Basics and tools for multi-core debugging

By Jens Braunes, PLS

In the past, debugging meant seeking for variables written with wrong values. These days, it's completely different: for the multi-core systems used nowadays in automotive control units, debugging means managing deadlocks, resource conflicts or timing issues of real-time applications.

The paradigm shift and the dramatic increase of complexity represent a big challenge for silicon vendors as well as for tool providers. And they can only master it together. The reason for this is because on-chip debug functions integrated by silicon vendors get fully effective only with powerful software tools which are able to completely utilize them and open a door for the developers for efficient use. If we look at the consumer market, multi-core systems have become mainstream since more than 10 years. But in deeply embedded systems, like motor control, the technology shift towards multi-core took place only in recent years and that often faint-heartedly. One reason is certainly the high demands on safety, reliability and real-time, and this has for sure the highest priority in the whole area of automotive applications. Another one is due to the existing huge portfolio of reinforced and well tested software modules for single-core systems whose porting to multiple heterogeneous cores would require a significant effort.

If we look at the world of PCs or consumer electronics dominated by Windows, Linux or Android operating systems then the CPUs used are based on homogeneous multi-core architectures. Identical cores with identical instruction sets, performance and interconnect to the other on-chip units allow executing any OS task or process by any core. Task creation and core allocation take place dynamically at run-time in order to balance the load and optimize the run-time behavior.

In the world of automotive control applications the multi-core approach is completely different. In general, tasks have dedicated processing times and slots, and must guarantee response within a specific time limit. And the tasks are heterogeneous. They have many different demands on performance, communication resources and instruction set features. For this reason, mostly heterogeneous multi-core architectures with several different cores, tailored to the needs of specific tasks are used.

One example of such a microcontroller is the AURIX from Infineon. This is a complete device family of multi-core controllers that are widely used in engine control units nowadays. Although the main cores all come from the TriCore architecture family, they differ in details. In some AURIX devices different flavors of the core architecture are implemented, e.g. performance cores (P-cores) and economy cores (E-cores). Some of the cores feature additional lockstep cores, enabling them to fulfill higher safety requirements stipulated by ISO26262, for example. The lockstep cores are based on the same core architecture and execute the same code. The results of both the actual computational core and the lockstep core are compared to each other and the reliability of the code execution and calculations checked permanently. If a deviation arises, the whole system has to be reset into a save state.

Optimized algorithms for advanced timing control, as needed for PWMs for electrical and hybrid drives, or for complex signal processing are supported by an additional core of AURIX, namely the GTM (Generic Timer Module), an intellectual property (IP) from Bosch. The GTM is completely different from the other TriCore-based cores. It has a lot of units dedicated for signal generation and processing as well as a number of processing units which can be programmed in a RISC-like manner.

The tasks executed by GTM are executed loosely coupled to the other TriCore tasks but can communicate using a kind of shared memory. It is quite obvious that the task allocation to the different cores is not only a question of load balancing. It is rather a question of which core is the most appropriate one for executing the task. That decision can hardly be made by an operation system during run-time. As a consequence, it has to be determined already during the design phase of the software which core will take over which task.

