Practical aspects of on-chip debug and test infrastructures

By Jens Braunes, PLS

The more complex the embedded application, the more expensive it usually is to troubleshoot and test the entire system. How efficiently you can actually work with multicore SoCs nowadays therefore depends significantly on the internal debug infrastructure of the particular chip. With a view to the most popular architectures, we reveal what needs to be taken into account in practice.

Experience has shown that chip manufacturers still do not invest very much in the debug infrastructure for low-cost standard systems. The situation is completely different in the automotive industry and industrial sector, where extremely powerful multicore controllers are increasingly being used. In terms of software development, very much higher demands are placed on debugging and observability of the systems. This also affects the available interfaces and the on-chip debug functions.

Debugging and testing on real hardware depends crucially upon an efficient access to the system and the possibility to observe the system state from the outside. In the simplest case, the software itself reports its current state to the outside via a terminal or serial interface. This is known as printf debugging.

Dedicated debug interfaces allow a much more efficient and comfortable access. However, this is not available for free. JTAG (according to IEEE 1149.1) is certainly the most widespread. Originally developed for testing integrated circuits, JTAG still assumes the role of de facto standard for debug access. With at least five JTAG pins, and often additional chip-specific pins for reset, reference voltage and vendor-specific signals used by the on-chip debug system, the implementation is relatively complex. The related costs are hardly acceptable, especially for small microcontrollers and for end products in the medium or low-price segment. In addition, the achievable speed and robustness against disturbances is in the meantime far from state-of-the-art.

Unlike the standardized JTAG, the available alternatives are mainly vendor-specific. Due to the market dominance of some microcontroller architectures, however, some quasi-industry standards have developed. In any case, the following should be mentioned here: Serial Wire Debug (SWD) of the ARM CoreSight Debug and Trace IP (intellectual property), Device Access Port (DAP) from Infineon and Low Pin Debug (LPD) from Renesas. As a true successor for JTAG, and officially standardized, is cJTAG (IEEE 1149.7). cJTAG is used for some devices from NXP. However, the current market development, especially due to the concentration on ARM Cortex and thus on SWD, does not show a significant use of cJTAG outside of the NXP portfolio.

All of the mentioned interfaces get by with fewer pins, but at the same time often achieve higher transfer speeds and are also more robust against data transmission disturbances due to built-in error detection in the protocols. Another common feature is that these interfaces essentially use a single bi-directional data pin and a clock pin.

Furthermore, for DAP and LPD, there are also variants that use multiple data pins. DAP in wide mode, for example, uses an additional pin and thus doubles the data rate. Even in the two-pin variant, DAP impresses with its performance. Data can be transferred between the debugger and the target at a maximum rate of 160 MHz, which ultimately corresponds to a data rate of up to 15 Mbyte/s for block transfers. Infineon has created a very special solution, the single-pin DAP (SPD), which requires only one pin and is based on encoding the clock in the data signal. Although this approach results in lower speeds, SPD is in particular suitable for transmitting debug signals over CAN.

cJTAG, which is backward compatible to JTAG, offers some interesting additional features such as support of multiple test access ports (TAPs) and power management. As a result, it is better suited for multicore and multiprocessor systems.

Multiprocessor systems are also a focus of ARM. The new SWD protocol version 2, which was introduced with the CoreSight SoC-600 specification, allows the so-called multi-drop architecture which addresses

