

Trends in Debugging Technology

Published: Embedded Systems Conference East, Chicago

Date: March 1998

*By Gerard Vink
TASKING Software BV
Amersfoort, The Netherlands*

Introduction

What will happen to the debugger during the next decade? Will the debugger (as we know it) change, disappear, or stay the same. Given the fact that debugging still takes 20-50% of the total development time of an embedded project, most managers would hope the debugger will disappear.

The breakpoint debug paradigm has been in use for over 20 years and during this time the most significant changes towards user interaction have been "symbolic debugging" and "graphical user interfaces". Execution environments have allowed more progress to be made. Previously only monitors were used, but now a plethora of execution environments such as simulators, Background Debug Mode (JTAG, BDM), emulators, and on-chip emulation (OnCE) are available for controlling program execution.

The debugger will not disappear but it will hide inside high level design tools. Software will be developed and debugged at higher abstraction levels than the JAVA, C or C++ level.

This paper outlines current trends in debug technology and discusses how this changes user interaction with the debugging system. The focus is on high level language software debugging and software-hardware interface debugging in an embedded environment. Since trends are influenced by history and the current environment the questions "where are we", "how did we get here" and "where do we go to" in respect to debugging technology are presented. The following trends are identified and discussed:

- The debugger architecture
- Integration in design tools
- Debugger languages
- Debugging of highly optimized code
- Feedback loops
- Tight integration with compiler (patching)
- Simulation, peripheral simulation and co-simulation
- On-chip emulation
- Debugging multi-tasking, multi-processor (parallel) systems and multi-core processor designs
- Changing the debug paradigm
- Checkpointing and reverse execution

Debugger Architecture

Let's first identify the components of a typical debugger for embedded applications. The main components are shown in figure 1. The most visible component is the user interface. Interaction between the user and debug system is accomplished via a graphical user interface. Typically this is GUI interface is combined with ASCII command language extended by some script language. The debugger core is the heart of the debugger. In a well-designed debugger the major part of the debugger is target processor independent. A processor behavioral module (not shown in figure 1) isolates target specific issues from the debugger core to allow fast retargeting. The debugger loads the application being debugged via a "reader". The application being debugged is run on the "execution environment". If the debugger is Real-Time Operating System (RTOS) aware, the "kernel support module" contains all kernel specific logic.

