The ZEN of BDM

Craig A. Haller
Macraigor Systems Inc.

Introduction

You may wonder, why *The ZEN of BDM*? Easy, BDM (Background Debug Mode) is different from other types of debugging in both implementation and in approach. Once you have a full understanding of how this type of debugging works, the *spirit* behind it if you will, you can make the most of it.

Before we go any further, a note on terminology. “BDM” is Motorola’s term for a method of debugging. It also refers to a hardware port on their microcontroller chips, the “BDM port”. Other chips and other manufacturers use a JTAG port (IBM), a OnCE port (Motorola), an MPSD port (Texas Instruments), etc. (more on these later). The type of debugging we will be discussing is sometimes known as “BDM debugging” even though it may use a JTAG port! For clarity, I will refer to it as “on-chip debugging” or OCD. This will include all the various methods of using resources on the chip that are put there to enable complete software debug and aid in hardware debug. This includes processors from IBM, TI, Analog Devices, Motorola, and others.

This paper is an overview of OCD debugging, what it is, and how to use it most effectively. A certain familiarity with debugging is assumed, but novice through expert in microprocessor/microcontroller design and debug will gain much from its reading.

Throughout this paper I will try to be as specific as possible when it relates to how different chips implement this type of debugging. The infomation comes from various sources. Motorola freely publishes all the information needed to use and implement their on-chip debugging features. Texas Instruments charges several thousand dollars for the information and it is included in their emulator porting package. IBM releases most of the information under Non-Disclosure Agreement. Analog Devices will not divulge the information with the belief that their debugger is all one needs (no comment from the author on this).

This paper consists of two sections.

SECTION I - The Past - a review for those new to microcontroller debugging

"Debugging through history" is a quick review of the debugging methods that have been popular with embedded system designs.

"Limitations, etc." discusses the limitations of the aforementioned methodologies.

SECTION II - On-Chip Debugging

"What is OCD - Hardware, Software?" describes various OCD implementations and exactly what OCD is.

"Types of OCD" goes into specifics for various processors on the market.

"Designing your Prototype" gives hints and advice on how to get the most out of OCD by properly designing your prototype system.

"Designing your Product" gives hints and advice on how to use OCD in your final product.

"Choosing a Debugger" discusses the features to look for in a debugger to get the most out of the OCD features.

About the author ...

The author is the president of Macraigor Systems, Inc., an OEM of embedded systems debug tools, and a firm believer that the silicon manufacturers are not marketing the advantages of OCD nearly enough! This paper is intended to be purely infomational and not a marketing tool, although you should feel free to contact the author about OCD products. Contact information is at the end of the paper.

Note: BDM, OnCE, and ColdFire are trademarks of Motorola Semiconductor. RISCWatch, RISCTrace, PowerPC, and IBM are trademarks of IBM Corporation. SHARC is probably a trademark of Analog Devices. Other terms which are trademarks are the property of their respective owners.

SECTION I

This section is not for old-timers, experienced debuggers, ICE gurus, and the like. It is purely a light hearted look at the state of the art of microprocessor and microcontroller debugging through the years. The background is very helpful in understanding on-chip debugging. Skip to SECTION II for the specific information about on-chip debugging.

Debugging through history

First there was the "crash and burn" method of debugging. We have all done this more times then we admit. You decide to make a paper airplane. You fold it to the best of your ability, toss it and it glides like a set of car keys. Do you A: take a slow motion video of its flight, study it and modify the design, B: rent time at the wind tunnel at MIT, C: crumble it up, toss it, and try again, or D: claim that its sudden nose dive is a feature?

You write your code, check it over once or twice, burn the EPROM and let it run. If it doesn't work, you look at the code some more. At some point this is all we have tools for. It is not necessarily a bad method but it can be extremely slow.

There are several ways to improve the crash and burn method, an "informed" crash and burn, if you will. Many times we will sprinkle output messages at various parts of our code. If we are lucky enough to have a monitor or LCD output device, different tasks or routines can "announce" their execution by displaying a message. Short of this, a single LED may blink or turn on at certain points of the code. If the code "goes in the weeds", "crashes", etc., we get an idea of where the problem lies by the last proper output.

After some time, the idea of a hardware single step was implemented. This let you cause the processor to execute one instruction and then stop. At that point, various signals could be checked, etc. and you could determine what code was executing. The Intel 8041 was an early microcontroller that allowed single stepping via a small, external hardware circuit.

