

## Course Introduction

**Purpose:**

- The intent of this course is to explain the operation and use of the background debug module (BDM) and on-chip in-circuit emulation (ICE) systems, which are used for FLASH programming and application debugging.

**Objectives:**

- Describe the operation of the single-wire BDM.
- Describe how the BDM can be used to read or modify on-chip memory or CPU registers, set breakpoints, and single-step through an application program.
- Describe how the BDM can be used to program on-chip FLASH memory.
- Describe the operation of the on-chip ICE system.
- Show examples of how the on-chip ICE system can be used to debug an application.

**Content:**

- 30 pages
- 4 questions

**Learning Time:**

- 45 minutes

The intention of this course is to explain two separate HCS08 systems; background debug mode (or BDM) and on-chip ICE, which are used for FLASH programming and application debugging. The first system you will examine in this course is the BDM. It provides a single-wire interface for reading and writing MCU memory, examining and changing CPU registers, setting up breakpoints, and single-stepping through a program. This type of interface is showing up in various forms on most modern MCUs.

The second system you will examine in this course is an on-chip real time in-circuit emulation (ICE) system. This on-chip ICE is equivalent to a traditional external emulator with hardware breakpoints, trigger logic, and a bus capture buffer that can capture address or data information relative to a trigger event. The on-chip ICE captures bus information without disturbing the application program in much the same way as a logic analyzer, except there is no expensive external hardware.

Please note that this on-chip ICE system is not available on most other MCUs. This is a very new development that completely replaces traditional external emulator systems. The on-chip ICE system comes at no extra cost on every HCS08 device.

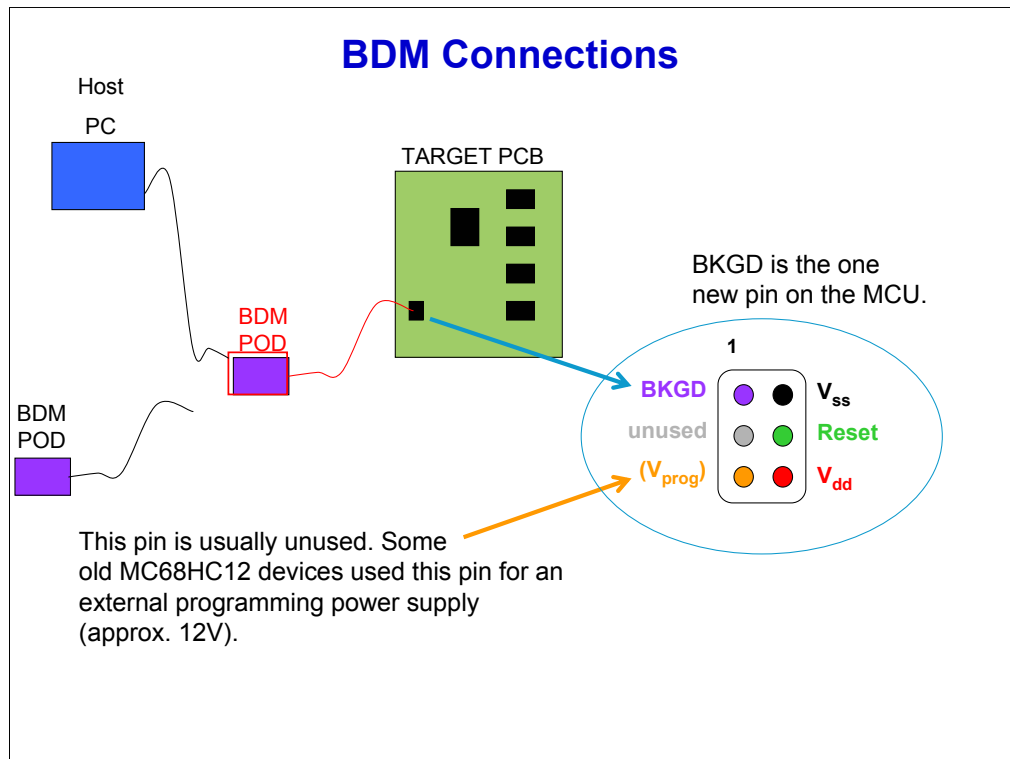
You should have some familiarity with in-circuit emulation before taking this HCS08 debug course.

## Background Debug Mode (BDM)

- Was introduced on the MC68HC12 family
- Is an advance over other serial debugging interfaces
- Allows the host to detect and adapt to the speed of the target MCU through the communication protocol
- Allows for robust communications

The first system we will look at is the BDM. Serial debugging ports similar to the BDM have been included on some MCUs for more than a decade. The BDM system in the HCS08 family devices was first introduced on the MC68HC12 family. In fact, the same BDM interface pod can be used for both the HCS12 and the HCS08 families of MCUs.

This BDM is an advance over other serial debugging interfaces because it uses only one MCU pin. The communication protocol provides a way for the host to detect, and adapt to, the speed of the target MCU. This protocol also allows approximately  $\pm 10\text{--}20\%$  speed mismatch between the host and target, so communications are very robust.



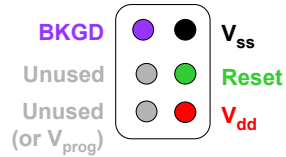
Here, you can see how you would connect from a host PC to the BDM of a target system for FLASH programming or debugging.

A BDM pod is used as the interface between a host PC and the BDM connection on the target PCB that contains an HCS08 MCU. One example of this BDM pod is the P&E Microcomputer Systems USB Multilink, which is about two inches wide by four inches long by one inch thick. In some cases, BDM pod circuitry may be included on an evaluation board so you only need a simple USB cable to connect the target system directly to the host PC.

The inset drawing shows the pin assignments for the standard 6-pin BDM connector with pin 1 in the upper left corner. Normally, only four pins of this connector are used. Pin 5 at the lower-left was used for some MC68HC12 devices for a 12V external programming power supply. HCS08 devices include an on-chip charge pump so there is no need for an external programming voltage source.

## BDM Physical Connection

- Single-pin interface
  - Adds, at most, one pin to the MCU
  - Selects mode (Normal or Active Background ) during reset
  - Standard 6-pin connector (BKGD, Reset,  $V_{ss}$ ,  $V_{dd}$ , and 2 unused)
- BDM “pod” interfaces from a host PC to a target HCS08 MCU
  - Target side connects to the standard 6-pin BDM connector on the application PCB
  - Host side connects to serial I/O, parallel printer port, or USB port
  - Pod translates higher-level debug commands from the PC into primitive BDM commands for the target MCU
- Pod can automatically adapt to the target communication speed using the sync command



Let's continue to look at the physical connection of the BDM. The BDM physical connector (a single-pin interface) is a standard 2-by-3 male, .025 inch, square-post header with pins on one-tenth-inch centers. It adds, at most, one pin to the MCU. During reset, the level on the BKGD pin selects either Normal User mode (when the level is high), or Active Background mode (if the level is low). Active Background is useful for FLASH programmers when the initial contents of the FLASH are unknown or all erased.

