The Software Engineer's Guide to In-Circuit Emulation

Increase your debugging skills with Motorola Microcontrollers!

800.686.6428 650.375.0409
www.icetech.com
Nohau

Version 1.2
October 16, 2000

Nohau
422 Peninsula Aveneu
San Mateo CA 94401

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

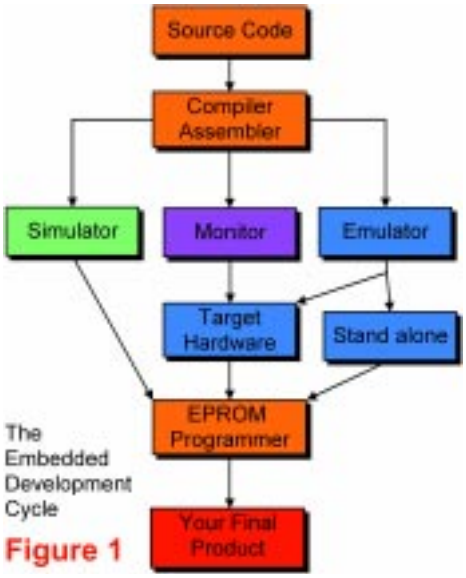
What if you are debugging code that is in ROM? Or even trickier: ROM inside the chip? What if you need the serial port that software debuggers commonly use to communicate with the controller? What if you want to detect a certain situation: such as a write of a certain value to an external peripheral 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.

Figure 2

External Mode

External mode is when the program memory and perhaps some data memory is located externally to the controller. This is the classic situation where the target board contains a microcontroller, some EPROM or FLASH, RAM and some type of peripheral chips. The address and data busses are available to access this memory and are the only means for an emulator to gather information and control the CPU. Therefore information about internal registers is not available in real-time. The address and data busses may not be used as general I/O ports. Production chips are effective for emulating this mode as is the BDM debugger port.

Internal or Single-chip Mode

Internal mode is when program and data memory is located in the controller chip in the form of FLASH or EPROM. This method is becoming preferred for embedded designs due to its low cost and easy software updating. Address and data busses are not accessible by the user or an emulator. These bus pins are then available as I/O ports. All program execution occurs in the internal ROM. This mode requires special considerations for effective emulation. BDM and JTAG debuggers are effective for single-chip mode emulation but have no trace or triggering capabilities.

Nohau emulators generally run production chips in external mode and use special circuitry to reconstruct the I/O ports (Port Replacement units - PRU). This process is transparent to the user. Nohau full emulators do not use normally use the BDM port for emulation purposes.

It is possible to embed a monitor kernel in the ROM, coexisting with the program code. The user can activate the kernel with a switch connected to an I/O pin or a special software sequence sent over some communication link such as radio, Internet or telephone modem. This will allow operation of a debugger via a serial, CAN or other port. Some system resources need to be reserved. This scheme is called remote debuggging and is useful to debug a target when it is not easily accessible such as down an oil well, on a mountain top, embedded deep in a machine or even on Mars! In these types of situations, a remote debugger is common.

BDM: Background Debug Mode

This is an exclusive Motorola feature using a dedicated serial port to gain access to a special debug module inside the microcontroller. This module operates parallel to the microcontroller and generally uses no resources. It has access to internal registers and memory and can control the CPU. Some models have two hardware breakpoints for ROM operation. Figure 3 shows the Nohau BDM debugger. The Motorola SDI appears similar.

Figure 3

A BDM debugger does not have trace memory, triggers or emulation memory. A BDM emulator is sometimes used in conjunction with a full emulator. It is the preferred method used to program FLASH and EEPROM in the HC12 controller in your target system.

The BDM debugger will function on any working target system with an appropriate BDM connector installed. No changes are needed to the hardware so debugging is easy and fast: even to existing products.

You can load a program and single-step or run the target processor. Source code will be visible for HLL debugging. You can set software breakpoints (hardware in HC12) and view memory in real-time.

Variable, arrays, memory areas and structures are viewed in a variety of formats. The Nohau BDM has Shadow Memory allowing the viewing of data writes in real-time. The target emulator usually runs at full clock speed. There are a few instances where CPU cycles are stolen to perform some operations. A BDM debugger is more robust than a monitor, but with less features than a full emulator. Many projects are successfully completed with a BDM or JTAG debugger. A full emulator used in your development cycle will still speed up your development projects with increased code reliability.

JTAG Debugging

The higher speeds and complexity of some microprocessors has encouraged chip makers to incorporate debugging facilities on-chip. The BDM is one example. The JTAG interface was designed to facilitate testing circuit board connections between large chips. This is a serial connection and signals can be sent to and read from I/O ports. This is also called the JTAG Boundary Scan because the serial data scans the outside I/O pins of the chip. A pattern sent from one chip to the next can be then compared to check if the circuit board connections are intact.