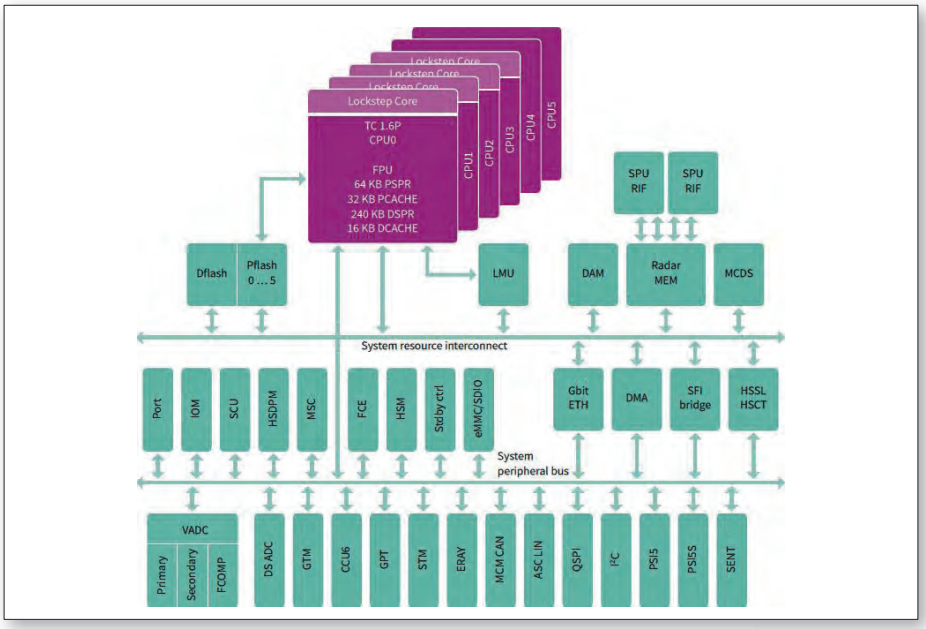

Figure 1. AURIX multi-core architecture (source: Infineon)

