I.Q.
Information Quarterly

The Smart Approach to Designing with the ARM® Architecture

Volume 8, Number 3, 2009

mbed

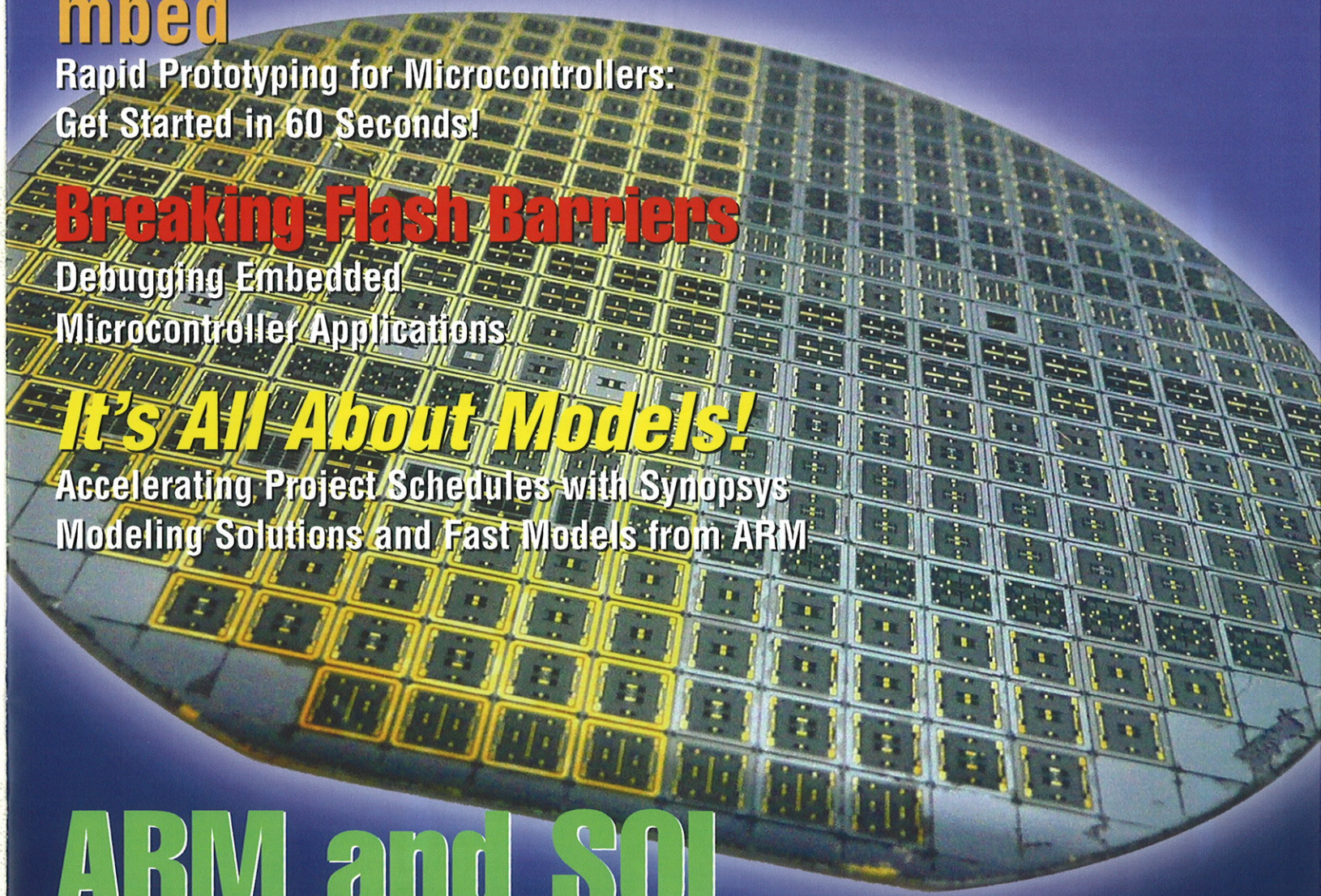
Rapid Prototyping for Microcontrollers: Get Started in 60 Seconds!

Breaking Flash Barriers

Debugging Embedded Microcontroller Applications

It's All About Models!