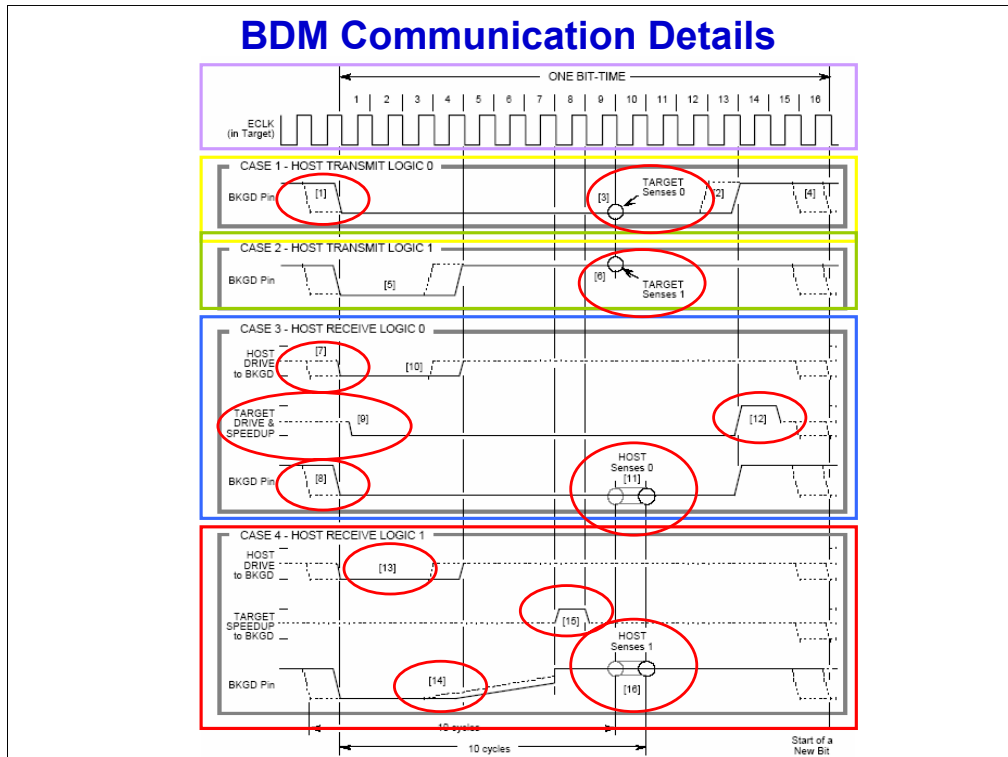
The BDM has a standard 6-pin connector that includes BKGD, Reset, Vss, Vdd, and two that are unused. Pin 1, the BKGD pin, is used for mode selection and single-wire bi-directional communications. Pin 2 is the ground reference. Pin 4 is used for Reset (when it is available on the target MCU), and Vdd is used by some BDM pods to provide a voltage reference for logic in the pod and to allow the host to display the Vdd voltage level of the target system. Pin 3 is not used and usually pin 5 is not used, although on some old MC68HC12 systems, this pin was used to supply an external programming voltage to the target MCU.

A host PC can always gain control of a target HCS08 MCU by sending commands via a BDM pod to the target, even if it is running a user program. The target side connects to the standard 6-pin BDM connector on the application PCB. Some development tools and pods even allow you to “hot-connect” in order to diagnose a target system that has crashed.

The host side of the BDM pods can be connected to the serial I/O port, a parallel printer port on a PC, or a USB port on a host PC. The USB type is becoming the preferred standard. The main function of a pod is to translate higher-level debugger commands from a host PC into primitive BDM commands for the target HCS08.

When a BDM pod is connected to a target system, or when the pod loses communication with the target, it issues a special sync command to detect the target communication speed and adapt to that frequency.

## BDM Communication Details



Let's discuss the details on BDM communication. The HCS08 BDM uses a custom synchronous serial protocol. Each bit time is a nominal 16 cycles of the target BDM clock frequency. The BDM clock is either the HCS08s bus clock or an alternate fixed clock source that may be different for different HCS08 derivative devices. In the case of the MC9S08GB60, this alternate fixed clock is an 8 MHz internally-generated clock source. This alternate clock source is useful to allow fast BDM communications, even when the bus clock is running at a low frequency, such as 32 KHz (divided by two).

A very brief description of the bit-level BDM protocol is shown here. For a much more detailed explanation, refer to the references at the end of this course. The timing diagram shows the four cases of a host transmitting a logic 0, transmitting a logic 1, receiving a logic 0, or receiving a logic 1. The target bus clock shown at the top of the diagram is for reference purposes.

The host transmit cases are very straightforward. The host initiates the start of all bit times. Because the host clock is asynchronous to the target clock, there is a 1-cycle uncertainty between where the host actually drives BKGD low and where the target perceives this falling edge. The target samples the level on BKGD nine cycles later. If the host wants to send a 0, it drives the BKGD low long enough to be sampled as a 0. If the host wants to send a 1, it drives the BKGD pin low for a much shorter time so it is sampled as a 1. In either case, BKGD must be high for a short time before the falling edge at the start of the next bit time.

The host receive cases are more complicated, so the host drive and target drive are shown separately. The effects of these drive signals are combined by the physical connections to the pseudo open-drain BKGD signal. As in all cases, the host initiates all bit times by driving the BKGD low so the target can detect the falling edge and there is a 1-cycle uncertainty about the position of this edge relative to the target BDM clock. The host samples the level on the BKGD pin ten cycles after it initiated the falling edge at the start of the bit time. Because the image is referenced to the target BDM clock and the host is asynchronous to this clock, the host sample point appears to have a 1-cycle uncertainty relative to the target BDM clock.

In case 3, where the host receives a logic 0, the target drives BKGD low when it detects the falling edge on BKGD. It holds it low long enough so the host samples a logic zero. Here, the target drives BKGD actively high for a single bus cycle and then reverts to high-impedance. This is called a "speed-up" pulse and it allows faster communications than would normally be possible with an open-drain connection. This also means the rise times on BKGD are not subject to an RC rise-time effect and the weak pullup resistor inside the target MCU is sufficient to hold the high level when nothing is driving BKGD.

In case 4, the host initiates the bit time with a short low pulse. Here, the BKGD pin begins a slow RC rise because nothing is driving it. Here, the target issues a 1-cycle speed-up pulse to force the BKGD pin quickly to a logic high before the host samples the level.

## BDM Commands

- **Sync command detects communication speed**
  - Drives BKGD pin low for at least 128 cycles; after a brief delay, target drives BKGD low for 128 BDM clock cycles
  - Provides the ability to automatically adapt to the speed of a target MCU
- **Non-intrusive commands (with a running application program)**
  - Memory read or write
  - READ\_STATUS – read the BDC status and control register
  - WRITE\_CONTROL – write to the BDC status and control register
  - BACKGROUND – engage Active Background mode
  - Setup the hardware breakpoints
  - Enable/disable hardware handshaking
- **Active Background commands**
  - Read or write CPU registers
  - GO – resume application program at current program counter
  - Trace1 – one user instruction at a time
  - Read or write memory and auto-increment address pointer (in H:X)
    - More efficient than non-intrusive memory read/write commands

Here is a look at the three types of BDM commands. In the first type, the BDM allows for a synchronization exchange, which is called a sync command. Technically, however, it does not have a binary command code like other commands, because at the time a sync exchange is initiated, the host does not know what the communication speed is. The host initiates the sync exchange by driving BKGD low for at least 128 cycles of the slowest expected target BDM clock. After a brief delay, the target MCU drives BKGD low for exactly 128 target BDM clock cycles. The host measures this 128-cycle low signal to determine the BDM communication speed. This provides the ability to automatically adapt to the speed of a target MCU.

Non-intrusive BDM commands can be issued even while the target MCU is running a user application program. These commands can be used to read or write to any target MCU memory location, including peripheral module control and status registers. Non-intrusive commands can also be used to read MCU status, or write control bits to enable or disable Active Background mode, setup the breakpoint that is in the background debug controller (BDC) module, or enable/disable hardware handshaking in BDM communications.

Active background commands can only be executed when the target MCU is halted in Active Background mode. These include commands to read or change the CPU registers, resume the user program at the current address in the program counter, or trace one user instruction at a time through application programs. There are also block-read and block-write commands that use the target MCU's H:X-register as a pointer to read or write memory and automatically increment the address pointer. These block-read and write commands are more efficient than the non-intrusive read-byte and write-byte commands because it is not necessary to send a 16-bit address for each byte that is accessed.

## BDC Registers

**These 3 BDC registers are not in the user memory map and can only be accessed by BDM commands**

ENBDM	BDMACT	BKPTEN	FTS	CLKSW	WS	WSF	DVF	<b>BDCSCR</b>
-------	--------	--------	-----	-------	----	-----	-----	---------------

ENBDM – Enable BDM

BDMACT – BDM active status (not running application code)

BKPTEN – Enable breakpoint C (in BDC)

FTS – Force/tag select (breakpoint on access/instruction execution)

CLKSW – BDM uses bus clock/alternate fixed clock (device specific)

WS – Wait or stop status

WSF – Wait or stop failure status

DVF – Data valid failure (current HCS08s don't have slow memory so not used)

								<b>BDCBKPT</b>

**16-bit breakpoint address**

Now, let's move on to background debug controller (BDC) registers. There is one 8-bit BDM control and status register and a 16-bit BDC breakpoint match register in the BDC module. These registers are not part of the user's memory map and can only be accessed with BDM commands.