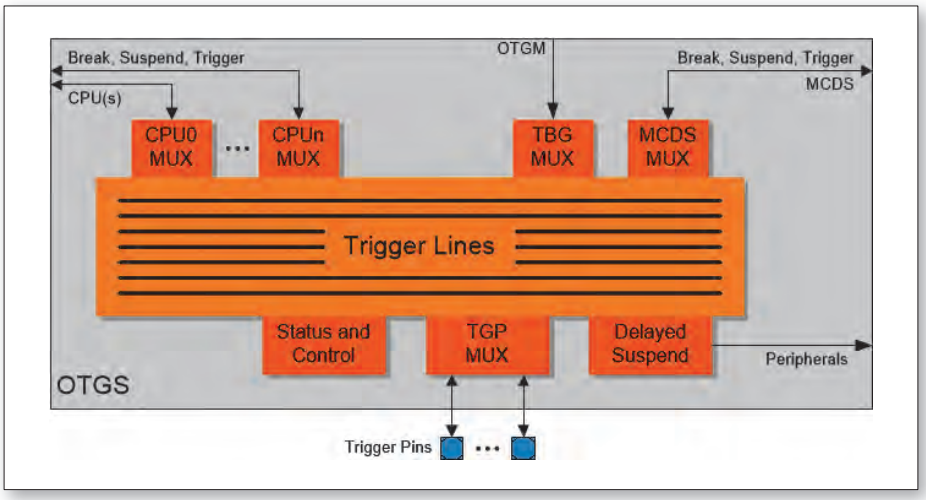

Figure 2. OCDS trigger switch of Infineon AURIX

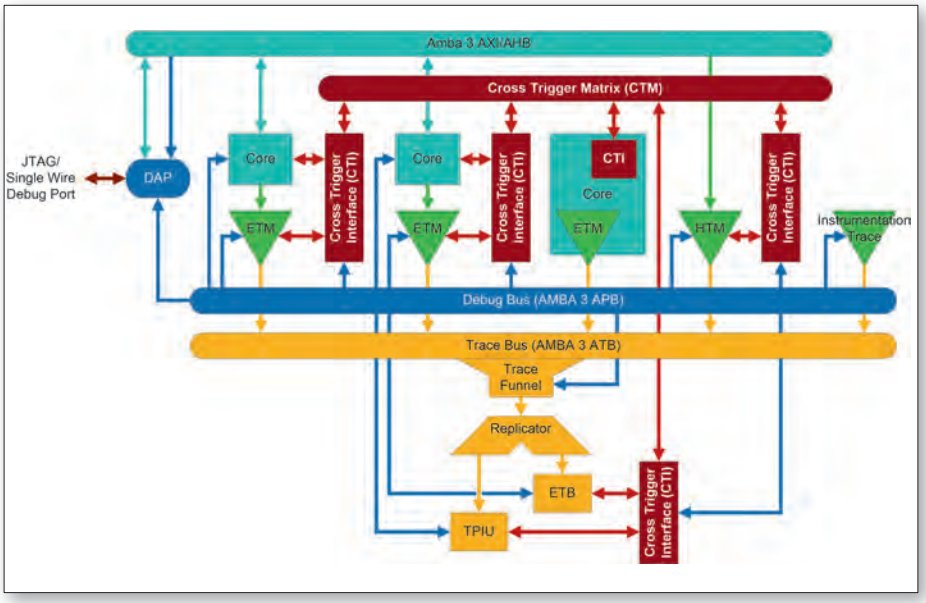

Figure 3. ARM CoreSight debug and trace infrastructure with cross-triggering

Those applications which are executing distributed across several cores and have to cope with high real-time demands are often the most challenging for debugging, test and system analysis. The typically existing large dependencies between the tasks running on different cores have a considerable influence on run-mode debugging or also known as stop-go debugging. It might be quite dangerous to break a single core while the others are kept running. In the worst case, the whole application would end up in chaos or crashes. Sometimes the other cores and also the peripherals have to be halted as well in order that the application does not get into an undefined state. The point is that heterogeneous cores with different clocking and execution pipelines do not allow a real synchronous stop. In practice, we will always have a delay. There is a complete opposite case; if for instance another, completely independent application is running in parallel on the same processor but using different cores, then it might be dangerous to halt the complete multi-core system. These scenarios show the importance of a flexible, synchronous run-control in a multi-core debug infrastructure.

A second, not less important aspect is the analysis of the run-time behavior without influencing it at the same time. This non-intrusive system observation plays an important role not only for real-time critical applications but also for profiling tasks or monitoring communication between cores. Often it is desirable to read out the system state from the target by means of the debugger at a certain point in time. However, if we halt the application for that purpose the system behavior would be fundamentally changed and has nothing to do anymore with the behavior of the application running later without an attached debugger. As a consequence, for an efficient non-intrusive system observation trace is indispensable.

Before we take a deeper look into trace features we will first come back to aspects dealing with synchronous run-control. Synchronous run-control necessarily requires short signal paths between the cores which can only be realized by on-chip debug hardware. Signaling of stop and go requests from the outside, for example from the debug probe via the debug interface, takes too much time, in particular for the high clock rates we have nowadays. And once the complete system is stopped finally the states of the individual tasks have lost their coherence completely.

All silicon vendors provide their own on-chip debug solution. There is no real standard at all in that area. Infineon for example is calling its solution OCDS (on-chip debug support). The central component of OCDS for run-control is a trigger switch, which propagates halt and

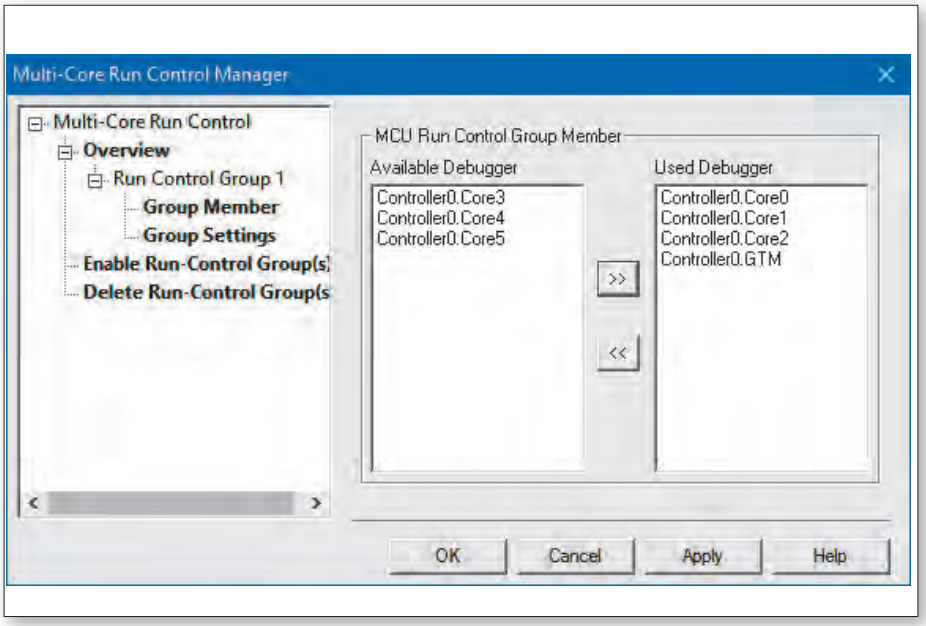

Figure 4. Multi-core run-control management of UDE

suspend signals via so-called trigger lines to all cores and also to peripherals. The trigger switch is configurable that individual cores as well as peripherals can be stopped and stated again at the same time and without having an effect on the others. In addition to that, the trigger lines can be connected to pins and make them available to the outside world. This offers interesting opportunities. For example, signals can be connected to an oscilloscope or a break can be triggered externally.

