LAUTERBACH
DEVELOPMENT TOOLS

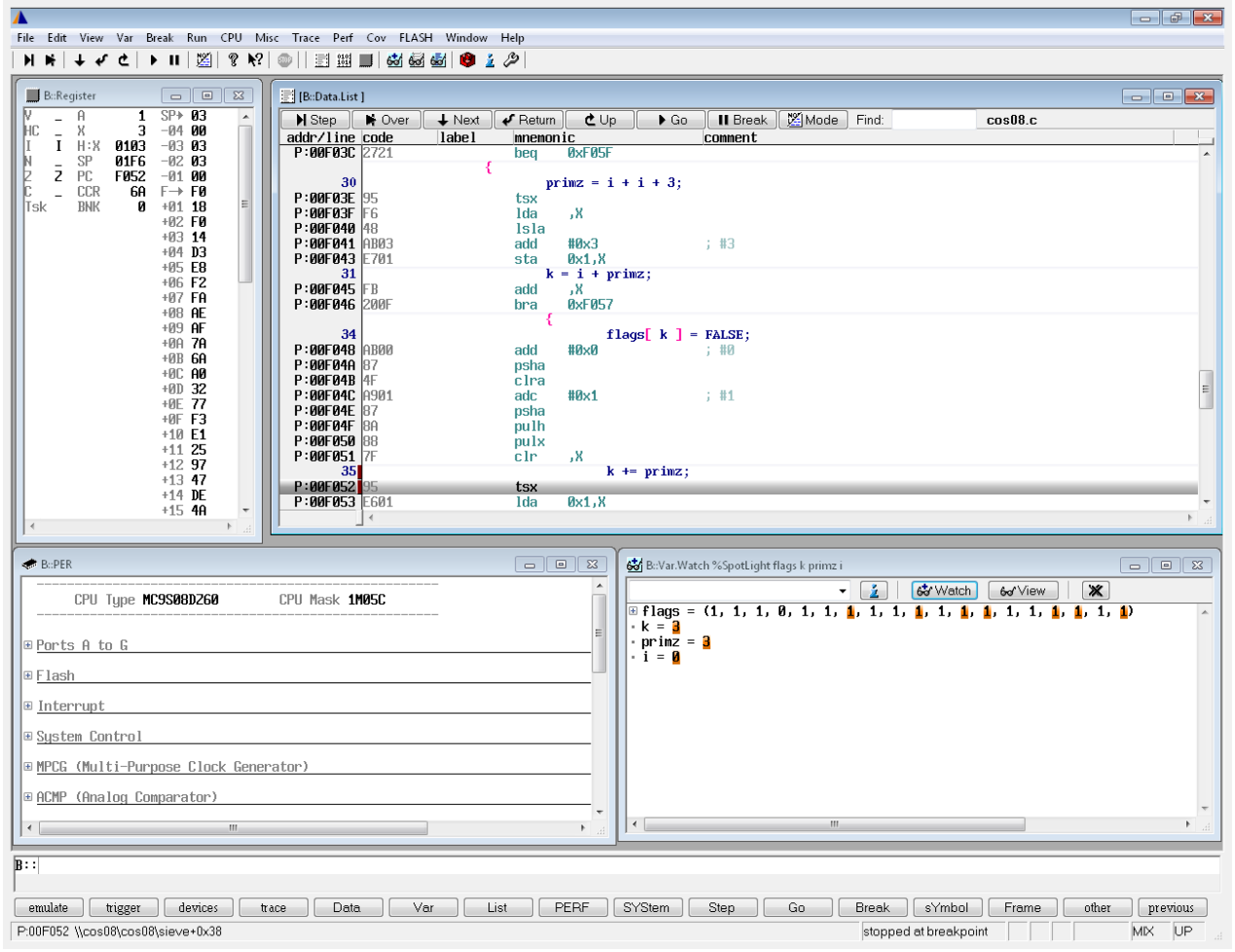
MCS08 Debugger

MANUAL

MCS08 Debugger

MCS08 Debugger

Version 06-Jun-2024



Brief Overview of Documents for New Users

Architecture-independent information:

- **"Training Basic Debugging"** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"T32Start"** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **"General Commands"** (general_ref_*<x>*.pdf): Alphabetic list of debug commands.

Architecture-specific information:

- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your debug cable. To access the manual for your processor architecture, proceed as follows:

  - Choose **Help** menu > **Processor Architecture Manual**.