When the HCS08 is reset in Normal User mode, the ENBDM control bit is reset to zero which disables Active Background mode. It is still possible to execute non-intrusive BDM commands, but the MCU cannot accidentally enter Active Background mode due to noise on the BKGD pin. BDMACT is a status bit indicating whether the HCS08 is in Active Background mode.

BKPTEN and FTS are used to enable and configure breakpoint C which is physically implemented within the BDC module. The FTS bit selects 'tag' or 'force' type breakpoints. Force breakpoints occur at the next instruction boundary after the address bus matches the 16-bit value in BDCBKPT. Tag-type breakpoints mark an instruction opcode as it is fetched into the instruction queue. If and when that opcode propagates through the instruction queue and is about to be executed, a breakpoint exception will be processed rather than the opcode at the address stored in BDCBKPT.

CLKSW is used to select the bus clock or the alternate fixed clock as the BDM clock, which controls the BDM communication speed. WS indicates whether the target MCU is currently in Wait or Stop mode. The BDM commands that read or write memory have "with-status" versions. When you issue one of these "with-status" commands, the returned data is accompanied by the current status information from BDCSCR.

WSF and DVF provide useful information about the memory accesses. WSF indicates the data access was not valid because the target MCU was in Wait or Stop mode. In this case, the host can force the target out of Stop or Wait into Active Background mode, repeat the memory access, and then return to Stop or Wait. DVF indicates an error related to slow memory, but since no HCS08s have slow memory, this bit is not used.

## BDC Registers

**This registers is in the user memory map  
but it can only be accessed by BDM commands**

0	0	0	0	0	0	0	BDFR	<b>SBDFR</b>
---	---	---	---	---	---	---	------	--------------

BDFR – Writing 1 to this bit using a WRITE\_BYTE command to the address of SBDFR causes an MCU reset. This provides a way to reset the target MCU from the host PC without requiring a reset pin on the target MCU.

There is one other control bit related to BDM. The system BDM force reset register (SBDFR) is located in the user's memory map, but this register can only be written using a BDM memory-write command. Writing one to the BDFR bit, causes an MCU reset. This provides a way for a host debugger to reset the target MCU from the host PC even when the target does not have a reset pin.

Because user software cannot write to this bit, a user program cannot accidentally reset the MCU.



## FLASH Programming through BDM

- BDM can get control of an HCS08 even if FLASH is blank and/or secure
- If secure, you can only erase the FLASH
- Programmers download a program into RAM and the loader program controls the FLASH programming process
  - Fastest programmers use alternating buffers in RAM for data
    - While data for one buffer is being loaded via BDM, data from the other buffer is programmed into FLASH
    - Non-intrusive data writes are used for programming and data transfers
  - CRC used to verify after programming rather than reading the data via BDM again

One of the most valuable uses of BDM is for programming the FLASH memory. By simply including a standard BDM header on an application PCB, you can perform post-assembly factory programming or field re-programming of the FLASH memory without having to remove the MCU from the application PCB.

The BDM can always gain control of a target system even if the FLASH memory is blank or secured. In the case of secure FLASH, the BDM can only erase the FLASH and check to be sure it is erased. After the FLASH has been erased, the BDM can be used to re-program it.

In the usual programming sequence, FLASH is erased, a loader program is downloaded into the RAM, and control is passed to this loader program. The loader program controls the transfer of data to be programmed and, in parallel, controls programming of this data into FLASH. When the programming is completed, a verify operation is performed.

The fastest programmers use two alternating data buffers in RAM so data can be read in through the BDM into one buffer, while data from the other buffer is being programmed into the FLASH. Non-intrusive write-byte commands are used so that programming can take place in parallel with the BDM data transfers. To verify the data, you can use a calculated CRC rather than reading all of the data in through BDM a second time. During programming you would compute a CRC value. After programming, the loader program in RAM can perform an independent CRC calculation on the contents in FLASH and then compare this to the CRC value that was computed during loading.

## Question

Can you remember the information presented so far? Drag the boxes on the left to the appropriate statements on the right. Each letter may be used more than once. Click “Done” when you are finished.

**A** True

**B** False

**B** The sync command allows the target HCS08 to adapt to the communication speed of the host PC.

**A** The BDM is an advance over other serial debugging interfaces because it uses only one MCU pin.

**A** A BDM pod is used as an interface between a host PC and the BDM connection on the target PCB that contains an HCS08 MCU.

**B** Non-intrusive BDM commands cannot be issued while the target MCU is running a user application program.

Done

Reset

Show  
Solution

Let's take a moment to review the information covered so far.

Correct.

The sync command allows the host PC to adapt to the speed of the target HCS08. Using only one MCU pin, the BDM is an advance over other serial debugging interfaces. A BDM pod is used as an interface between a host PC and the BDM connection on the target PCB. Sometimes, the BDM pod circuitry may be included on an evaluation board, so you only need a simple USB cable to connect the target system directly to the host PC. Non-intrusive BDM commands can, in fact, be issued while the target MCU is running a user application program. These commands can be used to read or write to any target MCU memory location. Click the forward arrow to continue on to the next page.]

## In-Circuit Emulation (ICE)

- Replaces traditional external emulators
- Eliminates troublesome problems
- Can access many internal control signals unavailable to external systems
- BDM and ICE systems are completely independent and perform different functions
- ICE is a unique system

Next, we will discuss the new on-chip real-time ICE system. This system completely replaces traditional external emulators that cost thousands of dollars.

In addition to eliminating the most troublesome problems of an external emulator, this on-chip system has access to many internal control signals that would not be available to an external system.

Many users mistakenly think of the on-chip ICE as an extension of the background debugging system. This is incorrect. The BDM and ICE systems are completely independent systems and they perform very different functions. Many MCUs have serial interface debug ports somewhat like the HCS08's BDM, although most use more pins and are more intrusive than BDM. But no known 8-bit MCUs outside Freescale have anything like the unique on-chip real-time ICE system.

Although the BDM is commonly used as a convenient way to setup the internal emulator and read out the captured results after trace runs, there are no direct connections between these systems. In fact, if there is a resident serial monitor program in the target MCU, the host debugging tools can access the on-chip ICE through a serial I/O port on the host PC just as if a BDM pod was being used as the interface to the host.

## Real-Time vs Single Step Debug

- The address comparators in the on-chip ICE can be used as hardware breakpoints and the BDM system can be used to single step through a program.
- Many programs function differently when run in real time as compared to being traced one instruction at a time or stopped by breakpoints.
- Emulators (including HCS08 on-chip ICE) capture bus information in real time without disturbing the application program.
  - The on-chip ICE may continue to run or halt after the information is captured
    - A separate breakpoint can stop the program at some later point
    - It is important in motor drives to avoid stopping the firmware at the wrong place
    - It may be important to avoid stopping a communication routine in the middle of a transfer.

Without an emulator, debugging uses non-real-time techniques including hardware breakpoints and single-stepping. This intrusive type of debugging can be very useful and has its place. The comparators in the on-chip ICE system can be used as conventional hardware breakpoints in an HCS08 system to allow traditional intrusive debugging. The BDM system can be used to single step through a program.

Many application programs present problems for these less-sophisticated intrusive debugging techniques because they function differently when they are run in real-time as compared to being traced one instruction at a time or stopped by a conventional breakpoint.

Logic analyzers and emulators, including the HCS08's on-chip ICE, solve these problems by capturing address and data information on the fly without disturbing the real-time operation of the application program. With the on-chip ICE system, you can choose to let the application program continue to run in real time after a debug trace, or you can choose to halt the application program after the information is captured and enter Active Background mode to await further commands. You can even set a separate breakpoint to stop the user application at some more convenient point in the future after the trace run is complete.

Some examples of places where this would be useful include motor drive applications and communications routines. In motor drives that control very high-current H-bridges, stopping the firmware at the wrong place can cause the destruction of driver transistors due to excessive shoot-through currents. Or, it might be undesirable to stop a mechanical sequence at the wrong place. In communication routines, the program may fail or other systems may fail if the firmware is stopped in the middle of a transfer.

