The Software Engineer's Guide to In-Circuit Emulation

Increase your debugging skills with Motorola Microcontrollers!

Nohau Corporation
51 East Campbell Avenue
Campbell, California 95008

Version 2.1
October 15, 2002

Introduction

Software simulators, target monitors and BDM (or SDI) interfaces offer economical debugging capabilities sufficient for many applications. In-circuit emulators offer additional advanced real-time debugging facilities. There are differences between these options in terms of debugging power, real-time operation, intrusiveness and productivity effectiveness.

With software debuggers, monitors and BDM tools one can load, single-step and run programs. Software breakpoints can be set and memory can be examined. Some microcontrollers, such as the HC12, have integral hardware breakpoints set via the BDM interface. Code Coverage and Performance Analysis may be available providing statistical information. These tools do a good job, but have some shortcomings that can be crucial in detecting and fixing some of the more elusive bugs or in special circumstances.

Interesting "what-if" Scenarios

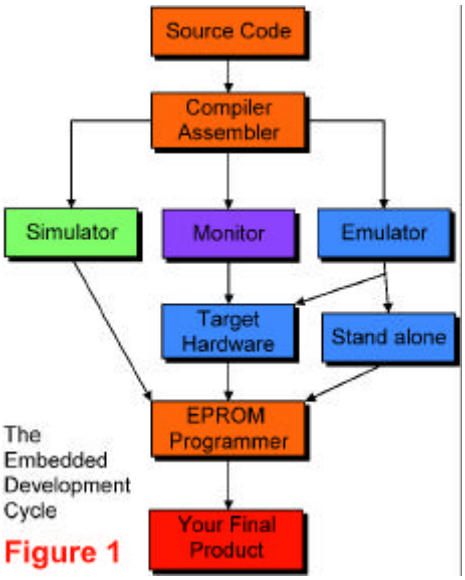
What if you are debugging code that is in ROM? Or even trickier: ROM inside the chip? What if you need the serial port that software debuggers commonly use to communicate with the controller? What if you want to detect a certain situation: such as a write of a certain value to an external peripheral AND if another variable equals 6, and then stop the execution? Or record these bus cycles? What if a bug causes your program to jump off into never-land and you need to know what was happening just before this event? What if your RTOS is causing strange problems? If you have bugs in different memory banks: is your tool aware of bank switching? Unlimited hardware breakpoints, a non-intrusive connection, bank aware conditional triggers and trace memory are solutions to these problems and more. You need an emulator.

This note examines these situations using Motorola microcontrollers such as the HC11, HC12, HC16 and the 68300 series. The focus is on the HC12 series.

The Development Cycle

The typical microprocessor development project begins with a C compiler producing an object file from your source code. This object code will contain the physical addresses and some debugging information using a standard file format such as IEEE695 or ELF-DWARF. This object code can be executed and debugged using a software simulator, a target monitor, BDM debugger or an in-circuit emulator. A most undesirable method is to program an Eprom or internal FLASH, run the target system and observe what happens.

The program is debugged by setting breakpoints to halt execution at selected instruction locations. When execution is halted, the memory and register contents are examined for clues to help find bugs.



The Embedded Development Cycle

Figure 1

The debugged object code is re-compiled. For the final product the debug information is removed and a file is produced in a standard format such as Motorola S-records. This file will be stored in the final product's nonvolatile memory such as EPROM or FLASH. This process is illustrated in Figure 1. This memory can be external or internal to the microcontroller. The HC11 has OTP (unerasable EPROM) and the HC12 has FLASH and EEPROM.

Why do we need emulators?

There are some cases where an emulator is needed to resolve difficult to find bugs. In all cases an emulator will pay for itself by providing you with decreased debugging time, ease of system integration, increased reliability and better testing procedures. Often, designers use both an emulator and a BDM debugger during different project stages, especially in larger design teams. A BDM running on a Motorola controller is more effective than a target monitor.

Software simulators and debuggers offer no means to detect events or conditions and then act on them, and certainly not in real-time. There are no means to record controller bus cycles to determine what actually happened to the program flow.

In-circuit emulators can easily do these tasks and more for you. Emulators are the bridge between software and hardware. At some point in time, you have to run your program in real hardware. An emulator will easily help you accomplish this.

What exactly is an emulator?

"An emulator is a computer that engineers use to design other computers" is the most basic definition I have thought of. Emulators replace the microcontroller in your target system. The emulator behaves exactly like the processor with the added benefit of allowing you to view data and code inside the processor and control the running of the CPU.

Figure 2 shows the Nohau emulators for the Motorola HC12 family. They are all compact handheld emulators that go anywhere you and your laptop can go. They are all powered with a 5 volt supply.

not functioning you might not be able to establish communication with the CPU. Typical problems include wrong or erratic clock speeds, shorted or reversed address and data busses or defective memory addressing logic. You may have difficulty determining why your target is not running and some tools might not provide many clues. Not so with an emulator. An emulator will run with no hardware at all or incomplete sections. You can usually peek and poke at memory areas and gain enough clues to guide you to the problem area such as stuck bit(s).

Evaluation boards are an economical and practical method of shortening your product development time. They are also a useful reference design for comparison.

Connecting to Your Target System

This is easy. Most issues will be handled by the board designer in conjunction with your emulator representative. Connection to the target is a two step process.