- **"OS Awareness Manuals"** (rtos_*<os>*.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

WARNING:	To prevent debugger and target from damage it is recommended to connect or disconnect the Debug Cable only while the target power is OFF. Recommendation for the software start: 1. Disconnect the Debug Cable from the target while the target power is off. 2. Connect the host system, the TRACE32 hardware and the Debug Cable. 3. Power ON the TRACE32 hardware. 4. Start the TRACE32 software to load the debugger firmware. 5. Connect the Debug Cable to the target. 6. Switch the target power ON. 7. Configure your debugger e.g. via a start-up script. Power down: 1. Switch off the target power. 2. Disconnect the Debug Cable from the target. 3. Close the TRACE32 software. 4. Power OFF the TRACE32 hardware.

Troubleshooting

SYStem.Up Errors

The **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages while executing this command this may have the following reasons.

- The target has no power.
- The target is in reset - another device may hold the reset line active.
- There is a short circuit on at least one of the output lines of the core.
- There is a problem with the electrical connection between ICD08 and the target - check if the BDM connector is plugged correctly and if the target is built corresponding to the **definition** of the used BDM connector.
- The MC9S08 has no Clock - check the frequency on the EXTAL pin with a scope.

FAQ

Please refer to https://support.lauterbach.com/kb.

CPU Specific Implementations

Breakpoints

There are two types of breakpoints available: Software breakpoints and on-chip breakpoints.

Software Breakpoints

Software breakpoints are the default breakpoints for program breakpoints. A software breakpoint is implemented by patching a break code into the memory.

There is no restriction in the number of software breakpoints.

On-chip Breakpoints

The resources for the on-chip breakpoints are provided by the CPU.

The following list gives an overview of the supported on-chip breakpoints:

- **On-chip breakpoints:** Total amount of available on-chip breakpoints.
- **Instruction breakpoints:** Number of on-chip breakpoints that can be used to set Program breakpoints into ROM/FLASH/EEPROM.
- **Read/Write breakpoints:** Number of on-chip breakpoints that can be used as Read or Write breakpoints.
- **Data breakpoint:** Number of on-chip data breakpoints that can be used to stop the program when a specific data value is written to an address or when a specific data value is read from an address.

	Onchip Breakpoints	Program Breakpoints	Read/Write Breakpoints	Data Value Breakpoints
MCS8	2	up to 2 single address	up to 2 single address (reduced to 1 if combined with data)	1

Quick Start of the ICD Debugger for HC9S08

This chapter should help you to prepare your Debugger for HC9S08. Depending on your application not all steps might be necessary.

For some applications additional steps might be necessary, that are not described in this Quick Start section.

1. Prepare the Start

Connect the Debug Cable to your target. Check the orientation of the connector. Pin 1 of the debug cable is marked with a small triangle next to the nose of the target **connector.**

It is not necessary to connect the Clock Cable for the first start.

Power up your TRACE32 system (This is not necessary on PODPC).

Start the TRACE32 Debugger Software.

Power up your Target!

	To prevent damage please take care on this sequence all the time you are preparing a start.

2. Select the Clock for the BDM Communication

The MC9S08 has the capability to do a automatic synchronization between CPU and debugger. This synchronization is done when you switch the SYStem.Mode to **Up**, **Attach** or **Go**.The resulting communication clock is shown in the area called BdmClock within the **SYStem** window.

The selection box for **SYStem.BdmClock** can remain on the default setting. This option may be helpful to increase download rate, but this is not important for the first debug session.

3. Configure the Debugger According to the Needs of the Application

Most of the configuration can be done with the ***SYStem*** Window which provides all CPU specific settings. Use ***System Settings...*** in the ***CPU*** menu to open this window.

The debugger will select t the CPU type automatically if the pull down menu in the field ***CPU*** shows AUTO. If this does not work, please refer to the command: **SYStem.CPU** *<cpu_type>*.

```
SYStem.CPU AUTO
```

Set the **SYStem Options** in the option field corresponding to your target configuration and application program. Generally the SYStem Options can remain at the default values for the first start.

4. Map the EPROM Simulator if Available (optional)

```
MAP.ROM 0x0--0x1FFFF
```

This command maps a standard 8 bit wide 27x010 EPROM.