## Problems With External ICE

- **Umbilical cable is awkward, expensive, noisy, and capacitive**
  - Can't fit in tight places
  - Affects loading and timing (very difficult to run reliably at full speed)
- **Uses many MCU pins**
  - Normal pin functions must be rebuilt in external logic
  - Not enough pins on small devices (16-pins or less)
- **Imperfect emulation**
  - Timing, drive characteristics, port replacement units, circuit loading, typically don't use application's crystal components
- **Very expensive (thousands of dollars)**
- **Problems keeping up with faster bus speeds and smaller packages**
- **Unknown if a problem is in your system or caused by the emulator**

Let's look at some of the more serious problems with traditional external emulators. These external emulators connect to your target system through an umbilical cable or impedance-matched flat cable with dozens of connection signals. These cables are awkward, expensive, and noisy. They don't fit into tight spaces and they cause undesirable loading and timing problems that make it very difficult to operate reliably at full bus speeds.

These emulators also require many MCU pins, which must share functions between user I/O and emulator address, data, and control signals. When the emulator is in use, the normal function of these pins must be rebuilt in external logic. Usually there are subtle timing and drive differences between these rebuilt functions and the normal internal MCU functions they replace. Some new MCUs have as few as six or eight pins so there aren't enough pins even if they do multiplex functions.

Besides the timing and drive differences already mentioned, there is also the problem of imperfect emulation. Emulators usually substitute an emulator clock source instead of using your crystal and oscillator components. Therefore, the emulator is not actually emulating the entire original system.

Traditional external emulators are very expensive. They typically cost thousands of dollars as compared to the negligible cost of the extra transistors inside every HCS08 MCU. The external emulators cannot keep up with faster bus speeds and smaller/denser packages such as QFN and ball-grid-array packages. Soon it will be impossible to build external emulators at all for new MCUs.

Finally, traditional external emulators have typically been difficult to work with. When there is a problem, it is hard to tell whether it is with the MCU, or a problem related to the emulator, its complicated connection system, or simply noise entering the vulnerable umbilical connection.

## Benefits of On-Chip ICE

- **Reduces interconnect by putting ICE inside**
  - Capture buffers, comparators, and logic are becoming much smaller than bonding pads
  - Full access and debug even in tight places
- **Eliminates timing, loading, and drive issues**
  - Target is the actual MCU not just an emulated equivalent
  - Capture buffer and logic are the same as the target MCU so no marginal timing
- **No issues with temperature and voltage**
- **Emulates your MCU including oscillator components**
- **Eliminates expensive external emulator box and interconnect**

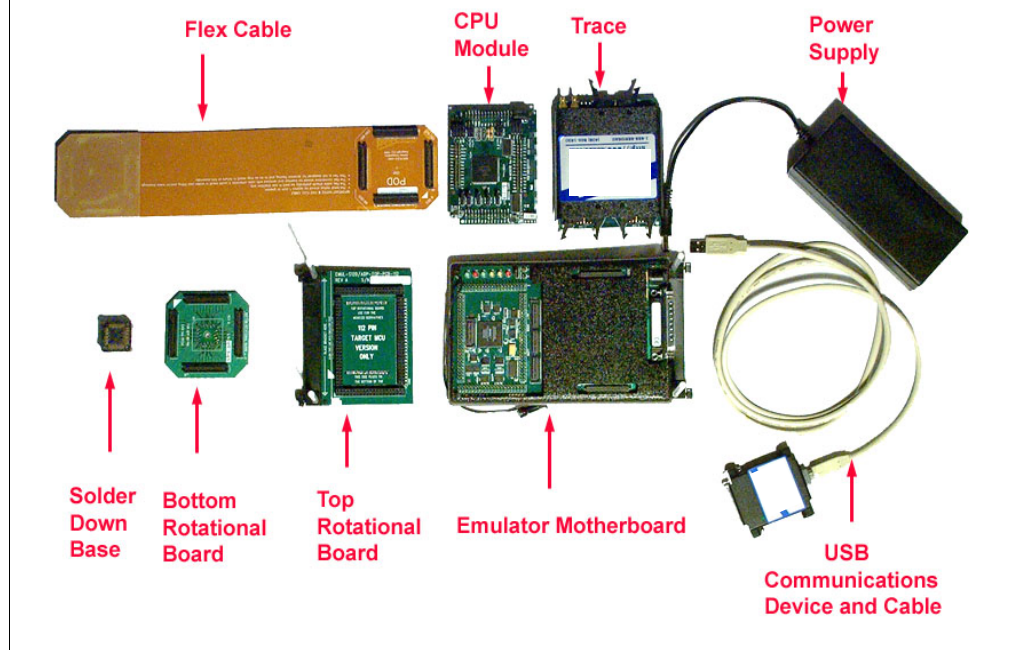
The HCS08's advanced on-chip ICE system eliminates the problems we just looked at and offers new capabilities that are not available with an external emulator. First of all, the umbilical connection is completely gone. All ICE signals are connected inside the die of every HCS08 so emulation does not alter the timing or loading of MCU busses at all.

The capture buffer, comparators, and trigger logic can be included at no extra cost. With modern silicon processes, thousands of logic transistors take less space than a few bonding pads. This means that you get full access and debug, even in tight places. It is less expensive to add the emulator logic inside the part than it would be to add bonding pads to get internal signals out to an external emulator.

The on-chip ICE eliminates concerns about timing, drive characteristics, and subtle functional differences between the MCU in your end product and the emulator. This is because the emulator is the actual MCU that is shipped in a product rather than a substitute emulated version of the MCU. The internal emulation logic runs over the same temperature and voltage ranges as the MCU because it is the MCU. The on-chip ICE emulates your entire MCU system including the oscillator components.

The on-chip ICE also eliminates the expensive external emulator box and interconnect. Because you no longer need external emulation hardware, this allows you to save thousands of dollars.

## External Emulator System

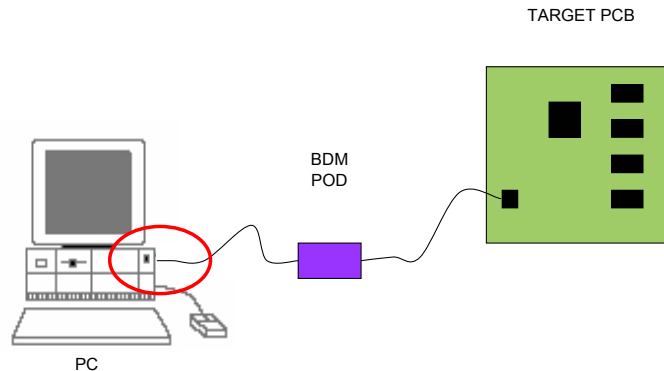


Now, let's look at an example of a traditional external emulator. On the left, you can see the umbilical flex cable that connects your system to the emulator electronics, a solder-down base that replaces the MCU after the application system is reworked, and bottom and top rotational boards that allow you to make the flex cable exit your application PCB in the least-inconvenient direction.

The emulator motherboard includes power supply circuits, communications, and some emulation circuitry as well as connectors for the CPU module and optional trace module. The trigger comparators and trigger logic for the emulator are built on the emulator motherboard. The CPU module breaks out CPU address, data, and control signals for the emulator and rebuilds the normal user functions of those pins. The optional trace module includes a large buffer memory that performs basically the same function as the capture buffer in the on-chip emulator of the HCS08.

## Debug Using BDM and On-Chip ICE

- Logic similar to the circuitry in the previously shown external emulator is included as logic inside every HCS08 MCU
- BDM provides a simple way to control and monitor trace runs
- BDM can read out trace capture information without stopping the target application



Here is a much simpler HCS08 system that uses the on-chip ICE. The emulator hardware shown previously is built inside the HCS08. Technically, the external emulator hardware shown on the previous slide also includes the equivalent of the BDM pod as circuitry on the emulator motherboard.

With the on-chip ICE, you do not alter the target PCB at all. Although the on-chip ICE has no external connections on the MCU, you do need to get access to setup trace operations and read out results. Here, you can see that you can control and monitor trace runs through the single-wire BDM interface by attaching a BDM pod to a standard 6-pin BDM header on your target application PCB. Note that the BDM can read out trace capture information without stopping the target application.

The host PC connects to the BDM pod through a port such as a USB port on the PC. Development software on the PC, such as CodeWarrior®, can then be used for FLASH memory programming and debugging operations.