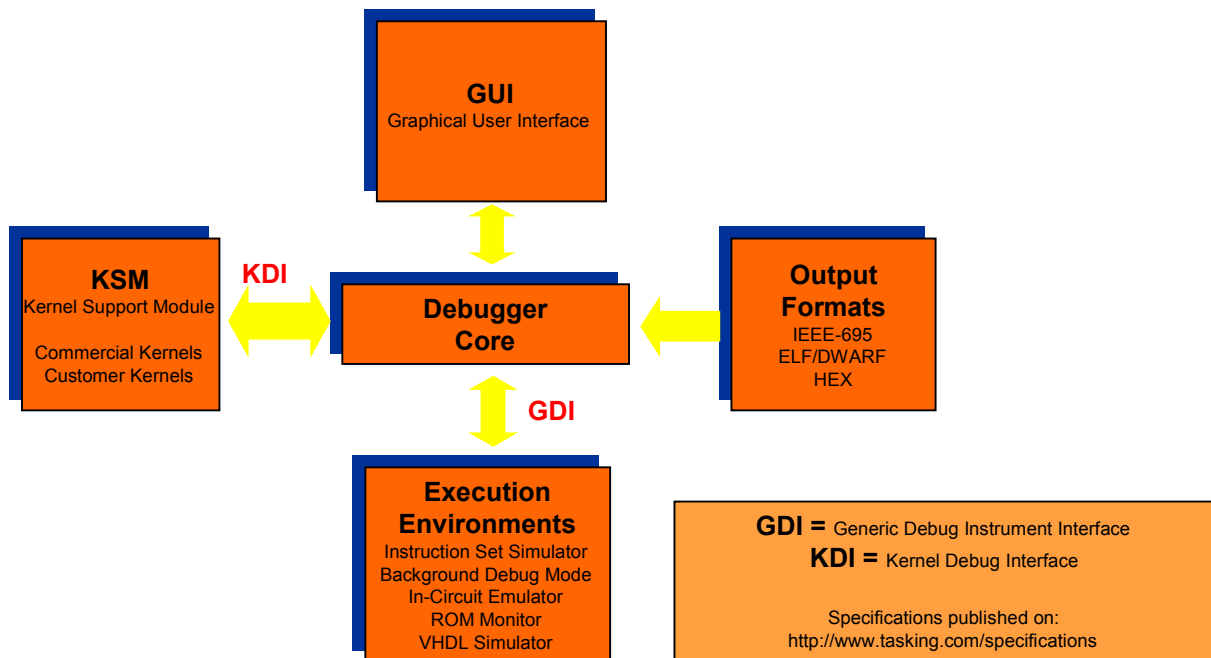


Figure 1, Debugger Architecture

GDI: OMI 325 Standard for Generic Debug Instrument Interface, released via the ESPRIT¹ OMI/Standards² project, specifies a target and execution environment independent interface between debugger core and execution environment.

¹ . **Esprit**, the information technologies (IT) program, is an integrated program of industrial R&D projects and technology take-up measures. It is managed by DG III, the Directorate General for Industry of the European Commission. Esprit focuses on eight intertwined areas of research of which software technologies is one. Software Technologies aims to maintain a strong base of high quality and relevant skills and key technologies within all sectors of the European economy for which software development forms an important component of business activity. Technologies for Components and Subsystems concerns the development and broad exploitation of a wide range of microelectronics solutions for electronic systems. Work encompasses equipment, materials and processes used in manufacturing semiconductors, through to electronic design tools, packaging and interconnect solutions. The area includes work on peripheral subsystems such as storage and displays, and work on microsystems. For more information see: <http://www.cordis.lu/esprit/home.html>.

KDI, Kernel Debug Interface, developed in the framework of the ESPRIT project 7325 OMI/Debug, specifies a target and kernel independent interface between kernel support module and debugger core.

Both specifications (and others) are public available. For more information about these standards refer to the OMI web site <http://www.omimo.be/develop/omidev/2home.htm>.

The User

Debug technology can be discussed from a scientific or technical perspective only, but all this wonderful technology is of no importance if it's not accepted and used by the people who develop embedded systems. The skills and interest of embedded software developers has changed during the nineties. A decade ago most embedded software developers had their roots in electrical engineering. They were familiar with assembly level programming and with the hardware their software was interacting with. Nowadays the embedded engineer is often specialized in application domain specific engineering problems, instead of low-level HW-SW interface problems. This is especially true in high end 32-bit and DSP environments. The complexity of the overall system can not be understood by one person, but is shared by members of the design/engineering team.

The Design Process and Tools

As system complexity increases the design process and the tools to support this process are changing. Simple systems are sometimes still developed without "design", however, most companies prescribe some method for system design. These design methods vary. Natural language and informal graphics are most frequently used. More formal methods such as SA/SD as developed by Yourdon, Ward & Mellor and refined by Hatley & Pirbhai where the hype of the early nineties. The SA/SD hype was followed by the Object Oriented design hype. Industry has not adopted these methods as fast as expected but we see that they still gain popularity and have positive effects on lead time and quality. There is an extensive set of tools available in the market that support the SA/SD and Object oriented design methods, ranging from share-ware to mature systems like TeamWork and Geode. Formal system specification languages like the Vienna Development Method (VDM) are hardly used in industry. Figure 2 shows the usage of the discussed design methodologies over time.

² **OMI** is the Open Microprocessor systems Initiative, which is a focused cluster in the ESPRIT program, the IT area of the European Union's Framework IV program. OMI Brings together:

- Silicon Manufacturers (which are also IP (Intellectual Property) sources)
- Independent IP providers
- Systems tools companies and some smaller EDA vendors
- Embedded software houses (compilers, operating systems, RTOS (Real Time Operating Systems), ranges of kernels)
- System integrators making the final applications

with the objective to improve (re)usability and productivity, and hence to reduce time to market and improve quality and reliability in the target area of Embedded Systems, and specifically in the areas of: telecommunications, automotive and transport, consumer and multimedia, smartcards, industrial and process control, and some of the emerging markets.

Much of the early work was geared to making the "system on a chip" more open and allowing the trading of IP between companies. Various standards and guidelines were developed in this process, with real applications providing the market pull via demonstrators to prove the concepts in practice.