At some point in history, someone (and after reading this I will get lots of email claiming credit) had the idea of a debugger monitor (aka ROM monitor). This is a small piece of code (that's a relative statement) to help with debugging. It usually communicates via a serial interface to a host computer or some form of terminal. A basic monitor allows for the download of code, reading and writing of memory and registers, and perhaps most importantly setting breakpoints, single stepping, and real-time execution. More complex monitors may allow source code profiling, complex breakpoints, and more.

Soon after this, a new device was introduced. The ROM emulator is a plug-in replacement for the ROM chips on the target. The device is connected to the host computer via a serial, parallel, or more recently, an ethernet link. In its simpliest form, the ROM emulator allows you to quickly download code (as opposed to programming the ROM in a separate programmer) and "crash and burn." In its most sophisticated form, you can download code that contains a ROM monitor and communicate with the monitor via the emulator's connector.

Similiar in concept to the ROM emulator, the next major breakthrough in debugging was the user friendly in-circuit emulator (ICE). This device allowed full access to the programmer's model of the processor, hardware breakpoints, execution trace, and much more. The ICE typically used a special version of the processor called a "bond-out". This form of the chip has many more leads on it, bringing typically internal signals to the outside to be monitored and manipulated. System, or external, memory is also supplied

and may be mapped into the user space. This allows for debugging code before a target system exists! Typically an ICE is a complex piece of hardware and software and is considerably more expensive than a monitor based debugger.

The latest addition to the debugger arsenal is on-chip debugging (OCD). Early on-chip debuggers were basically debug monitors written into the microcode of the target processor (Motorola CPU32 family of processors). More advanced systems added other features such as real-time reading of the program counter (Analog Devices' SHARC processor) and near real-time reading of memory locations (Motorola's ColdFire). The basic OCD allows for code download, reading and writing memory and processor resources, single stepping, processor reset, and status (running or halted). Typically, on-chip peripherals may be set to shut down during OCD (as opposed to while the chip is executing user code).

Some processors enhance their OCD with other resources truly creating complete on-chip debuggers. IBMs 4xx PowerPC family of embedded processors have a seven wire interface ("RISCTrace") in addition to the OCD ("RISCWatch") that allow for a complete trace of the processor's execution. Simply capturing these lines in real-time, a debugger can then display a full trace of the last *x* instructions executed.

Limitations, etc.

The limitations to crash and burn debugging come down to time and money. It is ultimately more expensive to spend a great deal of time tracking down each bug. Debugging this way is similar to trying to fix a mechanical device in a dark room. Just turn on the light, use a debugger of some type.

ROM monitors are actually very powerful in their debugging abilities. One of the first drawbacks is a chicken and egg type of problem. If you are writing the monitor for a new processor, how do you debug it? Typically, you crash and burn to get basic communication functions working. Then you add debug statements (fprints in C) and do a modified or "informed" crash and burn to get the debug monitor working. If you are good, and you are porting a known monitor, this should not take too much time (relatively speaking).

Another drawback to ROM monitors is that the target processor is always running. Why is this a problem? Many processors (especially smaller microcontrollers) have peripherals or features that never stop. For instance, the timer on Motorola's 68HC05 runs continually, even when you enter the monitor. Thus, any debugging of timer interrupts is difficult if not impossible. This includes real-time schedulers, a common piece of software.

ROM monitors are also just that, monitors in ROM. Thus, you must have some ROM for the monitor. Some processors let you get around this if they have a type of auto-bootload that will load code into RAM on start-up (Motorola 68HC11), usually via a serial port. In general, the monitor must be resident and takes anywhere from 256 bytes of code to 30K and up.

A typical ROM monitor will only debug code that resides in RAM. This means that your target must have enough RAM to hold the code and have it in the same (or equivalent) memory map as the final product's ROM. The main reason for needing the code in RAM is for controlling the program's execution. Breakpoints are typically implemented in software (inserting an opcode for a "trap") and thus must be in RAM. Single stepping is often done by inserting breakpoints in appropriate places.

ROM monitors use processor resources or require additional hardware. The monitor must

communicate with an external debugger (or, minimally, a “dumb” terminal) and this requires a UART. It may be an on-chip UART (thus using a resource) or an external UART requiring special hardware just for debug, eating up valuable board real estate, etc.