## Question

**Is the following statement true or false? Click Done when you are finished.**

“Hardware breakpoints and single stepping are real-time debugging techniques.”

True

False

Consider this question concerning real-time capture versus single instruction trace.

Correct.

The answer is False. The on-chip ICE system uses real-time techniques, however traditional hardware breakpoints and single-stepping are examples of intrusive non-real-time debugging. The on-chip ICE system is based on capturing bus information non-intrusively, and reading it out at a later time. The comparators in the on-chip ICE system can be used as conventional hardware breakpoints and the BDM system can be used to single step through a program.

# On-Chip ICE Features

- **Three hardware breakpoint comparators**
  - Two full-featured breakpoints in the on-chip ICE module (comparators A, B)
  - One address-only breakpoint inside the BDC (comparator C)
- **Nine trigger modes for on-chip ICE trace runs**
  1. **A-only** (basic)
  2. **A or B** (basic)
  3. **A then B** (sequence of events)
  4. **Event-only B** (captures data on access to address B)
  5. **A then event-only B** (sequence trigger then capture data on access to B)
  6. **A and B data** (full compare of address and data) (comprehensive)
  7. **A and not B data** (full compare of address and not data)
  8. **Inside range** ( $A \leq \text{address} \leq B$ ) (access or execute within address range)
  9. **Outside range** ( $\text{address} < A$  or  $\text{address} > B$ )
- **8-Stage real-time bus capture buffer**
  - Trace after trigger (begin trace) or trace until trigger (end trace)
  - Halt application and go to Active Background mode when the buffer is full - or
  - Continue running application (host can read out results non-intrusively through BDM)
  - Profile mode allows periodic capture of current PC address

Let's continue to look at the features of the on-chip ICE system. The HCS08 has three hardware breakpoints. Breakpoint comparators A and B are located in the on-chip ICE module. Breakpoint comparator C is located in the BDC module that was discussed previously. Comparators A and B can be used to match two separate addresses or they can be configured so that A compares to an address and B compares to the data for the same bus cycle. Comparators A and B also allow optional read-write and tag/force qualifiers.

The on-chip ICE uses the comparators and trigger logic to setup the conditions that start or end a trace run. The HCS08 supports nine trigger modes. 'A-only' mode and 'A or B' mode are basic modes where the trigger is based on a comparator match. 'A then B' mode requires a sequence to be satisfied. 'Event-only' modes are used to record the data that is read or written to a selected address. For 'Event-only B' mode, the address is specified in comparator B. For 'A then event-only B' mode, the address in comparator A must be detected and then the on-chip ICE will record the data for any subsequent access to address B.

In Full-compare modes, comparator A specifies the address to match and comparator B specifies the data to match or not match, when the address in A matches. 'A and B data' mode triggers when address and data match A and B comparators in the same cycle. 'A and not B data' mode triggers when the address in A matches but the data does not match the value in B. Range modes use comparators A and B to set the ends of the range to be compared. 'Inside range' mode triggers if the address falls between A and B, and 'outside range' mode triggers if the address ever falls outside the range specified by A and B.

The bus capture buffer holds eight words of capture information. In 'begin' trace runs, bus information capture begins when a trigger event is detected. In 'end' trace runs, capture begins immediately and the capture buffer is filled in a circular fashion. Capture stops when a trigger event occurs so the capture buffer holds the eight most-recently captured values.

The on-chip ICE can be configured to continue running after a trace run is complete, or it can be set to halt and go to Active Background mode when the run is complete. If you choose to continue running, non-intrusive BDM commands can be used to read the contents of the capture buffer without stopping the application program.

A profiling mode is available when the on-chip ICE is disarmed. When the arm bit is zero (each time the capture FIFO is read), a previously captured value is read, the current value of the program counter is copied into the capture buffer, and the buffer pointer is advanced. By reading the capture buffer periodically over a period of time, the host can build up a map of where the application program has been executing.

## On-Chip ICE Registers

### Debug Comparator A – Address match register


DBGCAH

DBGCAL

### Debug Comparator B – Address or Data match register


DBGCBH

DBGCBL

### Debug FIFO – Access information in capture buffer


DBGFH

DBGFL

Select a button to examine more on-chip ICE registers.



Debug Control  
Register



Debug Trigger  
Control Register



Debug Status  
Register

On-chip ICE registers include two 16-bit comparator match registers, a 16-bit FIFO access register, two 8-bit control registers, and an 8-bit status register. These registers are located in the user's memory map so they can be accessed from application programs or through the BDM. BDM is the most common way these registers are accessed.

Comparator match register A is used to store an address to be matched. Comparator B is used for either a second address or a data value. In full-compare modes, you would store the 8-bit data value to match in the low half of comparator match register B.

The 16-bit debug FIFO register is used to access information in the capture buffer. The capture buffer holds up to eight 16-bit words of captured information. If you read the 16-bit FIFO register eight times after the end of a trace run, you will read out the last eight values that were captured in the same order that they were captured. Normally, you would read the count value from the debug status register to determine how many valid values are in the capture buffer before reading out the results.

If you read the FIFO when the debugger is disarmed, the oldest value in the FIFO is returned, the current program counter value is captured into the FIFO, and the FIFO pointers are updated. By reading the FIFO in this manner periodically while an application program is running, the host debugger tool can build up a history of where your application program is executing. This is called Profile mode.

Select a button to examine more on-chip ICE registers. Once you have viewed all of the pages, click the forward arrow to advance to the next page.

## On-Chip ICE Registers

### On-Chip ICE Debug Control Register

DBGEN	ARM	TAG	BRKEN	RWA	RWAEN	RWB	RWBEN	DBGC
-------	-----	-----	-------	-----	-------	-----	-------	------

DBGEN – Enable on-chip ICE debug system

ARM – Arm control - arm a trace run to start when qualified

TAG – Trigger at an instruction (1) or a simple comparator match (0)

BRKEN – 1 = Break to Active Background or SWI when trace complete  
0 = continue to run (host can use BDM to read results)

RWA – Select the value of read/write to match for comparator A

RWAEN – Enable read/write as a qualifier for comparator A

RWB – Select the value of read/write to match for comparator B

RWBEN – Enable read/write as a qualifier for comparator B

Select a button to examine more on-chip ICE registers.



**Debug Control  
Register**



**Debug Trigger  
Control Register**



**Debug Status  
Register**



**Return to  
Comparator/FIFO  
Registers**

DBGEN is used to enable the on-chip ICE system. This is usually done once at the beginning of a debugging session. During the session, you would typically setup the triggering conditions you want and then set the ARM bit just before starting to execute the user application program. When the trigger conditions are satisfied and the capture buffer is full, the ARM bit is automatically cleared and capture operations stop. Depending upon the settings in BRKEN, the target system will either continue running or it will halt and enter Active Background mode to await further commands.

TAG and BRKEN control the way the trace run will end. TAG selects either Tag mode or Force mode to end the trace run. If TAG equals one, the instruction opcode at the trigger address is marked as it is fetched into the instruction queue. If and when this opcode propagates through the queue and is about to be executed, the ARM bit is cleared to end the trace run. If TAG equals zero, Force mode is selected. In this case, when the address that caused the trigger matches, an interrupt request is issued to the CPU. When the currently executing instruction finishes, the ARM bit is cleared to end the trace run.

BRKEN determines whether the CPU will halt or continue running the user program when the trace run ends. If BRKEN equals one, the CPU will halt and the MCU will enter Active Background mode to await further commands. If BRKEN equals zero, the CPU will continue to execute the user application when the trace run ends. Non-intrusive BDM commands can be used to detect that a trace run is finished and then read out the results from the capture buffer. If desired, breakpoint C can be set at a later point in the application program to halt the application program and enter Active Background mode.

The remaining four control bits in DBGC can be used to optionally extend comparators A and B to 17 bits. The read-write signal must match in addition to the address. For example, to include read-write with comparator A, set RWAEN to one and then put either 1 or 0 into the RWA bit to choose the value of read-write to match.

Select a button to examine more on-chip ICE registers. Once you have viewed all of the pages, click “Return to Comparator/FIFO Registers” to go back to the first “On-Chip ICE Register” page or click the forward arrow to advance to the next page.