Since the JTAG port is unused during normal chip operation, and since it runs in parallel to the controller's CPU, it can access an on-chip debugging module in real-time similar to the BDM interface. This module has direct access to the CPU core. Motorola uses this approach in the PowerPC and some 68K microcomputers. It is possible to build an emulator using a combination of the JTAG or BDM and standard trace and trigger hardware giving high performance but not having all the real time features of a full emulator.

No target or CPU resources used

Monitor kernels typically need about 10K ROM and 10-20 bytes RAM and a free communication port. A good emulator uses none of these. The emulator should be invisible to the target. Better emulators are; and in addition do not steal CPU cycle time for ordinary housekeeping duties such as setting triggers, viewing the trace buffer and modifying memory on the fly.

Getting the Hardware Working

Simulators are great, but they can not take all the variables into account. A simulator designer has to think of everything: the big problems are usually those items that come up after the hardware is constructed. Items like capacitance, timing, inductance, and chip versions. These become more important as CPU speeds increase. It is a very difficult task to replicate the pipeline found in many microcontrollers today and is best done with real hardware.

Target monitors and BDM debuggers are considerably better in that they run on real hardware. But the target system must be a complete working system in order to get the monitor kernel to run. If your target is

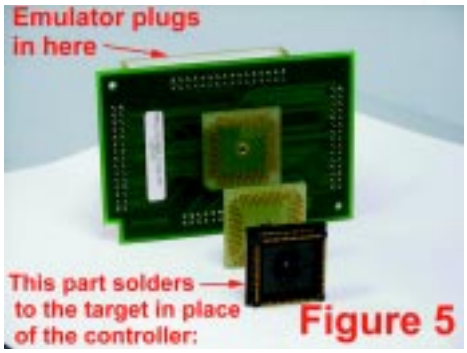
not functioning you might not be able to establish communication with the CPU. Typical problems include wrong or erratic clock speeds, shorted or reversed address and data busses or defective memory addressing logic. You may have difficulty determining why your target is not running and some tools might 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.



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 sincle 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 whaich are set to cover any contingency.

Trace Memory

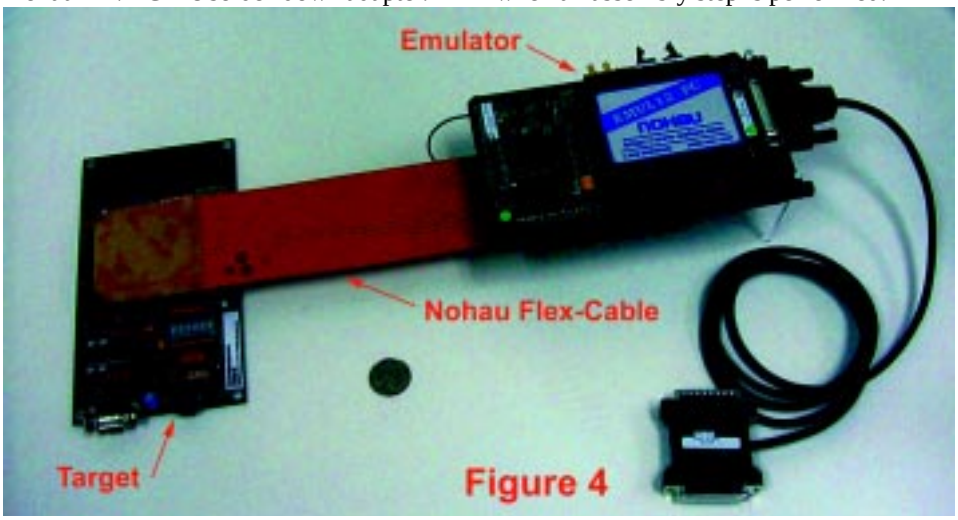
The Trace records each processor cycle along with a timestamp and optionally external signal levels. The trace can record all code fetches and will distinguish between instructions that are cancelled in the CPU pipeline and those successfully executed. False triggering is therefore avoided on unexecuted 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 new STAR12 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.



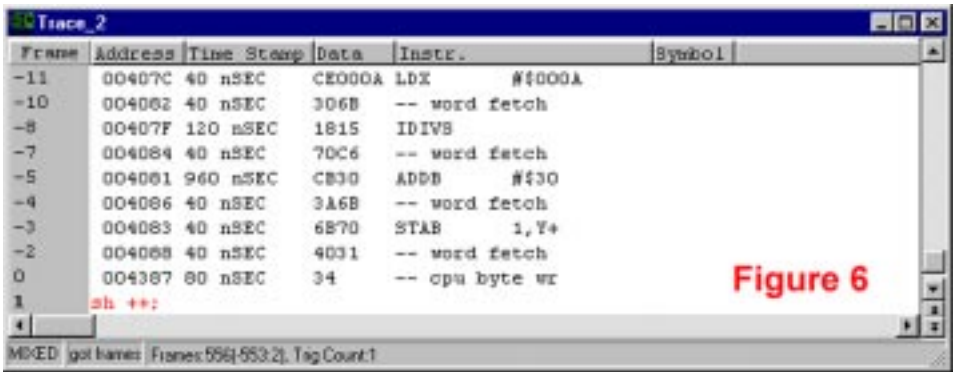


Figure 6

Note the address of the instruction or the data read or write and the corresponding timestamp. The address field has six hex digits. The right four are the value of the program counter or physical data address and the left two digits display the bank page number for the HC12. The timestamp is shown here as relative to each instruction but can be displayed as accumulated time and also in terms of clock cycles.