Finally, since the ROM monitor runs on the target processor, upon reset it typically does some housekeeping on the chip. This may include initializing chip selects, setting the stack pointer, clearing memory, etc. If the target code does not recreate this initialization, the environment that the target code runs in will be different than that environment during debug. This is assuming that the final product does not go to the ROM monitor on reset. This is a vitally important point. Again, *the final code may not be running in the same environment as the tested code.*

ROM emulators main limitation is that they are limited to systems that contain external ROM. This eliminates many microcontrollers that run in single-chip mode. Additionally, since they basically implement a ROM monitor, all the drawbacks just mentioned may apply.

In-circuit emulators also have some drawbacks. The vast majority have some type of interface to the outside world (your target circuit) that is not 100% identical to the interface on the actual chip (contrary to the advertising for the emulator). For instance, the oscillator circuit is usually in the emulator, not your target. Additionally, many emulators must re-create parallel ports on microcontrollers since the emulator uses them as address and data lines for its own use. There are exceptions. Philips Semiconductors' XA emulator chip uses a totally separate bond-out section for emulation and all user pins are available without buffering or recreation. The factory debugger/emulator is actually as pure an emulator as you can get, i.e.: no signal modification or additional loading on the user side of the chip. Do be aware that the author of this manual is the designer of that board and may be prejudiced.

Additionally, an in-circuit emulator is not always available. Many of the newer microcontrollers are actually designed with a single customer in mind. That customer may buy hundreds of thousands of chips but only need three or four debug stations. It is not worth the cost of developing an ICE for three sales! One solution is from Hewlett Packard with their “Distributed Emulation”, basically an OCD debugger and a sophisticated logic analyzer working with a single user interface to essentially create a custom ICE.

In-circuit emulators are generally expensive. They typically include high-speed memory (read: expensive), ASICs (read: expensive), and specialized cables (read: expensive).

On the positive side, an ICE will typically give you real-time trace, not use processor resources, and allow debugging without target hardware.

As for OCD, it has no drawbacks.

Seriously, it does have a few drawbacks. The target usually needs RAM instead of ROM for debugging, but this is not always necessary. There typically, but not always, is no form of real-time trace.

SECTION II - On-Chip Debugging

What is OCD? - Hardware, Software, ...

In the general sense, on-chip debugging is a combination of hardware and software, both on and off the chip. The part that resides on the chip is implemented in various ways. It may be a microcode based monitor (Motorola CPU32) or hardware implemented resources (IBM PPC403). There may be resources used that are available to the end-user's code such as breakpoint registers (most embedded PowerPC implementations) or dedicated hardware purely used by the OCD such as instruction stuff buffers (also in embedded PowerPC implementations).

On-chip debugging does require some external hardware, however minimal it may be. There must communication between the chip and debugger host. In most cases this is via a dual-row pin header and several pins on the processor. The IBM 403 family uses the JTAG port pins, in addition to RESET, power sense and ground, and connects via a 16 pin dual-row header. Motorola BDM typically uses five dedicated pins (sometimes multiplexed with real-time execution functions), power, ground, and at least one reset, all terminating in a 10 pin dual-row header. Many of the DSP chips use a Texas Instruments style standard JTAG interface. Motorola has expanded the interface's internal definition to include its DSP BDM equivalent, OnCE.

On-chip resources are only half the story. A target system with an OCD processor and its dual-row header are useless unless you have a host to communicate with. The host runs your debugger software and interfaces to the OCD header in various ways. The debugger on the host implements the user interface displaying your code, processor resources, target memory, etc. The hardware interface may be one of many types. The simplest is typically a “wiggler”, a device that interfaces the parallel port of an IBM type PC to an OCD header (Software Development Systems BDM, Motorola ICD cable). This is both simple and slow. Other interfaces are serial port (RS-232) to OCD converters (Cygnus), high speed parallel port to OCD (Macraigor Systems), ethernet to OCD (Cygnus, Macraigor Systems), ISA bus card to OCD (Nohau), and others. Cost of host software runs from $49 to several thousand dollars. The hardware typically costs from $100 to $3000 (wiggler vs. ethernet interface).