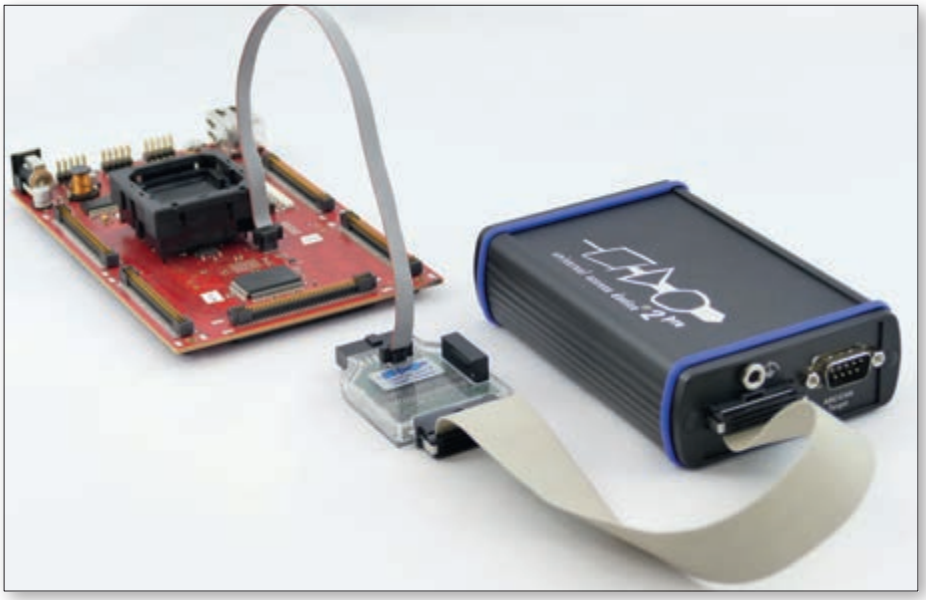

Figure 1. Use of a dedicated debug interface by the example of Infineon DAP and the Universal Access Device 2pro from PLS. A 10-pin standard connector is used which provides, besides the DAP signals, also further pins for reset, I/O voltage and ground.

	Class 1	Class 2	Class 3	Class 4
STATIC DEVELOPMENT FEATURES				
Read or write user registers and memory in debug mode	☐	☐	☐	☐
Single-step instruction in user mode and re-enter debug mode	☐	☐	☐	☐
Enter / exit a debug mode from / to user mode	☐	☐	☐	☐
Stop program execution on instruction/data breakpoint and enter debug mode (minimum 2 breakpoints)	☐	☐	☐	☐
DYNAMIC DEVELOPMENT FEATURES				
Ability to set breakpoint or watchpoint	☐	☐	☐	☐
Read or write memory locations while program runs in real time			☐	☐

Figure 2. Extract from the Nexus Compliance Classes. Static development features are available while the target is halted, and dynamic development features are available also while the target is running.

multiple processors in a multiprocessor system via a single debug interface. Besides the approaches described, there have also been efforts in recent years, if possible, to avoid dedicated debug interfaces completely in order to save costs. Instead, functional interfaces such as, for example, CAN, Ethernet or USB should be used for debugging, because these are mostly already available on the chip.

For data transfer via the CAN bus, for example, the single-pin DAP (SPD) already mentioned is suitable. This has led to the development of DAP over CAN Physical Layer (DXCPL). Due to the low speed of CAN, the achievable transfer speed is only 10 to 40 KByte/s. DXCPL is therefore used primarily for debugging in the field when the actual debug interface is no longer accessible, for example, by the housing of an electronic control unit (ECU).

Another interesting alternative could be the use of other standardized interfaces, for example, from the area of calibration. A working group of the Association for Standardization of Automation and Measuring Systems (ASAM) is currently pursuing the goal of utilizing the Universal Measurement and Calibration Protocol (XCP) – which so far has been used exclusively to connect ECUs with calibration systems – for debugging as well. This approach will, in future, make it possible to debug ECU software under real and even extreme conditions.

ARM SoC-600 goes much further. With this technology, in future debugging will be possible using almost any functional interface. The advantages include the following. Debugging could then still be carried out in the field when traditional debug interfaces are no longer accessible. And expensive specialized hardware on the target side and on the tool side needed for debug access could be replaced by cost-efficient standard components and IP blocks that already exist anyway. Among the disadvantages is that fewer interfaces may eventually be available for the actual application. In addition, the application itself must take over the initialization in order to open the functional interfaces for the debug channel.