## On-Chip ICE Registers

### On-Chip ICE Debug Trigger Control Register

TRGSEL	BEGIN	0	0	TRG3	TRG2	TRG1	TRG0	DBGT
--------	-------	---	---	------	------	------	------	------

TRGSEL – Trigger type (used to qualify A and B comparators)

1 = Tag type trigger on the instruction at trigger address

0 = Force type on any memory access to trigger address

BEGIN – 1 = Begin type trace - begin filling capture FIFO at trigger

0 = End type trace - continuously fill FIFO until trigger

TRG3:TRG2:TRG1:TRG0 – Select Trigger mode

0000	A-only	0101	A and B data (full mode)
0001	A or B	0110	A and not B (full mode)
0010	A then B	0111	Inside range: $A \geq \text{addr} \leq B$
0011	Event-only B	1000	Outside range: $\text{addr} < A$ or $\text{addr} > B$
0100	A then event-only B	1001-1111	Reserved - no trigger

Select a button to examine more on-chip ICE registers.



Debug Control  
Register



Debug Trigger  
Control Register



Debug Status  
Register



Return to  
Comparator/FIFO  
Registers

TRGSEL is used to select whether an instruction opcode qualifier will apply to address comparisons for triggers A and B. If TRGSEL equals one, tag type triggers are selected and a successful match from comparators A or B is marked. However, no actual trigger occurs until and unless the instruction opcode at the marked address is about to be executed. If TRGSEL equals zero, successful matches from comparators A and B cause an immediate trigger event.

The BEGIN bit selects the position of the trigger event relative to the data in the capture buffer. If BEGIN equals one, a begin type trace run is selected and the capture buffer begins to fill when a trigger event occurs. When the capture buffer gets full, the trace run ends.

If BEGIN equals zero, an end type trace run is selected and the capture buffer starts to fill immediately in circular fashion. When a trigger event occurs, the capture buffer stops capturing any new information. The values in the capture buffer are the eight most-recently captured bus events.

The four TRG bits select one of the nine trigger modes as shown in the table.

Select a button to examine more on-chip ICE registers. Once you have viewed all of the pages, click "Return to Comparator/FIFO Registers" to go back to the first "On-Chip ICE Register" page or click the forward arrow to advance to the next page.

## On-Chip ICE Registers

### On-Chip ICE Debug Status Register

AF	BF	ARMF	0	CNT3	CNT2	CNT1	CNT0	DBGS
----	----	------	---	------	------	------	------	------

AF – Comparator A matched

BF – Comparator B matched

ARMF – Arm status - Image of ARM bit in DBGCR (read-only)  
0 = trace run complete

CNT3:CNT2:CNT1:CNT0 – Count of valid capture FIFO entries

Select a button to examine more on-chip ICE registers.



[Debug Control Register](#)



[Debug Trigger Control Register](#)



[Debug Status Register](#)



[Return to Comparator/FIFO Registers](#)

The AF and BF status flags indicate whether comparators A or B have matched. In order to match, the address must match, read-write must match if it is enabled, and the instruction opcode must have propagated through the instruction queue if TRGSEL was set to one.

ARMF is a read-only image of the arm bit which is located in the DBGCR register. ARMF is set whenever you write a one to arm and it is cleared to zero when a trace run is completed.

The four CNT bits indicate the number of valid entries in the capture buffer after a trace run ends. This count automatically increments as values are captured into the buffer. Note that the count does not decrement as values are read out of the buffer. The host debugger is expected to read out the count after a trace run and then read that many words of information out of the FIFO buffer, keeping track of the count as values are read.

Select a button to examine more on-chip ICE registers. Once you have viewed all of the pages, click “Return to Comparator/FIFO Registers” to go back to the first “On-Chip ICE Register” page or click the forward arrow to advance to the next page.

## Question

**Identify the number that will complete the sentence. Select the correct answer and then click Done.**

The on-chip ICE uses the A and B comparators and trigger logic to start or end a trace run. The HCS08 supports \_\_\_\_\_ trigger modes for on-chip ICE trace runs.

- a. One
- b. Five
- c. Nine
- d. Twelve

Let's see if you can remember the on-chip ICE features.

Correct.

The HCS08 supports nine trigger modes. They are A-only, A or B, A then B, event-only B, A then event-only B, A and B data, A and not B data, inside range, and outside range. Click the forward arrow to continue on to the next page

## Tag vs Force Breakpoints

- **Force-type breakpoints cause a trigger or breakpoint at the next instruction boundary after the address compare is true.**
  - Used to detect a memory access to a specific address
- **Tag-type breakpoints cause a trigger or breakpoint as the instruction at the selected address is about to be executed.**
  - They are used to detect instruction execution
  - A branch or interrupt could cause the CPU to flush the contents of the queue and execute instructions from another address
  - Instruction opcodes are marked as they are fetched
  - This type of trigger is not usually possible on an external emulator
  - The address must be the address of the first byte of object code for an instruction (the opcode)

The concept of tag versus force breakpoints is used in two contexts in the on-chip ICE system. This distinction applies as a qualifier for the A and B comparators, as well as for the CPU breakpoint action at the end of a trace run.

Force-type breakpoints are the most straightforward and are used to detect a memory access to a specific address. Force-type breakpoints act as simple interrupt requests when the address of interest is detected. Force-type breaks cause a trigger or breakpoint at the next instruction boundary after the address compare is true. Because the CPU cannot simply stop in the middle of executing a multiple-cycle instruction, the break remains pending until the next available instruction boundary. This is how breaks would work in a traditional external emulator.

Tag-type breakpoints cause a trigger or breakpoint as the instruction at the selected address is about to be executed. Tag-type breakpoints are used to detect instruction execution. They take into account the instruction queue, which is sometimes called a pipeline. Instruction opcodes are fetched into the instruction queue a few cycles before they are executed by the CPU. However, before they get through this queue, a branch or interrupt could cause the CPU to flush the contents of the queue and start executing instructions from some other address.

With tag-type breaks, instruction opcodes are marked as they are fetched. If they get through the queue and are about to be executed by the CPU, the break takes effect rather than executing the tagged instruction. This type of break is not usually possible on an external emulator because it requires intimate knowledge of the instruction queue mechanism and control signals, which may not be available to an external emulator.

If a tag-type break is specified, you must make sure the address you store in comparator A or B is the address of the first byte of object code for an instruction. Force-type breaks can use any address.



## Capture Information Filtering

- **Because the on-chip capture buffer is limited in size, filtering techniques are used to capture the information you need rather than every bus access.**
  - Increases the effective depth of the capture buffer
- **Change-of-Flow (COF) events occur when a program jumps, branches, or interrupts to an address other than the next address after the current instruction.**
  - The host debugger easily reconstructs the complete program flow
  - 8 COF events can easily correspond to 50 instructions in a typical program
- **Event-Only modes capture data corresponding to a chosen address.**


Now, let's take a look at capture information filtering. One of the few advantages a traditional external emulator has over the on-chip ICE is the depth of the capture buffer is limited only by the amount of money you are willing to spend for fast static RAM. For that reason, typical external emulators can have thousands of words of capture information. The capture buffers in the on-chip ICE systems are limited in size. Therefore, these systems use intelligent filtering techniques to make sure they capture only the information required to debug application problems. External emulators typically capture every bus access and let you sort out what is useful after a trace run. This shows an increase in the effective depth of its capture buffer.

The on-chip ICE in the HCS08 has eight words in its capture buffer. There are two techniques that filter the captured information: change-of-flow trace runs and event-only trace runs. As a program is executed, most address values are entirely predictable. However, when the address does not follow the usual predictable sequence, it is called a change-of-flow, or COF. In other words, a COF event occurs when a program jumps, branches, or interrupts to an address other than the next address after the current instruction.

If you capture a COF, the development host can reconstruct the complete program flow after a trace so there is no need to capture those bus cycles. You will see an example later that shows that you can typically reconstruct fifty or more instructions from only eight COF values after a typical trace run.

A second form of intelligent filtering is used for event-only trace runs. In these trace runs, the information in the capture buffer consists of data values that correspond to accesses for a chosen address. For example, you can setup a trace run to capture all writes to a specific control register, but only after the instruction at address X is executed. At the end of the trace run, the capture buffer will contain the eight most recent values written to the selected register.

