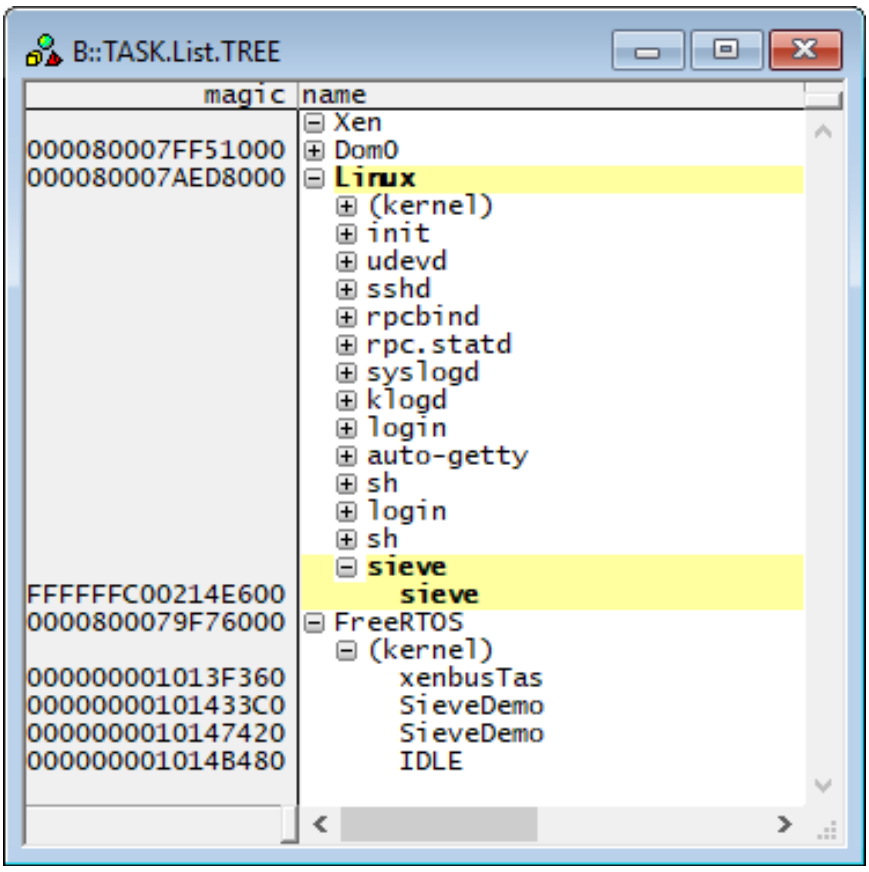

Figure 6: A tree structure illustrates the target system layout.

Lauterbach GmbH • Altlaufstraße 40 • D-85635 Höhenkirchen-Siegertsbrunn • Tel. +49 8102 9876-0 • sales@lauterbach.com • www.lauterbach.com

Lower costs, more complexity

Hypervisors are also increasingly being used in the embedded world. Advantages such as cost savings and runtime monitoring are clear arguments in its favour. However, this comes at the cost of higher system complexity. The hardware must provide virtualisation via a two-step MMU hierarchy that has to be managed by the hypervisor. A hardware-supported debugger (e.g. via JTAG) requires hypervisor and guest system knowledge in order to provide the developer with an insight into the software. For this purpose an "awareness" adapted to the respective hypervisor and the respective guest operating system is loaded on the debugger that reads the necessary information from the target system.

Lauterbach has created a reference implementation with the Xen hypervisor and the Linux and FreeRTOS guests on a Hikey board that demonstrates the functionality. The MMU support implemented in the TRACE32 debugger and an expansion of the address management to virtualised systems permit access to all components at all times. This enables a debugging of the hypervisor, all guest operating systems and all guest processes. Consequently, it is even possible for a retrospective analysis of a memory map without any problems whatsoever. Overall, the developer receives a tool that they can use to even effortlessly debug such complex systems.

.

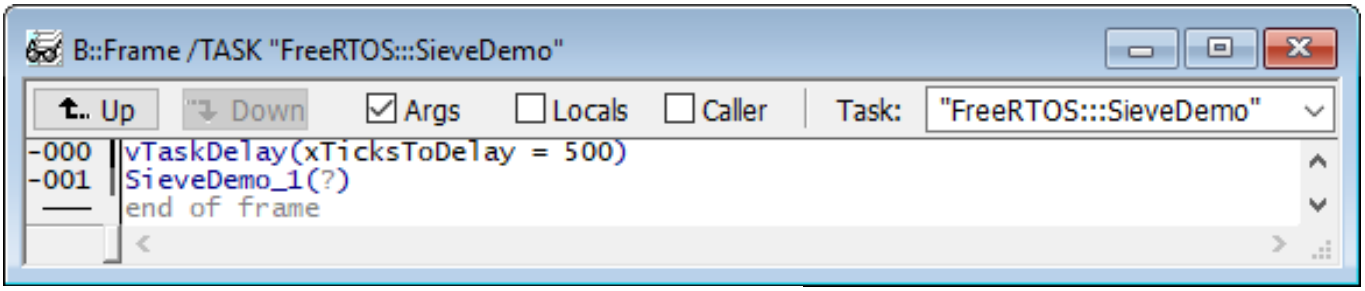

Figure 7: Call hierarchy of an inactive task within the inactive guest.

Lauterbach GmbH
Rudolf Dienstbeck
rudolf.dienstbeck@lauterbach.com

Published in Elektronik Magazine # 20
Oct 4th, 2017