Types of OCD

BDM (Motorola CPU16, CPU32, ColdFire)

As mentioned previously, Motorola coined the term BDM (Background Mode Debug) with its CPU32 family of microcontrollers. This was followed by the CPU16 family, and then ColdFire. These BDMs are extremely similar. They build upon the concept of a ROM monitor and have a similar command set. The core of the hardware interface consists of a serial data in, serial data out, serial clock/breakpoint, and freeze status signal. The commands are shifted into the chip serially and are 17 bits in length. The command set for the CPU32 is as follows:

RAREG/RDREG	read address or data register
WAREG/WDREG	write address or data register
RSREG	read system control register
WSREG	write system control register
READ	read memory
WRITE	write memory
DUMP	read memory block
FILL	write memory block
GO	run CPU
CALL	call user patch code
RST	CPU reset instruction
NOP	null command

These commands closely mirror those that have been used for years in ROM monitors. Single stepping is accomplished via hardware control of the BDM port or by placing a software breakpoint type of instruction in the code stream.

The processor is not aware of the BDM engine, it is not seen as an exception or interrupt. There is a “background” instruction, “BGND” which causes the processor to enter BDM when it is executed. BDM is left, and real-time code execution is resumed, upon the GO command being executed.

Embedded PowerPC BDM (Motorola MPC5xx, MPC8xx)

This BDM works quite differently from the CPU32 type of BDM. The hardware interface is similar with serial in, serial out, clock, reset, and status signals. The difference is that there is not a specific command set. Any serial stream entered into the chip is either 7 or 32 bits in length (not counting start, control, and length bits). Thirty-two-bit bit streams go into the instruction stuff register and come out of the debug data register. What actually happens is that the host debugger stuffs PowerPC opcodes into the processor and they are executed. This is actually a very powerful design allowing for all system resources to be accessible since this method gives the debug port the same power as executing system code. Seven bit data streams are used to control on chip breakpoint functions. Debug control registers exist to enable single stepping and other special controls.

The processor is “aware” of this BDM in that it is a CPU exception. BDM may be entered upon one of any number of exception causing events (invalid opcode, address bus misalignment, non-maskable interrupt, etc.) To resume real-time execution, the debugger stuffs a “return from exception” instruction, “RFI” into the processor’s instruction register.

OnCE (On-Chip Emulation)

The OnCE (On-Chip Emulation) interface is found on Motorola’s family of DSP chips. It allows for all the same type of debugging as the BDM interface. On most of the chips, the

OnCE interface is implemented via dedicated pins. On the more recent parts, the OnCE engine is accessed via the JTAG port pins. The OnCE port is more complex than the BDM port in that it is a state machine controlled by the external debugger. The capabilities of OnCE include:

Interrupt/break into debug mode on program memory address
Interrupt/break into debug node on data memory address
Interrupt/break into debug mode on an on-chip peripheral access
Enter debug mode using a DSP microprocessor instruction
Read/write any DSP core register
Read/write peripheral memory mapped registers
Read/write program or data memory
Step one or more instructions
Trace one or more instructions
Save or restore current chip pipeline
Read real-time instruction trace buffer
Exit debug mode

JTAG debugging (PPC6xx, IBM 4xx, TI C90, Analog Devices SHARC)

JTAG (Joint Test Action Group. pronounced “jay-tag”) is an IEEE specification (IEEE 1149.1). It is actually a method for doing full chip testing and was originally implemented to allow testing of all the pin connections of a chip and its interconnections to other chips on the circuit board. It is a serial protocol and chips on the board may be daisy-chained together. In simple terms, the JTAG serial chain through the chip may be wired through any on chip devices but typically minimally connects to all the I/O pins and buffers. The chain may be several score long or thousands of elements. There is no specification stating any inclusion of resources for software debug nor is there a prohibition.

Different processors implement OCD via JTAG in different ways. The 600 series of PowerPC microprocessors purely use the hardware test chain which winds its way through many of the on-chip resources. Somewhere in the multi-thousand stage serial chain is the instruction register, for example. Debugging with this system is tedious since each core OCD action (modify memory location for instance) may take many trips through the entire JTAG chain. Although the debugger may only be interested in a 32 bit piece of the chain, all elements must be traversed, and multiple times. Downloading user code may be as slow as less than one hundred bytes per second (vs. over 20K per second with other methods). Another drawback to implementations that use a shared hardware test/software debug chain (TI DSP chips, 600 family PowerPC, etc.) is the way the chain is routed during chip design. Since this is typically the least critical path and the least critical part of the chip design/layout (as well it should be), the designers let the silicon auto-router layout the chain's pathway after the rest of the chip has been laid out. This means that each revision of the silicon may have a different JTAG chain, hence the host debugger software must be aware of every revision of silicon. This is a nightmare. TI solves this problem by often updating their OEM emulator software tool kit. This does not help the end-user unless he/she has a very reliable debugger vendor.