5. Tell the Debugger Where it should use On-chip Breakpoints (optional)

By default the In Circuit Debugger for 68HC12 (ICD12) modifies the code to realize a program breakpoint. This will not work for ROM or FLASH memory locations. To provide breakpoints in ROM/FLASH areas the CPU' s on-chip breakpoints can be used. With the command MAP.BOnchip *<range>* you can specify where the debugger has to use the on-chip breakpoints.

```
MAP.BOnchip 0x1000--0x0ffff      ; activates the on-chip breakpoints
                                 ; within the range from 0x1000 to 0xffff
```

6. Enter Debug Mode

```
SYStem.Up
```

This command triggers a reset for the CPU and drives the line BKGD to GND. So the CPU will enter the “special” variant of operating mode defined by the pins MODA and MODB, which have to be configured by the target.

LAUTERBACH recommends to use Single Chip Mode for starting from reset. In this case the CPU will enter Special Single Chip Mode without executing any code. So all registers will contain reset values. In all other cases the CPU will try to execute code after reset, until the debugger gets control on it. So some registers may already be in use.

Some derivatives need a Power On Reset to enter debug mode out of reset without executing code. Dependent from their actual memory contents these derivatives might stuck in a reset loop. When you enter SYStem.Up the debugger tries to get the CPU out of such a loop, but if the time between two resets is too small this attempt might fail. Use **SYStem.Mode StandBy** in this case, remove power from the target and switch it on again.

7. Load the Program

Generally HC9S08 code will be located to internal FLASH memory. With help of the CPU specific menu “FLASH” you can erase the CPU’s FLASH memory and prepare it for code loading. For details please refer to the chapter **FLASH EEPROM Management**.

When the CPU is prepared the code can be loaded. This can be done with the command **Data.Load.<file_format> <file>**. Applications can be loaded by various file formats. The format depends from the compiler. Here are some typical load commands for HC9S08 applications:

```
Data.Load.COSMIC <file>.cos12       ; load application file generated
                                    ; with a COSMIC compiler

Data.Load.Elf <file>.abs /verify    ; load application file generated
                                    ; with a Metrowerks compiler and
                                    ; verify if it is written correct to
                                    ; memory
```

8. Initialize Program Counter and Stackpointer

Many compilers add these settings in the start-up code to the user program automatically. In this case no action is necessary. You can check the contents of Program Counter and Stack Pointer in the ***Register*** Window which provides the contents of all CPU Registers. Use ***CPU Registers*** in the ***CPU*** menu to open this window.

The Program Counter and the Stackpointer and all other registers can be set with the commands **Register.Set PC** *<value>* and **Register.Set SP** *<value>*. Here is an example of how to use these commands:

```
Register.Set PC 0c000          ; Set the Program Counter to address
                               ; $C000

Register.Set SP 0bff           ; Set the Stack Pointer to address
                               ; $bff

Register.Set PC main           ; Set the PC to a label (here: function
                               ; main)
```

9. View the Source Code

Use the command **Data.List** to view the source code at the location of the Program Counter.

Now the quick start is done. If you were successful you can start to debug. Lauterbach recommends to prepare a PRACTICE script file (*.cmm, ASCII format) to be able to do all the necessary actions with only one command. Here is a typical start sequence:

```
B::                            ; Select the ICD device prompt

WinClear                       ; Clear all windows

SYStem.Reset                   ; Bring all settings in the SYStem window
                               ; to default value

MAP.BOnchip 0x01080--0x0ffff   ; Select on-chip breakpoints for the
                               ; FLASH EEPROM area

SYStem.Up                      ; Reset the target and enter special mode

Data.LOAD.Elf hic.abs          ; Load the application - here an absolute
                               ; file in ELF/DWARF format

Register.Set PC main           ; Set the PC to function main

Register.Set SP 0xFFF          ; Set the stack pointer to address $0xFFF

List.Mix                       ; Open disassembly window    *)

Register.view                  ; Open register window       *)
```

```
Frame.view /Locals /Caller      ; Open the stack frame with
                                ; local variables           *)

Break.Set 0x400 /p              ; Set software breakpoint to address 100
                                ; (address 0x400 is outside the range,
                                ; where on-chip breakpoints are used)

Break.0xSet 0x8024 /p           ; Set on-chip program breakpoint to
                                ; address 0x8024 (address 0x8024 is
                                ; within the range, where on-chip
                                ; breakpoints are used)
```

*) These commands open windows on the screen. The window position can be specified with the **WinPOS** command.