## CodeWarrior® User Interface

- **Context-sensitive setup of trace runs**
  - Right-click on instructions to set tag-type trigger reference points or breakpoints
    - Trigger points marked with **A** or **B** icons
    - Breakpoints identified by arrow  icons
  - Right-click on variables or data values in memory window to setup event-only data capture trace runs
  - Only context-appropriate choices shown in menu lists
- **New trace component window used to display trace results**
  - Open a trace window before starting a trace run
  - States (lines) in trace window linked to source window

You will probably interact with the on-chip ICE through a software debugger tool, such as CodeWarrior, running on the host PC. You shouldn't have to memorize all of the control bits and what they do. CodeWarrior translates your intentions into the appropriate settings to accomplish common debugging tasks.

While debugging your application, you simply set breakpoints or mark addresses or instructions where you want a trigger. Once you set a trigger point, you need to specify the trigger mode. CodeWarrior helps simplify these operations by making the setup of trace runs context-sensitive and by including descriptive choices in drop-down menus. For example, if you right-click on an instruction in a source window, CodeWarrior knows you want to set a tag-type breakpoint or trigger point. Of course, you can always override such assumptions, but this should rarely be necessary.

Breakpoints and trigger points are indicated by colored icons. Trigger points A and B are marked with a bold red 'A' or bold blue 'B'. Breakpoints are indicated by a red arrow pointing at a small vertical bar.

If you right click on a variable in a variable window or a data value in a memory display window, CodeWarrior assumes you want to establish a force-type break or event-only trace run. Only the settings that are context-appropriate will be offered in the drop-down menus so it is easier and more intuitive to setup trace runs.

In order to support the new on-chip ICE system, CodeWarrior has added a new trace component window. When you intend to use the on-chip ICE, you would open a trace window on the debugger screen. After the race run, the results will be displayed in that window. Highlighted lines in the trace window and other debugger windows allow you to associate events in the trace window with lines in your source program.

## Example

### Tracing a FLASH Programming Routine

- The capability of the on-chip ICE is demonstrated by tracing a program sequence that could not be traced using conventional breakpoints and single-stepping.
- DoOnStack transfers a routine onto the stack, calls it to program 'DoOnStack' in FLASH, then returns to FLASH.
- SpSub is the 24-byte sub-routine that is copied onto the stack.
- The on-chip ICE is used to trace the entry to SpSub.
- We don't want to halt after this trace run.
- Breakpoint C (in BDC) is used to halt the program after it has returned from SpSub (and FLASH is back in the memory map).

To give you an idea of what the on-chip ICE system is capable of, we will run through an actual example. We will be debugging an HCS08 program that programs the on-chip FLASH memory. This program would be difficult to debug using traditional hardware breakpoints and single-stepping because the FLASH memory is temporarily removed from the memory map during a critical period in program execution. Breakpoints and debugger displays cannot operate correctly at such times. The on-chip ICE lets us capture critical bus events without disturbing the running program.

The 'Do-On-Stack' program transfers a small sub-routine from FLASH onto the stack, and then calls this sub-routine to perform a FLASH programming operation. After the FLASH programming operation is done, the sub-routine causes a return to 'do-on-stack' in FLASH where the stack space used by the small sub-routine is de-allocated and the main program continues.

SpSub is the small 24-byte sub-routine that is copied onto the stack. This sub-routine is written in position-independent code so it will still run as expected even when it has been relocated to a new address.

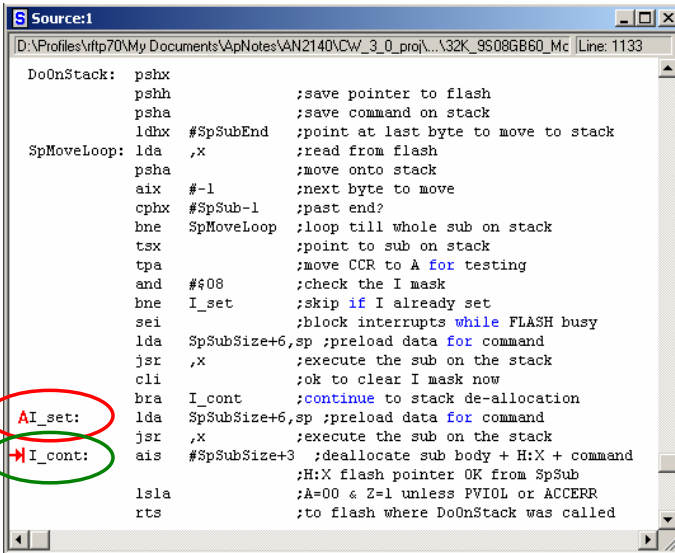
We will use the on-chip ICE system to trace program execution as we enter the SpSub routine in stack RAM. We don't want to halt after this trace run because the FLASH is out of the memory map at the time. Instead, we will set breakpoint C to halt the program after the FLASH is back in the map and we have returned to the do-on-stack program.

## DoOnStack Routine

Right-click at 'I\_set:' to  
"Set Trigger Address A"

Right-click in source  
window to choose  
'Trigger Settings' for  
"Start recording after  
trigger hit and DO NOT  
halt when FIFO is full"

Right-click at 'I\_cont:' to  
'Set Breakpoint' –  
Because the ICE  
module is already in  
use, this will use  
breakpoint C



```
Source:1
D:\Profiles\ltp70\My Documents\ApNotes\AN2140\CW_3_0_proj\...\32K_9S08GB60_Mc |Line: 1133

DoOnStack: pshx          ;save pointer to flash
           pshh          ;save command on stack
           ldhx #SpSubEnd ;point at last byte to move to stack
SpMoveLoop: lda ,x       ;read from flash
           psha          ;move onto stack
           aix #-1        ;next byte to move
           cphx #SpSub-1  ;past end?
           bne SpMoveLoop ;loop till whole sub on stack
           tsx           ;point to sub on stack
           tpa          ;move CCR to A for testing
           and #008       ;check the I mask
           bne I_set      ;skip if I already set
           sei           ;block interrupts while FLASH busy
           lda SpSubSize+6,sp ;preload data for command
           jsr ,x         ;execute the sub on the stack
           cli           ;ok to clear I mask now
           bra I_cont     ;continue to stack de-allocation
           lda SpSubSize+6,sp ;preload data for command
           jsr ,x         ;execute the sub on the stack
           ais #SpSubSize+3 ;deallocate sub body + H:X + command
                           ;H:X flash pointer OK from SpSub
           lsia          ;A=00 & Z=1 unless PVIOL or ACCERR
           rts           ;to flash where DoOnStack was called

AI_set:
I_cont:
```

Here, you can see the do-on-stack program in a CodeWarrior source window. If you would like to study this routine in greater detail, refer to the references at the end of this course and application note AN2140.

Upon entry to the do-on-stack routine, a data value is on the stack and the FLASH address to operate on is in the H:X index register. To setup the example trace run, right-click at 'I-set' and choose "Set Trigger Address A" from the pop-up menu. CodeWarrior marks the line with a bold red letter 'A' to the left of 'I-set'. Next, right-click anywhere in the source window and choose "Trigger Settings" and then "Start recording after trigger hit and DO NOT halt when FIFO is full". Notice that you don't have to remember the on-chip ICE control registers and bits. CodeWarrior sets these controls correctly based on your choices.

Next, right-click at 'I-continue' and choose 'Set Breakpoint'. Because the on-chip ICE is being used, breakpoints A and B are not available and CodeWarrior automatically sets up breakpoint C at this address. The breakpoint is marked by the red arrow pointing at the short vertical bar. This icon is positioned to the left of the line corresponding to the breakpoint.

Before starting the application program, open a trace window in CodeWarrior. When you execute the application program, the requested real-time trace run will be done, COF events will be recorded in the trace buffer until it is full, and later, when breakpoint C is reached, the program will halt. The results of the trace will then be displayed in the trace window of CodeWarrior.