Of course, besides the AURIX, a number of other microcontroller architectures exist, which are used for automotive applications. Two other examples are the SoCs based on the ARM Cortex-R architecture and the PowerArchitecture based SPC5 from STMicroelectronics. Both bring along an own implementation of on-chip run-control support. On the ARM side, it is called CoreSight. Let's have a look at this.

In CoreSight a so called cross-trigger matrix (CTM) is used in order to propagate break and go signals across the cores. The cores themselves can trigger such signals and respond to them but not directly. A cross-trigger interface (CTI) attached to each core takes care of it. Up to four channels in a CTM broadcast the signals to all attached CTIs. The CTIs can be configured that way, for either passing the run-control signals to the core or blocking them. Thus, simply a core gets halted along with others or not. Because of hand-shake mechanisms, which are necessary between different components, there is a little delay of several clock cycles. The actual amount of that delay highly depends on the implementation. In fact, avoiding it is technically not possible. One drawback of the ARM solution is that CoreSight is in fact a set of components and IP blocks from which silicon vendors can choose. As a consequence, debug tool vendors cannot rely on the existence of CTMs and CTIs in a particular multi-core SoC.

As expected, the PowerArchitecture based controllers of the SPC5 family support synchronous run-control by means of hardware. The unit in charge is called DCI (Debug and Calibration Interface). The advantage compared to the CoreSight is that, as we already know from the trigger- switch of AURIX, peripherals are also connected to the debug signals. That allows halting the complete system, not only the cores.

In real life developers don't want to take care of all these details. For this reason multi-core debuggers like the Universal Debug Engine® (UDE) from PLS make the complex configuration of on-chip debug units transparent to the users. The integrated run-control management, for example, easily allows creating run-control groups containing all the cores which should be stopped and started synchronously.

Especially when it comes to debugging and system analysis of real-time applications, on-chip trace is mandatory and is available for almost all high-end microcontrollers like the AURIX or SPC5. STMicroelectronics for example implements Nexus class 3 for tracing, for Infineon microcontrollers the on-chip trace is called MCDS (Multi-Core-Debug Solution) and for ARM trace hardware blocks come from the already known CoreSight. They all have in common that they are able to capture trace data from different cores in parallel. Timestamps allow a time correlation between the data of the different trace sources and thus we can reconstruct the exact sequence of events. This allows us to detect deadlocks and race conditions and communication bottlenecks can be found too.

Now, the most challenging task is transferring the captured trace data off-chip in order to analyze them by the debugger. From the current trace systems we know two ways to do so. Either the trace data is buffered in an on-chip trace memory and transferred via the standard debug interface, or a high bandwidth trace interface exists. The first allows a much higher bandwidth between the trace sources (CPUs, busses) and the trace sink, namely the on-chip trace memory. The major drawback is the very limited capacity. The later allows a theoretically unlimited observation period, but the bandwidth is in fact the limiting parameter. For both cases clever filter and trigger mechanisms as part of the trace solutions can help. These allow qualifying the captured trace data while they are created in order to record only the really necessary data. With it cross-triggering is also possible. Cross-triggering allows, for example, starting the trace recording for one core if a specific event arises at another core. That function is helpful for debugging of inter-core communication.

Experience has shown that for an effective use of trace the user needs comprehensive support by debug tools. That is true not only for the analysis of the recorded data but also for the definition of trace tasks and the configuration of the filters and triggers. UDE, for example, even provides a graphical tool for that purpose which allows managing even complex cross-triggers easily. Several tools help to analyze the recorded trace: from visualization of parallel execution of cores, via profiling, to providing code coverage information. ■