You can find suggestions for such PRACTICE script files (*.cmm) in the TRACE32 demo folder **~~/demo/m68hc08/compiler**.

For information about how to create a PRACTICE script file (*.cmm file), refer to **"Training Basic Debugging"** (training_debugger.pdf). There you can also find some information on basic actions with the debugger.

Please keep in mind that only the Processor Architecture Manual (the document you are reading in at the moment) is CPU specific, while all other parts of the online help are generic for all CPUs. So if there are questions related to the CPU, the Processor Architecture Manual should be your first choice.

Restrictions

External watchdog	With the debugger in break mode an external watchdog will not be triggered and so it will do its job and drive the system to reset. External watchdogs must be disabled.

SYStem.BdmClock Select clock for BDM communication

Format: **SYStem.BdmClock BusClock** | **Fixed**

The MC9S08 offers two clock sources for the communication between debugger an CPU: BusClock and Fixed.

BusClock Time base for BDM communication is the bus frequency of the CPU. This allows a faster download if the data rate is increased by activating the PLL. On the other hand moving the communication frequency can cause problems, because the debugger has to synchronize again after each change of frequency.

Fixed The fixed internal clock of the CPU is time base for BDM communication. The bus frequency can be modified without impact to the BDM. This selection requires no resynchronization on bus clock changes.

This command tells the debugger how to configure the CPU when you start your debug session. In general “Fixed” is the selection of choice. If you have to download big files you can use “BusClock” and activate the CPU’s PLL to get the highest available bus frequency and as a result the highest download performance. LAUTERBACH recommends to drive the debugger to reset again after download and to start the following debug session with the selection “Fixed”.

TRIMS08FLL(<*frequency_in_KHz*>**)**

TRIMS08FLL() sets the registers ICSSC and ICSTRM (ICS_C3 and ICS_C4) to achieve the requested frequency and returns the reached frequency in KHz. The debugger does this job using the SYNC command of the background debug controller (BDC). So the frequency you request is compared to ICSBDCCLK (BdmClock) while CLKSW is set to zero (use alternate BDC clock source).

NOTE:	The alternate BDC clock is often connected to the FLL through a divider by two. So if you use the following commands, as shown in the PRACTICE code snippet below, you will trim the FLL to 20 MHz:

```
LOCAL &freq
&freq=trims08fll(10000.)
```

SYStem.CONFIG Configure debugger according to target topology

The **SYSTem.CONFIG** command group is not supported.

Format:	**SYStem.CPU** <*type*>
<*type*>:	**AUTO** **MC9S08GB32** | **MC9S08GB60** | **MC9S08GT32** | **MC9S08GT60**

With this command the processor type is selected. The MC9S08 has internal device and mask specific registers which allow the debugger to select the CPU type automatically. This is done for all known versions if the pull down menu in the field ***CPU*** shows AUTO. For new mask revisions or devices you can select the CPU manually with this command.

NOTE:	If you type **SYStem.CPU** to the command line followed by blank, the softkeys (below the TRACE32 command line) provide you with all supported derivatives.

SYStem.LOCK Lock and tristate the debug port

Format:	**SYStem.LOCK** [**ON** | **OFF**]

Default: OFF.

If the system is locked, no access to the debug port will be performed by the debugger. While locked, the debug connector of the debugger is tristated. The main intention of the **SYStem.LOCK** command is to give debug access to another tool.

Format: **SYStem.MemAccess Enable | StopAndGo | Denied**
SYStem.ACCESS (deprecated)

Enable Memory access during program execution to target is enabled.
CPU (deprecated)

Denied (default) Memory access during program execution to target is disabled.

StopAndGo Temporarily halts the core(s) to perform the memory access. Each stop takes some time depending on the speed of the JTAG port, the number of the assigned cores, and the operations that should be performed.

SYStem.Mode

Select target reset mode

Format: **SYStem.Mode** <*mode*>

SYStem.Down (alias for SYStem.Mode Down)
SYStem.Attach (alias for SYStem.Mode Attach)
SYStem.Up (alias for SYStem.Mode Up)

<*mode*>: **Down**
NoDebug
Go
Attach
StandBy
Up

Down Drives a pulse to the reset line and disables the Debugger.

NoDebug Disables the Debugger. The state of the CPU remains unchanged.