Accelerating Project Schedules with Synopsys Modeling Solutions and Fast Models from ARM



ARM and SOI

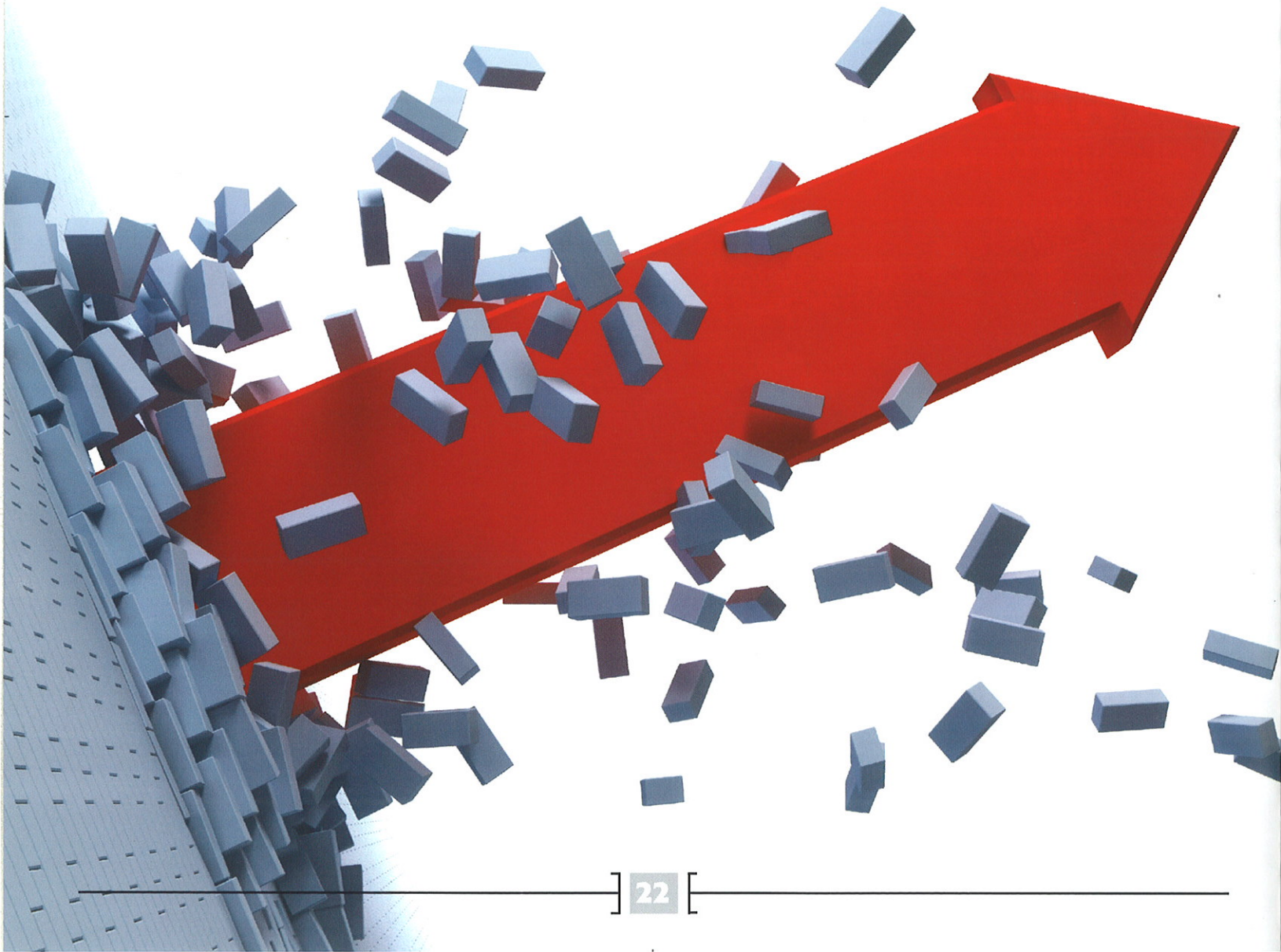
Going Green with a Combination of Low-power Processors, Silicon on Insulator (SOI) Libraries, and Power Management IP

You are debugging an application in flash memory, when the reaction time of the system becomes so slow that debugging becomes impossible. The system has used all the hardware breakpoints and the debugger has switched to low-level stepping. Dirk Akemann looks at how to overcome this and other barriers to debugging flash microcontrollers.