An alternative method to the JTAG OCD is to use a different chain via the JTAG port. This is allowed for in the IEEE specification. Using this method, one chain is available for the hardware test and debug of the chip, another for software debug. This method is used in the IBM 400 series of PowerPC as well as in the SHARC DSP from Analog Devices. This secondary chain allows access to debug specific registers, usually only two or three are needed. In the IBM chips, the debug port has access to an instruction stuff buffer, a debug control register and a debug status register. The instruction stuff buffer allows the

debugger to stuff any opcode into the core processors' instruction register, in effect causing a single step to occur. By executing the proper instructions, any action needed may be performed. The debug control and status registers allow for the typical debug commands such as single step and run. Since a chain separate from the hardware test chain is used, the length of the chain is typically under 50 bits long. There is some small overhead with each JTAG action to ensure that the proper chain is being accessed.

Note that TI uses different flavors of the JTAG port on the DSP chips. The C30 family actually has what is referred to as an MPSD port, similar but not exactly JTAG.

An advantage to using the JTAG port for software debug is that it does not need any additional pins on the processor for separate hardware and software debug. A disadvantage is the added overhead needed for each basic action.

Designing Your Prototype

There are many things to consider in designing your prototype target to take advantage of the OCD capabilities.

1: Use it!
This may sound simplistic but some designers are so accustomed to ROM monitors or emulators that the decision is made not to use the on-chip debugging features. If you are not going to use the features, maybe another member of the design or integration team will. It takes very little to leave the option open to others. Minimally …

2: Place the specified header on your board, if possible.
If the prototype is not the same PC card as the end product then there is sure to be room on the board for the header. If there is not room, there are other options. Take a look at the header specification from the manufacturer and the debugger you plan on using. You will probably find signals that you do not need. The RISCWatch header from IBM contains several "no connect" signals and a "key" pin (rather, missing pin). If you must, you can substitute a smaller header and make a conversion cable. The Motorola BDM for the CPU16/CPU32 family contains two ground signals and a DATA STROBE signal. Typically, one ground may suffice and most debuggers do not use the data strobe. The header does not have to be a dual-row header on .1 centers. Although this is the specification, feel free to modify it to fit your needs. Be somewhat careful about the layout, it is helpful to keep higher frequency lines separated. If you are using a wiggler, this is not a problem since the frequency does not usually surpass 100K bps.

3: Watch those traces!
It is best to keep the OCD connector close to the CPU since the lines are typically not buffered. It is important to keep the traces approximately the same length, especially the serial communications lines. For Motorola, these are the DSI, DSO, and DSCK lines. For JTAG interfaces, the TMS, TDO, TDI, and TCK lines. I have seen problems, even with low speed wigglers, when the lines meander around the board from the processor to the header, particularly with 3.3 volt parts. Remember that some JTAG ports are used for both hardware and software testing. The hardware use may necessitate connecting many chips on the board together via a JTAG daisy chain. This will greatly effect software testing and noise on the chain. Think this out carefully and watch your design.

4: Watch those resistors!
Motorola chips, in particular, set up the OCD configuration and access during the hardware reset. It is vitally important that the debugger correctly control these lines during this time. Equally important is what happens when you test the board without a debugger attached. The manufacturer will also offer a recommended circuit for the OCD/JTAG header which may or may not include resistor pull-ups and/or pull-downs. Additionally, other signals on the header may have recommended circuits (i.e.: IBM suggests a 1K series resistor on the power sense line for the 4xx processors).

5: Connect the boot chip select to RAM.
Virtually all of the on-chip debugging equipped processors have multiple chip selects. Typically, one of them is configured during a hardware reset to be used with whatever boot ROM or start-up code ROM is in the system. Many newer designs use a FLASH EEPROM for this purpose. During debug, it is obviously advantageous to use RAM instead of ROM to hold the code under test. One possibility is to have another chip select pointing to a bank of RAM (that may or may not be in the final product). The problem here is that when you or the debugger cause a hard reset, the chip select for the RAM is NOT appropriately

configured. This must somehow be done (see section on why most OCD debuggers do not have ZEN). Additionally, you will be testing code running with a chip select different from that in the final product, something better to avoid.