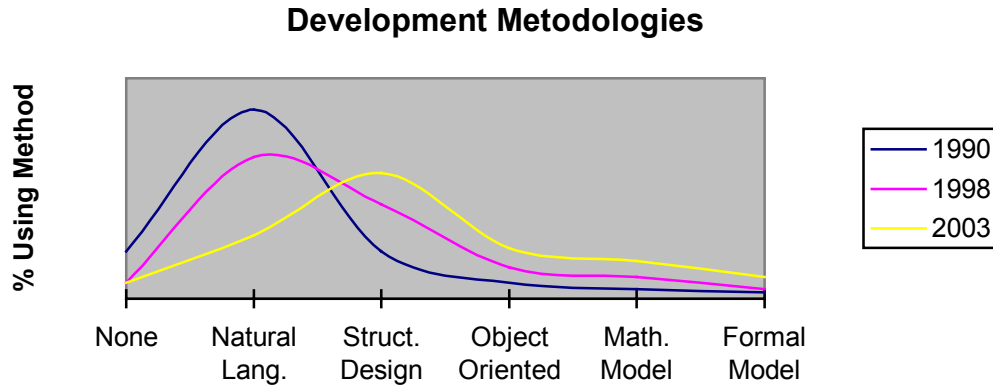


Figure 2, Design methodologies over time

Next to tools for design method support, specific tools for fuzzy logic system design and mathematical modeling are gaining popularity.

The User Interface

Currently most commercial available debuggers are equipped with a full blown graphical user interface (GUI). The advantages of the GUI are well known, but one of the most important advantages is reducing the learning curve. During the development cycle a debugger is used for different purposes. In the early stages of the development cycle the debugger is used to become familiar with the hardware within the system, i.e., to play with and comprehend the capabilities of the hardware. Later on, the debugger is used for its main purpose, to debug the system. Towards the end of the development cycle its role as (regression) test and validation tool becomes more important. Features for test and validation like coverage and profiling are often implemented in debuggers. Since debugging is an interactive process and testing is batch oriented the user interface (UI) requirements differ also. The record and playback features that existed since the seventies and that disappeared in the GUI age are now returning.

Generally you want to debug a system at the highest possible abstraction level. Currently debuggers provide a view at the high level language (HLL) C++, C or JAVA level. If a bug is introduced due to problems not visible within the HLL scope, assembly level debugging comes into play.

Since systems will be designed and completely described at higher abstraction levels than the HLL level, the debugger will be embedded in the new design and verification tools. This issue is currently addressed in several OMI projects. One example is the OMI/TREFAB project, for which the objective is to create a rapid development environment for the automotive domain, specifically the Brake by Wire (BBW) system. An open, coherent and integrated framework and toolchain is composed of a fuzzy logic design tool, a mathematical modeling tool, a compiler and C-level debugger. These tools are integrated as outlined in figure 3 and provide a simulation and debug environment at the specification level. For example when execution is halted the debugger passes the application state to the fuzzy design tool which shows the triggering of the fuzzy rules.

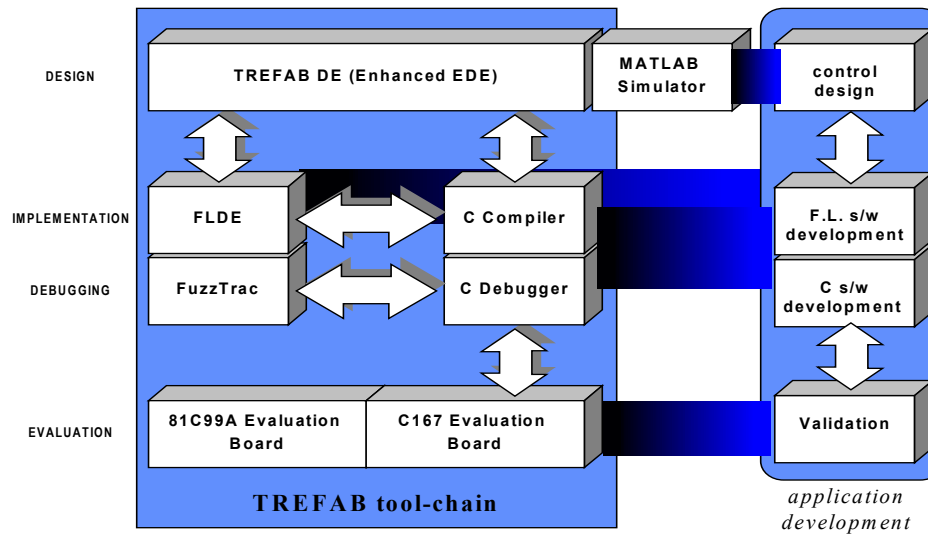


Figure 3, The development of BBW with TREFAB toolchain.