The debug system on the chip is even more important than the debug interfaces. This is because the capabilities of system observation and control rely entirely on them, that is to say, what the debugger software can achieve on the PC in the end. The debug infrastructure, also often referred to as on-chip debug system, has two basic tasks.

Firstly providing target information, such as memory and register contents or the status. Of course, it should be possible for the developer to modify them. Secondly

control of program execution on the target. This includes breaking and restarting the application triggered by the debugger, as well as single-step operation and hardware breakpoints.

As in the case of interfaces, the debug infrastructure is also highly vendor- and architecture-specific. Ultimately, however, the debugger ensures that the developer hardly has to worry about it.

The only real standard to be highlighted here is Nexus (IEEE-ISTO 5001). The Nexus standard defines four compliance classes – each with a fixed set of debug functions – where each higher class also contains all functions defined by the lower class. To ensure compliance with a particular class, the chip vendor must implement at least the debug functions required by that class. Whereby the two basic tasks described above are already covered by Nexus Class 1 criteria. Nexus-compliant implementations can be found primarily in Power Architecture based controllers from NXP and STMicroelectronics, but also Renesas relies on Nexus for some of the RH850 devices.

A complete debug IP with breakpoints, watchpoints (breakpoints for data accesses), read and write of memory contents by the debugger while the system is running, trace and cross-triggering functionality etc is provided by ARM with its previously mentioned CoreSight. However, which functions from the CoreSight kit are available in the actual chip depends on the respective semiconductor vendor who is licensing the ARM IP. The semiconductor vendor has a high degree of

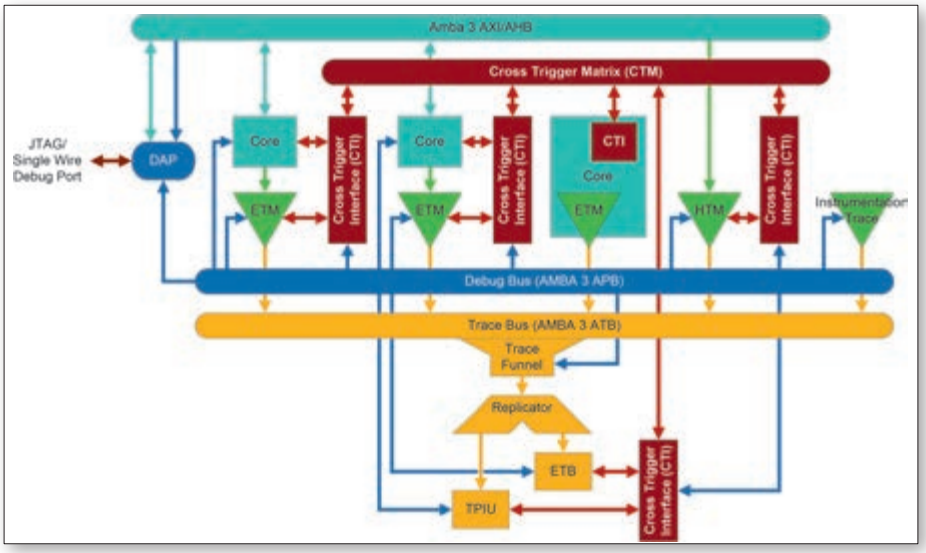

Figure 3. System overview of the ARM CoreSight debug and trace infrastructure. CoreSight defines components for target access and traditional stop-go debugging as well as for cross-triggering and trace.