Go Resets the target with debug mode enabled and prepares the CPU for debug mode entry. After this command the CPU is in the system.up mode and running. Now, the processor can be stopped with the break command or until any break condition occurs.

Attach

This command works similar to the **SYStem.Mode Up** command. The difference is that the target CPU is not reset. The BDM/JTAG/COP interface will be synchronized and the CPU state will be read out. After this command the CPU is in the system.up mode and can be stopped and debugged.

StandBy

The debugger drives the BKGD line to GND and waits for power up. The debugger connects to the CPU when target voltage is sensed and reset is released. This sequence is repeated again if power returns after a power fail.

Different to other debuggers SYStem.Mode StandBy is used here to have a save way to get control on the CPU before it can execute any code.

Up

Resets the target and sets the CPU to debug mode. After execution of this command the CPU is stopped and prepared for debugging. All register are set to the default value.

Format: **SYStem.Option.IMASKASM** [**ON** | **OFF**]

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during assembler single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step.

SYStem.Option.IMASKHLL Disable interrupts while HLL single stepping

Format: **SYStem.Option.IMASKHLL** [**ON** | **OFF**]

Default: OFF.

If enabled, the interrupt mask bits of the CPU will be set during HLL single-step operations. The interrupt routine is not executed during single-step operations. After single step the interrupt mask bits are restored to the value before the step.

Hardware Breakpoints

Most MC9S08 derivatives have three Hardware breakpoints, one in the BDM module (only address compare) and two in the debug module. Refer to the *Technical Summary* of your derivative to check if Hardware Breakpoints are available.

Program Breakpoints

Generally the In Circuit Debugger for MC9S08 uses Software Breakpoints to realize Program Breakpoints. Software Breakpoint means that the code at the desired memory location is modified by the debugger to make the CPU break when the program counter meets this address. After this the original contents of this memory location is restored.

This mechanism can not work in Read Only Memory. To provide breakpoints in ROM areas the CPU' s Hardware Breakpoints can be used. The memory ranges. where Hardware Breakpoints should be used must be defined with the command **MAP.BOnchip**.

```
MAP.BOnchip0x1080--0x0ffff
```

With the command **Break.List** the actual breakpoint configuration can be checked.

Read and Write Breakpoints

Read and Write Breakpoints always use the CPU' s Hardware Breakpoints regardless off the ranges defined with **MAP.BOnchip**.

Read and Write Breakpoints can be set with the command **Break.Set…**

```
Break.Set 0x4738 /w           ; The CPU will be stopped if there is a
                              ; write access to address $4738

b.s 0x0b223 /r                ; The CPU will be stopped if there is a
                              ; read access to address $B223
```

Data Breakpoints

Data Breakpoints always use the CPU' s Hardware Breakpoints regardless off the ranges defined with **MAP.BOnchip**. To provide a Breakpoint on address, data and cycle type match and data match two 16 bit registers are needed. So there is only one breakpoint of this type available. Use the command Break.Set or the "Set…" entry in the Break menu to open the control window for the Breakpoints.

```
Break.Set 0x421 /READ /Data.Byte 0y11x10001     Emulation will be stopped if the
                                                CPU does a read access to
                                                address $421, while this memory
                                                location contains the value 0xD1
                                                or 0xF1.
```

Onchip.Mode.EventTrace — Start recording after trigger event

Format: **Onchip.Mode.EventTrace**

Recording starts after trigger event.

Onchip.Mode.FlowTrace — Flow trace mode

Format: **Onchip.Mode.FlowTrace**

Flow trace mode.

Onchip.Mode.LoopTrace — Inhibit redundant entries

Format: **Onchip.Mode.LoopTrace**

Flow trace inhibiting redundant entries to capture memory.

TrOnchip.Mode — Select trace and trigger mode

Format: **TrOnchio.Mode** <*mode*>

<*mode*>:
BreakAORB
BreakATHENB
TraceAORB
TraceATHENB
OFF

BreakAORB — Real-time execution is stopped if one of the two Comparators A or B matches.

BreakATHENB — Real-time execution is stopped if as a first step the comparators A matzohs and then as a second step the comparator B matches.

TraceAORB — Recording cycles to the on-chip trace is stopped if one of the two Comparators A or B matches.

TraceATHENB — Recording cycles to the on-chip trace is stopped if as a first step the comparators A matzohs and then as a second step the comparator B matches.