Breaking Flash Barriers:

Debugging Embedded Microcontroller Applications

By Dirk Akemann, SEGGER Microcontroller



Over 50 years ago computer pioneer Maurice Wilkes had a depressing insight. "... the realization came over me with full force that a good part of the remainder of my life was going to be spent finding errors in my own programs."*

Today the number of programmers in the world is measured in the millions and yet they are still spending a good part of their life finding errors in their programs.

Within the embedded environment, while good programming practices and the use of appropriate tools can reduce dramatically the number of bugs that appear in compiled code, there will inevitably be issues that appear only when the code is executed in the target system. To track these down it is common to use a debugger, a software tool that provides insight into the code execution and the state of the software environment.

One of the most valuable tools within the debug environment is the use of breakpoints. The idea is both simple and powerful: when a program encounters a breakpoint it stops execution and hands control to the debugging software. The debugger can then display the status of the application environment (variables, memory, stacks, registers, etc) to help the developer discover how the application has reached this state.

Once the cause of the bug is identified, changes can be made to the code and the debug process restarted.

The host processor communicates with the target microcontroller, and the standard method is through the JTAG interface, originally created for hardware checking (boundary scan) and now used also for software debugging and for loading software into the target microcontroller. With ARM based architectures, the device implementer may also offer SWD (Serial Wire Debug) which uses only two pins compared to JTAG's five and Freescale products usually offer BDM (Background Debug Mode.) Between the host PC and the JTAG, or other, interface the physical connection is usually an emulator.

This typically has a USB connection with the PC and a ribbon cable to the target board, and translates from one environment to the other. (Emulator is a name inherited from a previous technology - no emulation takes place.)

Memory and Debugging

Different memory architectures present different debugging challenges, particularly when writing to memory requires extra effort or is even impossible.

Debugging in RAM is fairly easy, since the debugger uses a simple breakpoint instruction, which is as short as the shortest instruction of the CPU. RAM allows for multiple reads and writes without any noticeable effect.

However, since RAM is normally limited in microcontroller systems, debugging in RAM is not always possible.

For ROM the microcontroller designers provided the hardware breakpoints now commonly used for debugging in flash. Hardware breakpoints simply compare the instruction pointer with the breakpoint position and check whether the instruction is actually called. If this is the case, the program is halted and the debugger is started.

The debugger hands over the management of breakpoints to the J-Link debug probe. For a high-level step the debugger only needs to issue aSetBP()-Go()-ClearBP()-sequence to the J-Link

The hardware breakpoints are normally limited in number and in capability. In ARM7 or ARM9 implementations there are only 2 watchpoints (ARM terminology for a hardware breakpoint.) The new ARM Cortex-M3 has 6 hardware breakpoints available.

How many Breakpoints do I need?

For more detailed debugging many more breakpoints are needed. Even just stepping over an instruction without single stepping requires at least two breakpoints. This is because, as the system hits a breakpoint and stops, the breakpoint is removed and a new breakpoint is set at the next step in the program. The application is restarted and runs until it hits the next breakpoint, when it halts again. If the next step in the program is a branch, then each possible branch requires a breakpoint. So a simple if() needs two breakpoints while a typical switch() needs at least three; one at each case with another at the default: instruction. Yet another breakpoint is needed if the debugger needs to display the terminal output. If a debugger tries to set more breakpoints than the microcontroller can provide, the debugger will normally default to low-level-single step execution, with a dramatic reduction in performance.

**(M. V. Wilkes, Memoirs of a Computer Pioneer, The MIT Press, 1985.)*

Adding breakpoints in the flash memory would seem a logical move to overcome these problems, but flash presents its own challenges.

Writing to flash needs code in RAM, and writing to flash when the application is running can be difficult, particularly when the application uses techniques to improve speed of code execution or power consumption, like changing the CPU speed at run time. Writing a breakpoint to flash, as with any flash write, may change the contents of registers and RAM, destroying the evidence needed for debugging. There are also problems of flash memory speed and endurance (the physical life) when compared with RAM. Unlike writing to RAM, where a write operation can address a single memory location, when a flash write takes place a huge block of memory (which may be up to 64k) has first to be cleared and then written to.