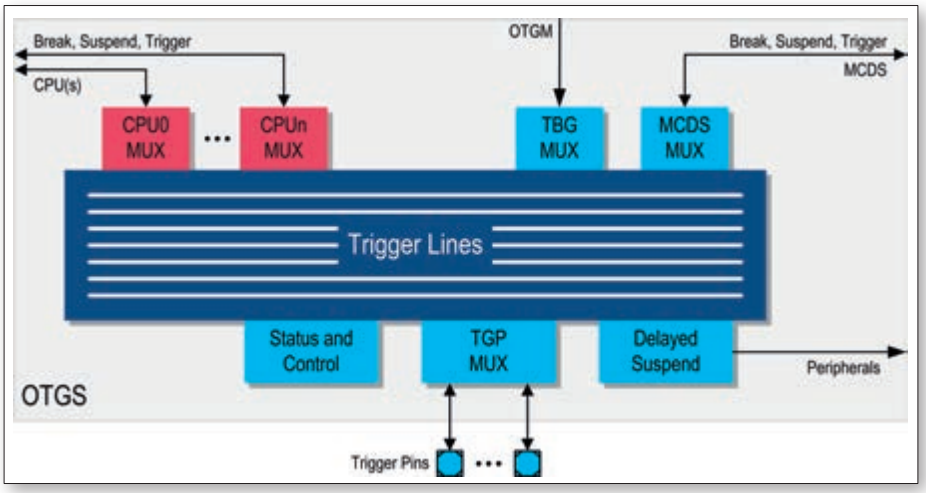

Figure 4. With Infineon OCDS Trigger Switch, the debugger can implement quasi-synchronous breaking and restarting of multiple cores. Which cores, for example, should halt at the same time at a breakpoint can be defined by configuration of the trigger lines.

freedom in configuring its implementation. Customer requirements certainly play a role here, but cost is the most important factor.

On-Chip Debug Support (OCDS), which is used for Infineon architectures (TriCore, C16x and successors), is completely proprietary. Support for breakpoints, data breakpoints (at least for data addresses) and access to memory and registers at runtime for the debug tools are, of course, also provided here. Furthermore, with the AURIX family, a kind of cross-trigger mechanism has been added, which is here referred to as OCDS Trigger Switch.

It is in the nature of things that in all on-chip debug systems the number of available hardware breakpoints is limited. After all, these are nothing more than address comparators for code addresses and, if available, also for data addresses and thus real hardware units. If one of the comparators signals a hit, a debug action, for example a HALT signal, is triggered. The configuration of the comparators is hidden to the user by the debugger, so that at the end he can simply put a breakpoint in a certain line of the code. If all hardware breakpoints have already been used, software breakpoints can be used instead. Primarily these are realized by a technique called code patching: the machine instruction at the desired breakpoint location is temporarily replaced by a special breakpoint instruction or sometimes also by an illegal instruction. A trap triggered by the execution of this instruction can be caught by the debugger, the code patch is undone and the user gets offered the halted system. However, this only works if the code is executed from RAM, because only there can a code patch be done easily. For flash located applications the effort involved is much higher. Complete flash regions would need to be reprogrammed at the cost of lifetime of the flash memory. In that case it is a good idea to get along with the actually available hardware breakpoints.

With the introduction of multicore into embedded systems, there was, of course, at the same time a need for extended debug functions. Especially the run control – breaking, stepping and starting – deserves special attention here. Depending on the application, several cores have to be synchronized with each other so that, for example, halting them at the same time at a breakpoint is possible. Due to the significant differences between the clock frequencies of the cores and the speed of the debug interfaces, external synchronization by the debugger does not lead to the desired result. The signaling that, for example, all cores should halt simultaneously needs to take place directly on the chip. The already previously mentioned cross-triggering is responsible for this. Due to signal delays, different clock frequencies and clock domains of the individual cores and also the pipelines, latencies can however never be ruled out. For example, breaking, single-stepping or starting again of multiple cores is only quasi-synchronous, whereby the slippage, with a few clock cycles or instruction cycles, is low.

Debugging of multicore applications is, of course, a real art and places the highest demands on the debug infrastructure, on the debugger and, last but not least, on the user. At the very least, this is where "printf" has no place anymore. ■