First, the adaptation method must be chosen. Solder-down and socket methods are preferred. The HC11 and 16 are often in a socket and adapters are available from Nohau. Clip-over adapters are handy but expensive. They require the target controller to have the ability to be put into a tri-state mode. The HC12 does not have this resource. Therefore solder-down or custom adapters are needed. The BDM needs only the Motorola specified BERG connector. If you need to access the target in a hard-to-access area, consider Nohau's Flex Cable shown in Figure 4. The cable can approach the target from any of the four quadrants. Figure 5 is the Nohau DA/DG128 solder-down adapter.

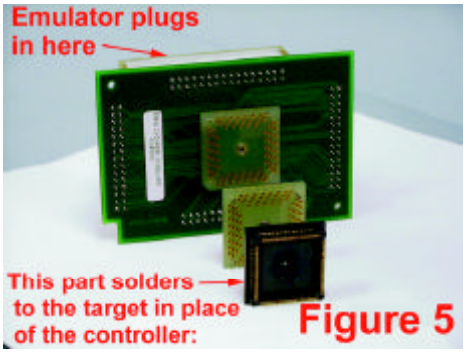


Figure 5

Second, the software and jumper settings on the emulator must be correctly set to match the target board and the software initialization routines. This is easy to do and here is where good technical support counts. Usually the default settings work.

Hardware Breakpoints

A software breakpoint is created by inserting a 2 byte instruction which will divert normal program flow to the debugger. The program may crash if the program counter lands on the second byte. Nohau hardware breakpoints use comparators to detect accesses to a location and no code memory contents are modified. Breaks on regions need hardware breakpoints. Software breaks are still useful and Nohau provides both types.

Software breakpoints are useless with ROM memory since the instruction can not be inserted. Only hardware breakpoints function in ROM systems.

The two hardware breakpoints provided in the HC12 are not sufficient for easy source code single stepping in FLASH memory. Source code single stepping is accomplished by setting hardware breakpoints at all locations the program could jump to when an assembly step is performed. In assembly stepping, there is no problem but with source stepping there are usually many assembly steps associated with a single line of C source code. The problem is where to set the two breakpoints to stop the execution at the end of the sequence and what if there are any jumps out of the sequence? Many workarounds are used especially in BDM debuggers. Some of these workarounds are quite elegant. The Nohau full emulator solves this problem perfectly since it has an unlimited number of hardware breakpoints which are set to cover any contingency.

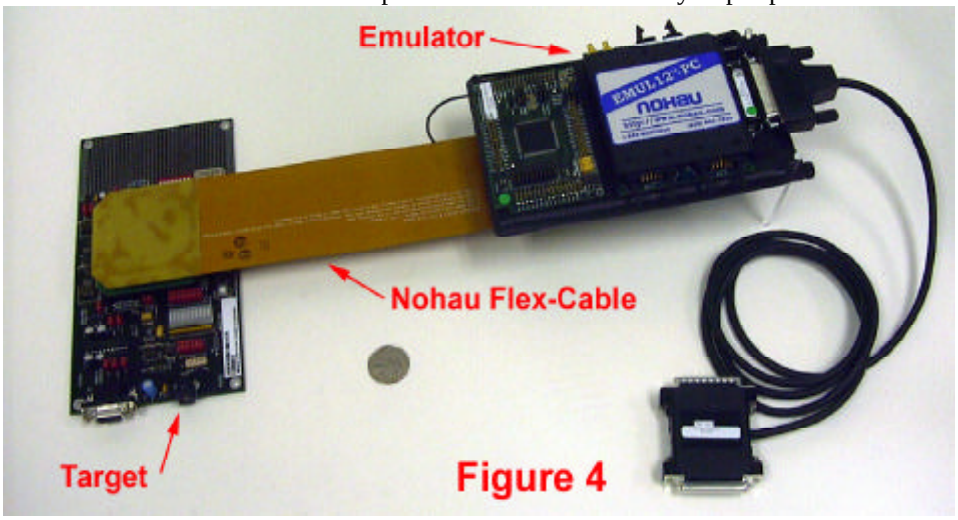


Figure 4

Trace Memory

The Trace records each processor cycle along with a timestamp and optionally external signal levels. The trace can record all code fetches and will distinguish between instructions that are cancelled in the CPU pipeline and those successfully executed. False triggering is therefore avoided on unexecuted instructions.

The trace can be filtered with the triggers so that only specified cycles are recorded. This saves time searching for bugs. You can select what is recorded in the trace and can include data reads & writes, instruction fetches and/or executions, free cycles and power down cycles. You can trigger on specific addresses, data values, and qualifying them as reads or writes and many other similar qualifiers. The trace memory is a powerful tool displaying events as they really happened. Simulators, monitors and BDM debuggers do not have trace memory or triggers.

Trace Memory: An example

The trace window shown in Figure 6 is from the Nohau DG128 emulator. This recording was unfiltered and represents all executed instructions, data reads and writes and a timestamp. Instructions entering the pre-fetch queue, then cancelled, are not erroneously classified as been executed.

The trace can be filtered with the triggers so that only specified cycles are recorded. This saves time searching for bugs. You can trigger on specific addresses, data values, and qualifying them as reads or writes and many other similar qualifiers. The trace memory is a powerful tool, displaying events as they really happened.