Standardization of Interfaces

Standardization of interfaces is fueled by two interests. First silicon vendors are a driving force. Excellent tool support is one of the most important requirements for success when launching a new μ controller core. However the typical case is that compilers of vendor X don't work with debuggers of vendor Y which don't work with simulator engines developed by vendor Z.

Combining the tools of different vendors is only possible if the interfaces between the tools are well specified. For example, Motorola has expended considerable effort in standardizing the object format (ELF/DWARF). The benefit is that all third party software libraries can be used in combination with any toolchain and that every debugger understands the object format. However this standardization also has a drawback. The support for debugging optimized code is minimal in DWARF1. IEEE-695 can be considered as an alternative for ELF. The major drawback of IEEE-695 is that it does not describe the syntax and semantics for debug information. The HP/MRI extensions for IEEE-695 can't be considered as a standard but are a guideline for storing debug information. Every compiler vendor makes target processor specific extension to this "standard".

The second driving force towards standardization of interfaces is the tools industry. At least some companies in this branch of trade are opening their tool interfaces. This is rather surprising because historically vendors opposed interoperability of tools to protect their market share. This attitude is changing because it is recognized that a large percentage of the R&D costs are associated with relatively simple (and boring) interface related work instead of enhancing technology.

Instruction & Peripheral Simulation

The interrupt driven nature of most embedded applications significantly affects the performance of the hardware and software. Interrupt processing influences: cache hit ratios, instruction pipeline scheduling, and the performance of a real-time kernel. The market demands tool support for analyzing both the hardware and software aspects of interrupt processing caused by peripheral interaction. Also the embedded systems developer wants to simulate peripheral behavior with respect to the interaction with the microcontroller's core as well as to the user interface of the embedded system with its environment.

Simulation and peripheral simulation are of interest for different activities: CPU core design, ASIC design, and embedded system and application design. Each activity has its own specific requirements.

Peripheral simulation is in all cases of particular interest because the generated interrupts affect overall system behavior and performance.

The CPU core designer is primarily interested in hardware related issues, such as: effects on cache and pipeline design, effects on pipeline behavior and interactions between pipelines, and the trade off between adding intelligence to the CPU core versus to the peripherals.

The ASIC designer is focussed on hardware related issues such as: effects of different peripheral designs on system performance, effects of different instruction and/or data cache designs and size and external RAM and ROM partitioning.

The embedded application designer's interest is software and user interface issues such as: correctness of application code, the behavior of the real-time kernel and interrupt handling of his software.

The basic features a simulator should provide for these target groups are: gather performance data, simulate application behavior, and visualize peripheral behavior

Profiling, supported by either the compiler or from within the execution environment is a technique for gathering performance data. To provide detailed and accurate data the execution environment must in detail behave like “the real thing”. This means that if simulation techniques are used: the simulator must correctly implement timing related issues regarding: the cache and the instruction pipeline(s), register renaming, and others.

The CPU and ASIC designer often desire cycle accurate simulation, but in some cases instruction accurate simulation is sufficient. From the perspective of the embedded application developer cycle accurate simulation and often instruction accurate simulation is not a requirement. His main interest is to test and verify the interrupt handling of his software. Next to this, for him, peripheral simulation is also demanded on “non-simulator” execution environments such as: an emulator or on a ROM-monitor.

The main discriminating characteristic of simulator engines is their accuracy regarding timing issues. A simulator can be classified as: a cycle accurate simulator, an instruction accurate simulator, and an instruction set simulator.

Cycle Accurate Simulation

The cycle accurate simulation model implements the exact behavior of the μ controller. The pipeline behavior, memory interface and bus protocols are simulated in detail. CPU stalls due to bus activity or register file ports and stalls due to data contention are also simulated. The CPU model to be simulated is often implemented in VHDL or C.