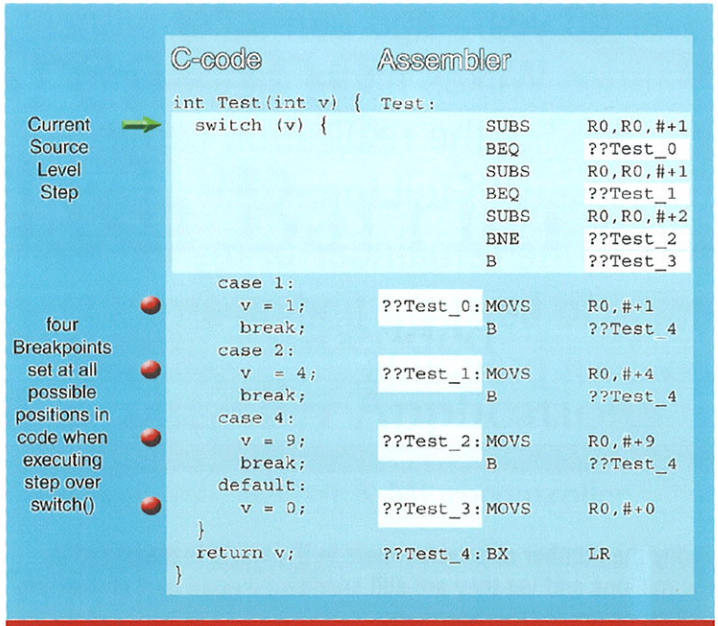
Normally setting even a single breakpoint would require this write cycle. And flash has a finite life: typically after 100,000 writes to a flash memory cell the time it will retain data begins to fall. For a microcontroller memory it is possible that specific cells could reach this number of write cycles unless appropriate measures are taken to avoid it.

Breakpoint in Flash

These are not insuperable obstacles. SEGGER Microcontroller supplies the J-Link JTAG emulator for a wide range of microcontrollers and has developed the optional FlashBP feature, which allows developers using J-Link to use flash breakpoints as easily as RAM breakpoints. It is designed to assist the developer to make the best use of resources available and reduce the number of times flash is programmed. (Reducing flash programming steps both speeds up debugging and extends flash life.) During the debugging process, all the resources of the microcontroller are available to the application program; no memory is lost for debugging.

At the heart of the feature is a RAM code specifically designed for setting flash breakpoints: when using this code on devices with fast flash, setting and executing flash breakpoints is as fast as setting breakpoints in RAM. There are occasions when hardware breakpoints, even with their limitations, can be useful, for example when stepping through code one source-level instruction at a time. When they are appropriate, J-Link will use hardware breakpoints or a mix of hardware and software breakpoints, reducing the number flash write cycles.

There are a number of other performance enhancements used by J-Link. Flash sectors are programmed only when necessary, usually the moment execution of the target program starts. Where possible, J-Link locates several breakpoints in the same flash sector: programming a single sector then sets multiple breakpoints. And debugging does not consume memory, leaving all the resources of the processor available to the application.



This example demonstrates how many breakpoints the debugger needs for a simple switch()-instruction while stepping through the source code.

When single-stepping and the current instruction is already breakpointed, the built-in instruction set simulator reduces the number of flash programming operations. Without instruction set simulation, the debugger needs to clear the breakpoint, step over the current instruction and set the breakpoint back again. The instruction set simulator allows instructions to be simulated or emulated in RAM. Simulation creates the results of the breakpointed instruction.

If a register is set, or a memory area is moved, J-Link simply executes this as if the instruction actually has been executed and increases the program counter accordingly. Some instructions cannot be simulated because their effect is unclear or system/implementation dependent, such as a co-processor instruction. If the instruction allows to be executed at a different place in memory, J-Link will emulate the instruction. The emulation is executed by copying the instruction with its arguments to RAM and executing the instruction there.

Before flash programming takes place, the contents of memory and registers are saved. This protects their contents against changes when programming takes place.

Debugging in the target system will continue to be a significant part of ensuing that a product functions as it should. It will never be an easy task, requiring insight and professional skills from the developer. The different elements of the J-Link toolkit and FlashBP have been designed to allow the developer to select those parts that will make the debugging task as simple and as fast as possible. **END**