OFF — Complex on-chip debug features are disabled.

The 9S08 has an on-chip debug module which offers some complex trigger features and a small on-chip trace. Please refer to your CPU' s manual for details. These debug features are based on two address comparators which can be specified in the following way:

```
Break.Set 0x8000 /Alpha /Read    ; Set the Debug Comparator A to match on
                                 ; a read access from the address 0x8000.

Break.Set 0x8010 /Beta /Write    ; Set the Debug Comparator B to match on
                                 ; a write access from the address
                                 ; 0x8100.
```

Note that it is a must to specify whether the access is read or write.

TrOnchip.state — Open the control window for the on-chip trigger resources

Format: **TrOnchip.state**

Opens the control window for the on-chip trigger resources.

TrOnchip.RESet — Reset the on-chip trigger resources

Format: **TrOnchip.RESet**

Sets all controls in the trigger on-chip window to default settings.

Memory	Description
P:	Program
D:	Data
E:	Hidden access to CPU memory while foreground emulation is active (“Emulation memory access” is the original of this name)

The 9S1208 does not separate program and data memory, so “P:” and “D:” share the same memory. “E:” is used to access any memory access, even internal special function registers, while the CPU is executing user’s program.

FLASH EEPROM Management

Flash programming on the HC9S08 is based on a mechanism called “Target Controlled Flash Programming”. The file ~~/demo/m68hc08/flash/byte/mcs08.bin contains the programming code. It is loaded and executed within the internal RAM of the CPU when you enter FLASH commands. Several settings are necessary to do the link between TRACE32 software and mcs08.bin.

For convenience all this is can be done with the PRACTICE script file **~~/demo/m68hc08/flash/flashs08.cmm**. This PRACTICE script file can be called with several parameters:

Parameter	Description
prepare	Creates all entries to make the **FLASH** commands deal with mcs08.bin.
erase	Prepare FLASH and erase all FLASH cells.
prog	Prepare FLASH, activate FLASH programming, open a window to select a file and load it down to the CPU.
eraprog	Prepare FLASH, erase all FLASH cells, activate FLASH programming, open a window to select a file and load it down to the CPU.
secure	Open a dialog to specify security level and secure the CPU.
unsecure	Erase all FLASH cells and unsecure CPU. Before the erase a warning has to be accepted.

With help of the menu FLASH you can call flashs08.cmm with each of these parameters by mouse. If you can not find the menu FLASH, check for the file mens08.men in your TRACE32 directory. If it is there use menu.rp mens08.men to add it.

LAUTERBACH recommends to call flashs08.cmm with the needed parameters if you want to do FLASH programming from your own PRACTICE file. The benefit of doing so is that you can keep your script file when changes relating the flash programming have to be done, because these changes will be done by updating flashs08.cmm. The file ~~/demo/m68hc08/flash/flashexamples08.cmm is an example for FLASH programming by means of a PRACTICE script file (*.cmm).

To be able to debug within FLASH EEPROM areas the CPU’ s Hardware Breakpoints must be activated. Refer to chapter **Using Hardware Breakpoints**.

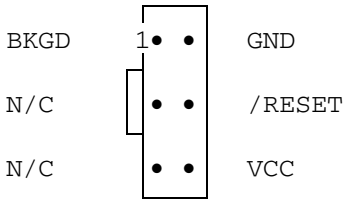
Secure and Unsecure

To avoid illegal copies of code he MC9S08 offers a security feature. Please refer to the CPU's *technical summary* for the technical background. There are three cases where this feature is important at debug sessions.

1. Development reached a level where a prototype is given out. In this case you can secure the chip with help of the item ***Secure CPU*** in the **FLASH** menu. Follow the dialog to do the necessary configuration. You can specify a back door key to get access to the secured chip. To use this back door key your application code has to contain code to open the chip. This can not be done by the debugger. However you can use the debugger to make your code opening the chip. An idea is to specify a variable for the key and another one to start the opening routine. Enter the key to this variable by the debugger, set the start variable and execute your code.

It is not possible to unsecure a chip without erasing the whole FLASH with help of the debugger.

2. You receive a secured chip and want to start debugging with this. If the code on the chip does not support back door key, you have to erase all FLASH cells and to reprogram the chip. Use the item ***Unsecure CPU*** in the **FLASH** menu.



This image shows the top view to the male connector on the target board.