The purpose of a cycle accurate simulation model (implemented in a HLL) is threefold: 1) validation of the corresponding VHDL model, 2) fast (compared to VHDL) simulation of CPU core behavior, and 3) profiling alternative core designs.

Instruction Accurate Simulation

The instruction accurate simulation model implements an algorithmic model of the μ controller. It processes the application program as a sequential list of instructions without regard to pipeline behavior, memory interface and bus protocols. Interrupts never interrupt the processing of an instruction, i.e. are serviced after completion of the current instruction. However, timing information provided by the simulator is expected to be correct with respect to latencies, memory map, register dependencies and instruction type.

The purpose of a cycle accurate simulation model is to enable customers to: evaluate chip performance and functionality before silicon is available, tune chip/system performance with application specific test programs, and avoid installation and configuration problems associated with evaluation boards.

Instruction Set Simulation

The instruction set simulator executes the application program as a sequential list of instructions without taking notice of any timing constraints. Interrupts are serviced after completing the current instruction(s). Simulation speed is of more interest than timing accuracy.

The purpose of instruction set simulation is to: Test correctness of application algorithms. Test interrupt handlers of application code. Test correctness of compiler generated code. Avoid problems and costs with evaluation boards and emulators. Visualize system/peripheral behavior.

Simulation Speed

Simulation speed, measured in instructions per second, is of critical importance for every type of simulator. Of course simulation accuracy significantly influences simulation speed.

Table 1 gives an indication of simulation speed for the diverse simulator types. It's assumed that the simulator engine runs on a high-end PC (Pentium 166), which is comparable in speed to a low-end work station.

	Instructions per second
VHDL simulation	1 - 1K
Cycle accurate simulation	1K
Instruction accurate	10K - 100K
Instruction set simulation	100K - 2M

Table 1, Simulation Speed

Peripheral Simulation

Since the interrupts generated by peripheral have significant influence on overall system performance and peripherals are system specific, it is important that it is relatively easy to describe peripheral behavior and to include this model in the simulation engine. A peripheral behavior model can be described using compilable or interpretable languages. Both approaches have specific strengths, and are currently used. Interpreted languages offer the user a rapid edit-debug cycle while implementing the peripheral. However, the drawback of interpreted languages are slow simulation speed and often less accurate simulation. Several debugger vendors have published APIs for interfacing peripheral models written in C to their simulation engine. Standardization of these APIs is difficult since the API significantly influences the design of the simulation engine and it's run-time performance.

On-chip Emulation

The trend is easily identifiable on-chip emulation is implemented by every μ controller manufacturer whether it's Motorola, Siemens, ARM or Intel. The reasons are clear, the problems with standard ICES are:

- An ICE's pod interferes with the normal timing of the target system.
- Physical processor replacement requires intricate pods that are hard to connect to the system.
- Typically 6 months between processor release and ICE availability.
- A deeply-embedded core requires a large "bond-out" to make internal signals visible
- Custom variants of a processor family may not be supported by ICE at all.
- ICE can be very expensive

The benefits of on-chip emulation are:

- No target resources or special hardware are required. The on-chip debug unit is often access via the standard JTAG port.
- Debugging takes place on "the real thing", the same processor that shipped to the customer. So the change for *Heisenbug*³ to occur is minimal.
- Low cost solution.

Modern on-chip debug units support features such as: multiple complex breakpoints and watchpoints on data access and instruction fetch. Even realtime trace is not out of the question anymore.

Not every debugger fully supports the features offered by the on-chip emulator. Many debuggers use the BDM or JTAG link to the on-chip emulator as an ennobled communication interface only. Instead of using the available on-chip debug facilities, code-breakpoints are still implemented by patching in "trap" instructions while data-breakpoints⁴ are not supported at all. Such implementations do not support debugging of code located in ROM, since they can not install code-breakpoints in read-only memory. Currently more and more vendors take the effort to adapt their debugger to the specific capabilities of on-chip emulators.

Hardware-Software Co-Design

In the most general case codesign is just concurrent design (engineering) of hardware and software. Figure 4 shows the place of co-design in the product lifecycle. Co-design focuses on the areas of system specification, architectural design, hardware-software partitioning, and the iteration between the hardware and software as the design progresses. Co-verification focuses on the hardware-software integration and test.