I recently solved the problem this way … My boot chip select and general chip select zero went to a 2x3 pin header. On the board was a 128K FLASH EPROM and a 128K static RAM. By changing the jumpers on the header, either device could be controlled by either chip select. During initial debug, the RAM received the boot chip select. After code was burned into the FLASH (in target, via an OCD Flash programmer), the jumpers were changed and final debug and test was conducted. You will also find that if you socket your boot ROM, there may be a RAM with the same footprint that will fit in the socket during debug, or a simple socket adapter may be fabricated (thank goodness for technicians!). Don't forget to make the socket writable. This is the ideal solution.

Another common practice in a final product is to have the application code in as slow, small, and narrow a boot ROM as possible. This allows inexpensive storage. The actual boot code then sets up the hardware (minimally, the other chip selects) and copies the application code from the slower ROM to faster system RAM. This is followed by a jump to the start of the application. This kind of simple boot program is easy to implement. You may want to have the boot section of the code written and placed into a ROM on the boot chip select line. During debug, the application code would not be in the ROM but still on the host. When you reset the target under test, you would then execute the beginning of the boot code to set up the hardware and then return to the OCD. Now your application under test may be downloaded to the RAM on the chip select with which it will run in the final system.

6: Set up FLASH ROM for writability.
There are tools on the market that allow for programming of FLASH EEPROM while it is on the target board. These tools work through the on-chip debugger. By configuring the FLASH in such a way that the processor can write to it, the work of programming it is much easier. This might entail simply running a WRITE line to the chip (5 volt only FLASH) and/or the addition of 12 volts controlled by a port pin (preferred) or a jumper. Typically this means adding a trace or two to the PC board. Definitely worth the added copper. (The author sells such a program!)

Designing Your Product

Many of the same issues apply here as in designing your prototype.

1: Use it!
This is not as obvious as it is with the prototype. There are many reasons to have access to the OCD in a final product. With the proper host support, FLASH EEPROM programming is possible, production line testing, in-field debug and much more. Even if you don't intend to use it after production starts, the lack of access to OCD, if it is ultimately needed, may be very costly.

2: Place the specified header on your board, if possible.
It is not nearly as important as with the prototype to use the factory specified header. See the hints in the prototype section for using a different header if necessary.

3: Watch those traces!
This is as important, if not more, than with the prototype. Your product may be in a less friendly environment (electrical noise, etc.) and you may not be able to control the environment as easily.

4: Watch those resistors!
As important as with the prototype if not more so. You want to ensure that all start up parameters are correct, this is especially important with Motorola's BDM interfaces.

5: Set up FLASH EEPROM for writability.
This is extremely important. The ability to easily program FLASH, both on the production line and in the field, will prove invaluable. Additionally, this allows you to eliminate the use of sockets for the EEPROMs. See the prototype section for hints on implementation.

Choosing a Debugger

Some general thoughts and information on debuggers, then specifics about OCD debuggers.

First, the “invasiveness” of debuggers. By this I refer to the amount of system setup that the debugger does for the user. A ROM monitor typically must do some setup, such as setting up the stack, initializing chip selects, etc. An OCD debugger does not have to do this but often does. Why does this matter? If the debugger does ANY setup work and your code does not reproduce this setup in the exact way (and possibly at the exact time) your code will not be running in the same environment as when it is tested. This is a perfect example of why your code will work with the debugger but not directly out of ROM. An OCD debugger may or may not have to do setup, it actually depends on your hardware configuration as we will see. Other similar invasions are the initialization of general registers, setup of an oscillator pll, etc.

Second, there are also different thoughts on how much the debugger should protect you, the user. A common target has a bank of RAM into which your code is loaded for testing. Assume your code is running in real-time and it “goes into the weeds” (not *your* code!), in other words there may be an errant pointer and you are now executing out of uninitialized RAM, garbage code. You may not know this and the code may go for a while, reeking all sorts of havoc, until, for some reason the debugger regains control. This is a tough one to debug unless you have a large trace buffer. Alternatively, the debugger could have filled memory with some specific instruction before downloading your code. If this is a BREAK, BGND, TRAP, or INTERRUPT instruction that the debugger would recognize, a break would have occurred at the first errant instruction. Should the debugger do this automatically? What about interrupt vector tables? Should the debugger fill in all uninitialized vectors and trap on their use?