The red line "sh++;" is the C source code and the associated dissassembled assembly instructions are shown below them. Any related symbol information is shown in the Symbol field but none are shown in this example. There are more fields and items that can be displayed such.

The trace can contain 131,072 lines of data but only 556 are saved here. The trace can be viewed without stopping the emulation process allowing your target to continue running. The triggers greatly magnify the effectiveness and utility of the trace memory which leverages your ability to find and correct bugs quickly and easier.

Conditional Triggers

These are extremely powerful and easy to use. They allow you to specify an action when some event happens. The triggers can be set or edited "on-the-fly" without stealing cycles from the emulation process. The Trigger Configuration window shown in Figure 7 is a simple example. If a data write cycle to the address "show + 7" (4387) with a value of 34 occurs, a trigger event will be created. The trace window in Figure 6 will be created with these settings.

The trigger can include address, data, clock cycles and external signals. These can trigger a break, start/stop the trace capture, record a timestamp or many other things determined by the emulator's capabilities. Note there are three trigger mechanisms plus a filter facility. There are many more features in the Nohau emulator.

The emulation and the trace recording was set to both stop when the qualifier becomes true in this example. The emulation and trace recording are independent operations as discussed previously.

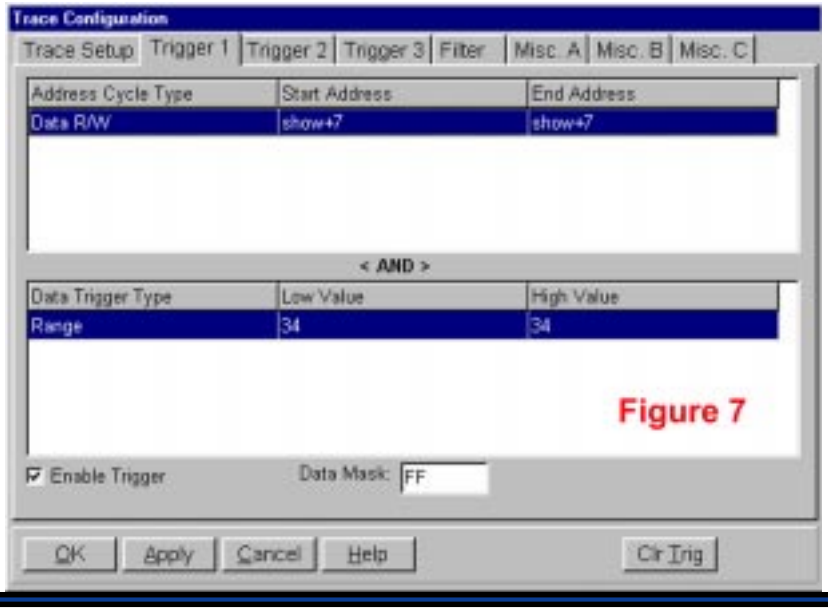


Figure 7

What Figure 6 Really Means:

Refer to the Frame numbers at the right side of Figure 6. Line # 1 represents the skid that happens by definition of a trigger.

Line # 0 represents the trigger point in this example. The byte write of 34 to address 4387 (show+7) is clearly shown. This was the qualifier becoming true and stopping the emulation and the trace recording.

The STAB instruction at Line # -3 is the instruction that caused the write to 4387. This instruction was executed at this time.

The STAB opcode is 6B70. The 6B was fetched at Line # -10 and the 70 was fetched at Line # -7.

We can clearly see when the STAB instruction was fetched, executed and its subsequent data write occurred. We could have also displayed the HC12 free cycles.

The triggers are most useful for detecting writes that you do not expect and that can be causing software problems. The offending instructions and/or data can be displayed providing crucial information.

Emulation Memory and I/O Ports

Ports and memory can be viewed from real hardware parts and not simply a software simulation. It is possible to wire your favorite peripheral chip to the bottom of the emulator pod and access it.

Often, it seems that problems only develop when the program is run on the actual hardware. How often does it seem that things depend so much on the rise or fall time of some input signal? Or the routing of a certain wire? An emulator will help get your development finished faster by getting you to this point directly.

Since the emulator has its own internal RAM which can be substituted for ROM, you can debug and modify the program code and data easily in ROM systems.

In the same fashion, memory not yet installed on the target can be substituted by the emulator. The size and address of this RAM is selectable. The granularity is 1 byte allowing mapping around any external peripheral in your target memory.

Conclusion

This article has provided information about In-Circuit Emulators and the benefits that accrue to you, the designer. See www.nohau.com for more information on Nohau emulators supporting Motorola.