

# Using Lauterbach to debug Linux systems

Embedded designs are becoming more and more complex especially when it involves Linux based systems. The least predictable stage in the development process – yet maybe the most critical - is to debug the system.

One of the hardest parts in embedded development is to understand the behavior of the system, and debuggers are the tool that allow you visibility into the inner workings of a Linux system.

There are in principle two alternative ways of debugging a Linux target: software based and hardware based. Hardware based debuggers - robust JTAG based debuggers like the Lauterbach TRACE32 – offer more functionality and control than software debuggers like GDB. Now we may ask – how and which functionalities?

## Full transparency between user space and kernel space

Since the TRACE32 provides both Linux and MMU support, the debugging of the kernel and across process boundaries is possible. With the Lauterbach debugger it's possible to start debugging a process/thread already from its initialization in the kernel.

For the GDB the debugging execution engine is part of the target program and therefore all software restrictions apply to the debugger too. The MMU of the processor "sees" only the kernel address space and the address space of the currently running target.

Thus the debugger also has access only to these parts of the software. There is no way of accessing data of a not yet scheduled process.

## No overhead or intrusion is added to the system

A JTAG debugger like Lauterbach does not use any hooks into the executing system and consumes no resources like CPU power or memory.

Compare this with a GDB - debugger which requires a "stub" or client running on the target for every debugged process.

## Full control of both physical and logical memory

This is especially useful when developing drivers since the only requirement for "Stop Mode Debugging" is a functioning JTAG interface - debugging can start at the reset vector.

Because of this, it's possible to profile the total Linux system or specific parts of it in a non intrusive way.

## Trapping segmentation faults before they are executed

Let's assume that a segmentation fault for a process happens from time to time. The Lauterbach debugger can be used to set software breakpoints in the kernel area that handles segmentation faults.

Later, when a segmentation fault occurs, the TRACE32 breaks when the system is about to execute a segmentation fault. Because these breakpoints are software breakpoints, handled from the JTAG debugger, the TRACE32 breaks before the segmentation fault has been executed.

Therefore it's possible to view data areas, local variables, registers and also the specific code line which made the segmentation fault to happen. A GDB debugger can not do this, it is even likely to crash if the application crashes because of an segmentation fault.

## Offers many different scenarios and ways to connect to the target

Software based debuggers, as GDB, usually use a standard interface to the target like a serial line or Ethernet. This implies that the target must be up and running and at least the interrupts for the interface line must be working. Hence no bootstrap, interrupt or post mortem debugging is possible.

The Lauterbach debugger on other hand does not require any software resources on the target and it uses either the JTAG/BDM port or "mictor" connector to handle the debugging. From that point of view its possible to use the serial line or Ethernet for other purposes than debugging.

## Complex breakpoints can be set – without affecting the system

First of all, using a Lauterbach debugger you can set breakpoints everywhere and on all levels. They can be task related, thread specific, can be set for specific functions, logical and physical memory both in user and kernel space.

When using GDB to debug a process a breakpoint halts only the desired process. The kernel and all other processes in the target continue to run.

This may be helpful if protocol stacks need to continue while debugging but is a hinder to the debugging of inter-process communication.

## Using the Lauterbach debugger as a GDB – debugger

Sometimes its desirable to start and stop processes independently to each other. Because of this its possible to transform the Lauterbach debugger to a software debugger, so that it works like the GDB.

This may be helpful if, for example, protocol stacks need to continue while debugging, when testing and debugging intercommunication between processes and of course when it's desirable to stop and run processes independently of each other.

The implementation is in principle the same as the native GDB e.g. running one instance of GDB for every process to be debugged.

## Requirements

Apart from being able to connect physically to the target using JTAG/BDM or a mictor connection, you need to compile the whole Linux kernel with debug symbols switched on - and that the symbols of the "vmlinux" file are loaded.



**Joakim Larsson, M.Sc E.E**

Solutions Architect

RTOS Integration

joakim.larsson@nohau.se

+46 (0) 708 83 84 83