Some of these issues are easier to deal with than others, depending on the target chip. The embedded PowerPC chips have many options for protecting the user. By setting bits in a register you can cause the on-chip debugging mode to be entered (hence, a breakpoint) for various events such as execution of an unrecognized opcode, misaligned data fetch, etc. If the debugger secretly sets these bits, you are debugging in a different environment than that in which your code will run. This is probably OK, but is it? Does your debugger give you access to these bits?

OCD specifics

There are a handful of OCD debuggers available on the market. They range in price from freeware to several thousand dollars. All of them give you the basics: read and write registers, read and write memory, download code, single step, run, etc. Most give you source level debug capabilities. Some work only with assembly code, others with any language. The differences of most concern are how the aforementioned situations are handled. Is there “hidden” initialization? Are there “user friendly” traps? And what about that start-up stuff?

Just like getting out of bed in the morning is tough for many of us, some processors need a helping hand coming out of reset. Let us assume that my suggestions for designing your prototype were ignored. Your target has its boot chip select attached to a ROM chip of some type. Since this is debug time, there is no code in the ROM as of yet. You have a debugger connected to the OCD header and want to start testing your code. You have the debugger reset the target and then download your code. WRONG! Upon reset, the only properly setup chip select line will be the boot chip select. This is pointing to useless ROM. Whatever chip select is attached to your RAM must be initialized. But by who (or what)?

Some debuggers have built in setups for known hardware such as various manufacturer’s development boards. Usually you can describe your custom target via dialogs to tell the debugger how to setup the board. Other debuggers allow you to write command files (“macros”, “scripts”, etc.) to do the setup. These files have commands such as WRITEL 0x1234, 0x5678 which would write a LONG value of hexadecimal 1234 to location hexadecimal 5678. With some debuggers you must explicitly run the command file every time you reset the processor, others do this automatically. Again, the main problem with doing this is that your code is now in an environment that is different from the reset environment, and your code did not cause this change. If the only command is a setup of the RAM chip select, this is probably not too big a problem. Probably.

Another setup issue is the speed of the target processor. Many of the newer processors use an inexpensive 32 kilohertz crystal along with an on-chip phase lock loop (pll) to boost the system frequency. Upon reset, the pll is at some default value, quite possibly a very slow one. Often, your application’s initialization code will set the pll to some faster value, but during debug this only happens after your code is downloaded. If the debugger does not do any type of setup (hidden or not) and you do a download (via the boot chip select), the processor is most likely running at a slow speed. This will slow down your download which is always too long, no matter what the speed! This is another reason that you must thoroughly think about how the setup will work.

Talking about speed, one more issue that has not been discussed is raw OCD speed. All of the OCD protocols are implemented serially. The limit, or maximum OCD speed, is usually a function of the CPU clock speed. Typically, the OCD may only run one-third or one-half of the CPU speed. Most OCD hardware interfaces start at a slow speed since the processor speed usually cannot be determined. If the speed of the interface is not set for the maximum speed (either the fastest the CPU will handle or the fastest the interface can run, which ever is slower) the speed of debugging is affected. This is most obvious in the download speed of code. Some debuggers will allow you to modify the interface speed in a command file. You would do this only after you set any pll speed, of course. Additionally, you will probably have to set the interface speed to be slow at the start of the command file. Why? Once you RESET the target processor, it is running at its default speed. If this is slow, you must slow the interface to do your pll setup, then speed up the interface. And you thought this was going to be easy. Ideally, you may have a macro that runs whenever you hit the

debugger RESET TARGET button or command on your debugger that will:

1. Reset the target CPU
2. Lower the OCD speed
3. Set the processor PLL for desired speed
4. Raise the OCD speed to as fast as the processor will allow

So how do you choose a debugger?

Bascially, use the information in this paper, ask questions of the vendor, and see the debugger in use. Does it work with your favorite compiler? How does it communicate with the target? What is the “invasiveness” and are those items fully documented?

I am prejudiced about debuggers. I have written and marketed several from basic DOS assembly language based debuggers to complete Windows based high level systems. For these reasons, I will leave you to your own devices. Feel free to contact me for information about debuggers and what I have to offer.

Good luck and good debugging.

Craig Haller
President
Macraigor Systems Inc.
P.O. Box 1008
Brookline Village, MA 02147

(617) 739-8693
(617) 739-8694 - fax
www.macraigor.com
craig@macraigor.com