## SpSub Sub-routine

- Position-independent routine – so it can be copied from FLASH into RAM (stack)
- On entry the address and command code are on the stack above the relocated SpSub routine

```
SpSub:    ldhx <SpSubSize+4,sp ;get flash address from stack
          sta ,x               ;write to flash; latch addr and data
          lda SpSubSize+3,sp ;get flash command
          sta FCMD             ;write the flash command
          lda #mFCBEF          ;mask to initiate command
          sta FSTAT            ;[pwpp] register command
          nop                  ;[p] want min 4~ from w cycle to r
ChkDone:  lda FSTAT             ;[prpp] so FCCF is valid
          lsla                 ;FCCF now in MSB
          bpl ChkDone          ;loop if FCCF = 0
SpSubEnd: rts                  ;back into DoOnStack in flash
SpSubSize: equ (*-SpSub)
```

Here you can see the SpSub sub-routine, which is a position-independent routine. The SpSub sub-routine is copied from FLASH into RAM and then executed. Note that this routine will still operate regardless of the absolute starting address. On entry, the data to be programmed is in A and the address and command code are on the stack above the relocated SpSub routine.

Our trace run will capture the JSR that takes us to the SpSub routine and the flow of instructions within SpSub until the capture buffer gets full. This would allow a user to examine the flow of this routine during development and debug.

## Source Code and Trace Result

Source	Trace	Address	Data	Instruction	PTM analyse remark
D:\Profiles\ltp70\My Documents\ApNotes\AN2141	Frame				
lda SpSubSize+6,sp ;	0	FF90	9E	LDA 30,SP	
jsr ,X	1	FF91	D6		
cli ;ok	2	FF92	00		
bra I_cont ;con	3	FF93	1E		
AI_set: lda SpSubSize+6,sp ;	4	FF94	FD	JSR ,X	
jsr ,X	5	0FE8	9E	LDHX 26,SP	Instruction outside application, DBG FIFO
ais #SpSubSize+3 ;d	6	0FE9	7E		Instruction outside application
H:X	7	0FEA	1C		Instruction outside application
lsla ;A=0	8	0FEB	F7	STA ,X	Instruction outside application
rts ;to	9	0FEC	9E	LDA 27,SP	Instruction outside application
	10	0FED	D6		Instruction outside application
	11	0FEE	00		Instruction outside application
	12	0FEF	1B		Instruction outside application
SpSub - This variation of SpSub	13	0FF0	C7		Instruction outside application
programming or erasing flash fr	14	0FF1	18	STA 0x1826	Instruction outside application
stack, SP is copied to H:X, and	15	0FF2	26		Instruction outside application
called using a JSR 0,X instruct	16	0FF3	A6	LDA #0x80	Instruction outside application
	17	0FF4	80		Instruction outside application
At the time SpSub is called, th	18	0FF5	C7	STA 0x1825	Instruction outside application
for an erase command), is in A	19	0FF6	18		Instruction outside application
stack above SpSub. After return	20	0FF7	3E		Instruction outside application
6 and 5 of A. If A is shifted l	21	0FF8	9D	NOP	Instruction outside application
should be zero unless there was	22	0FF9	C0	LDA 0x1825	Instruction outside application
	23	0FFA	18		Instruction outside application
Uses 24 bytes on stack + 2 byte	24	0FFB	25		Instruction outside application
SpSub: ldhx <SpSubSize+4,sp	25	0FFC	48	LSLA	Instruction outside application
sta ,X ;	26	0FFD	2A	BPL *-4	Instruction outside application, DBG FIFO
lda SpSubSize+3,sp ;	27	0FFE	FA		Instruction outside application
sta FCMD ;wxi	28	0FF9	C6	LDA 0x1825	Instruction outside application
lda #aFCBEF ;mas	29	0FFA	18		Instruction outside application
sta FSTAT ;[pw	30	0FFE	25		Instruction outside application
nop ;[p]	31	0FFC	48	LSLA	Instruction outside application
lda FSTAT ;[p]	32	0FFD	2A	BPL *-4	Instruction outside application, DBG FIFO
lsla ;FC	33	0FFE	FA		Instruction outside application
bpl ChkDone ;lo	34	0FF9	C6	LDA 0x1825	Instruction outside application
SpSubEnd: rts ;bac					

Here, you can see a screen shot of the results of our trace run. The source window is shown here, partially covered by the trace window, which is in the foreground to the right.

The leftmost column in the trace window shows a frame number starting with zero at the top of the screen. CodeWarrior knows the LDA instruction at frame zero was executed because it is located at the trigger point. It also knows the JSR was executed as well because the LDA does not cause a change-of-flow.

The JSR ,X uses the indexed addressing mode so the debugger cannot reconstruct the destination address because it does not know what was in H:X at that time. The on-chip ICE recognizes this COF and captures the destination of the jump in the capture buffer. You can see "DBG FIFO" in the rightmost column of the trace window at the next line below the JSR, X line. This is the first entry that was recorded into the capture buffer for this trace run.

The rest of the lines in the trace buffer correspond to execution of the SpSub routine on the stack. These lines are marked "Instruction outside application". If you look at the address in the second column you see 0FF8, which is the address of a stack RAM memory location. This address is indeed outside the area that CodeWarrior recognizes as our application program, which is in the FLASH memory. Usually this would indicate an error, but in our case, we are intentionally copying SpSub to RAM and executing it there.

Notice that 16 instructions are shown in the 34 frames or lines of the trace window that you can see. However, only three of these lines correspond to changes-of-flow that were recorded in the capture buffer. This demonstrates that although the capture buffer in the HCS08 is only eight words deep, the host debugger can reconstruct dozens of instructions from the captured information.

## Question

Do you know the difference between tag- and force-type breakpoints in the on-chip ICE system? Drag the boxes on the left to the appropriate statements on the right. Each letter may be used more than once. Click "Done" when you are finished.

**A** Tag-type breakpoints

**B** Force-type breakpoints

**B** These breakpoints act as simple interrupt requests when the address of interest is detected.

**B** These breakpoints are the most straightforward and are used to detect a memory access to a specific address.

**A** These breakpoints are used to detect instruction execution.

**A** With this type of breakpoint, instruction opcodes are marked as they are fetched.

Done

Reset

Show  
Solution

Let's take a moment to review tag versus force breakpoints.

Correct.

Tag-type breakpoints are used to detect instruction execution. Tag-type breakpoints also take into account the instruction queue, where instruction opcodes are marked as they are fetched a few cycles before being executed by the CPU. Force-type breakpoints are the most straightforward and are used to detect a memory access to a specific address. Force-type breakpoints also act as simple interrupt requests when the address of interest is detected. They cause a trigger or breakpoint at the next instruction boundary after the address compare is true. Click the forward arrow to continue on to the next page.

## BDM and On-Chip ICE References

- HCS08 Family Reference Manual -  
[http://www.freescale.com/files/microcontrollers/doc/ref\\_manual/HCS08RMV1.pdf](http://www.freescale.com/files/microcontrollers/doc/ref_manual/HCS08RMV1.pdf)  
(see section 7 Development Support)
- Application Note AN2140 -  
[http://www.freescale.com/files/microcontrollers/doc/app\\_note/AN2140.pdf](http://www.freescale.com/files/microcontrollers/doc/app_note/AN2140.pdf)
- Application Note AN2596 -  
[http://www.freescale.com/files/microcontrollers/doc/app\\_note/AN2596.pdf](http://www.freescale.com/files/microcontrollers/doc/app_note/AN2596.pdf)

Here are the BDM and on-chip ICE references that were previously mentioned in this course. For more detailed information about the HCS08 Family Reference Manual, visit the Freescale Web site shown here.

To see the application note AN2140 that describes the FLASH programming programs that were used for the debugging example, visit the Freescale Web site shown here.

To see the application note AN2596 that provides a good step-by-step tutorial on how to get started using the on-chip ICE with the CodeWarrior debugger, visit the Freescale Web site shown here.



## Course Summary

- Single-wire BDM
- BDM to read or modify on-chip memory or CPU registers, set breakpoints, and single-step through an application program
- BDM to program on-chip FLASH memory
- On-chip ICE system: features, benefits, registers
- On-chip ICE system to debug an application

This concludes the training course for the background debug system and the on-chip real time ICE system in the HCS08. In this course you examined the operation of the single-wire BDM. You looked at how BDM can be used to read or modify on-chip memory or CPU registers, set breakpoints, and single-step through an application program, as well as how BDM can be used to program on-chip FLASH memory. You explored the operation of the on-chip ICE system—looking at the features, benefits, and its various registers. Finally, you looked at how the on-chip ICE system can be used to debug an application.