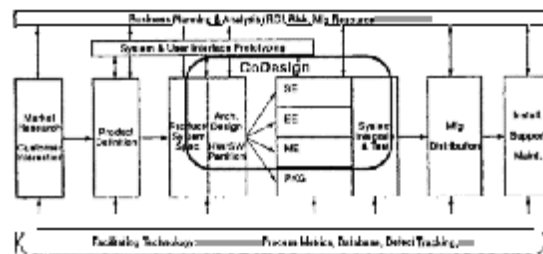


Figure 4, Co-design in Product Life Cycle

Hardware-software codesign, simulation and verification is hardly a new idea. However, until recently, verifying the interaction between hardware and software required the building of actual prototype hardware.

Two trends of the '90s have both exacerbated the problem and opened up new potential solutions. Semiconductor processes now permit entire systems to fit on a single chip. The advent of the possibility of complex systems on a chip including one or more programmable processor cores, and the continuous expansion of digital electronics into virtually all electronic application fields requiring the flexibility of such programmable cores. Thus, embedded systems designed around one or more processor cores probably represent the prototypical design of the future and is the driving force as codesign turns from merely a good idea into an economic necessity. Keep in mind that although the difference between a system on a chip and the conventional design may not be big from a technical point of view the economics change dramatically if errors occur in a prototype.

³ *Heisenbug* is the, somewhat whimsical, name given to a bug whose manifestation disappears when a debugger is used to find it, for example timing related bugs.

⁴ Code-breakpoints can be implemented by software whereas data-breakpoints always need hardware facilities.

The goals behind co-design are:

1. Enable the creation of a system specification that is verifiable via simulation.
2. Enable the design team to explore numerous design alternatives using various hardware versus software partitioning and physical implementations.
3. Cosimulation is used to verify that this final product will be equivalent to the initial specification.

The main problem with co-simulation is simulation speed. Currently VHDL simulators execute a few lines of C code a second, whereas the processor being simulated executes millions of C statements during the same time interval. Speed can be achieved by sophisticated simulation techniques, hardware accelerators or large FPGA-based arrays.

Hardware/software codesign is a very active research field which is currently still emerging and structuring itself. Many different problems with different flavors are being addressed, and it is not totally clear yet which of these topics will be fruitful scientifically and relevant in practice.

Multi CPU Debugging

Currently, some debuggers designed for DSP systems are capable to handle applications that execute on multiple DSPs. Often their functionality is restricted to multiple DSPs of the same family.

The two trends mentioned above, "on-chip emulation" and "systems on a chip", drive the need for debuggers that are capable to support multi core designs. Imagine a chip that contains both a μ controller and a DSP core, both cores are controlled by an on-chip emulator that interfaces to the debugger via a JTAG port. In this scenario, the debugger must be capable to control two applications, that use different instruction sets, via one physical interface. Most debuggers are able to control multiple applications (kernel aware debugging), however hardly any commercial available debuggers are designed to switch between various "processor behavior modules" at run-time.

The Debug Paradigm

Although the degree of hardware and operating system support has strongly influenced the caliber of service a debugger can provide, no fundamental changes have occurred in recent history in the way programs are debugged.

Currently the breakpoint paradigm is implemented in (every) commercially available debugger. This paradigm is adequate for a lot of debugging tasks. In general, bugs whose effects are almost immediately manifested can be easily found with a breakpoint and then mentally looking back along the execution paths.

However, little support is provided for bugs whose effects are not apparent until long after their erroneous action occurs. The (well known) examples are: pointer problems, accessing uninitialized memory, and actions that have an element of indeterminism like interprocess interactions and real-time events. The combination of *checkpointing* and *reverse execution* are the principal tools to find such sporadic bugs. However these features are at the moment a rarely available form of debugging support.

Checkpointing is defined as: taking a snapshot of the complete state at a particular point of an executing program. Reverse execution: the ability to rollback execution to a previous program state, i.e., undo execution by returning to a previous state. The step-back feature available in some debuggers can be considered as a first step towards reverse execution. The main problem with checkpointing is the (huge) amount of data that needs to be saved for later reuse. Algorithms are developed to minimize this data set.