Freescale 68HC08 Family In-Circuit Emulation

Contents

1 Introduction

Debug Features

- Unlimited breakpoints
- Access breakpoints
- Real-time access
- Trace
- Execution profiler
- Execution coverage

1.1 Differences from a standard environment

The In-Circuit Emulator emulates the target CPU, which is removed from the target, as good as possible. Beside the CPU, additional logic is integrated on the POD. The amount of additional logic depends on the emulated CPU and the type of emulation. A buffer on a data bus is always used (minimal logic) and when rebuilding ports on the POD, maximum logic is used. As soon as the POD is connected to the target instead of the CPU, electrical and timing characteristics are changed. Different electrical and timing characteristics of used elements on the POD and prolonged lines from the target to the CPU on the POD contribute to different target (the whole system) characteristics. Consequentially, in worst case signal cross-talks and reflections can occur due to bad target connection, capacitance changes, etc.

Beside that, pull-up and pull-down resistors are added to some signals. Pull-up/pull-down resistors are required to define the inactive state of signals like reset and interrupt inputs, while the POD is not connected to the target. Because of this, the POD can operate as standalone without the target.

1.2 Common Guidelines

Here are some general guidelines that you should follow.

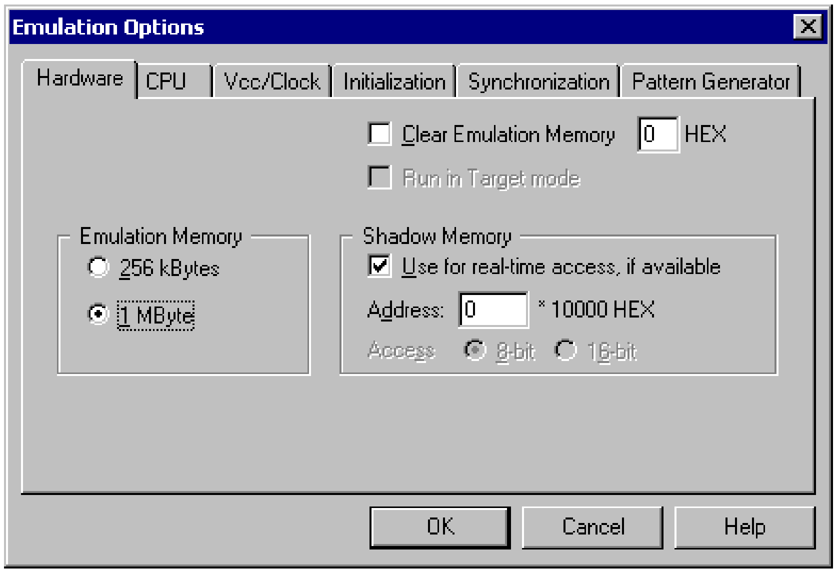
- Use external (target) Vcc if possible (to prevent GND bouncing),
- Make an additional GND connection from the POD to the target if the Emulator behaves strangely,
- Use the reset output line on the POD to reset the target whenever Emulator resets the CPU,
- Make sure the appropriate CPU is used on the POD. Refer to the POD Hardware reference for more details.
- When interrupts in background are enabled, take note that the interrupt routine must return in 25 ms, otherwise the Emulator will assume that the program is hung.

1.3 Port Replacement Information

In general, when emulating the single chip mode, some ports have to be rebuilt on the POD because original ports are used for emulation – typically ports used as address and data bus in extended mode. Special devices, so called port replacement units, provided already by the CPU vendor or other standard integrated circuits are used to rebuild "lost" ports. Rebuilt ports are logically compatible with original CPU's ports, but electrical characteristics may differ. If a special device (the port replacement unit (PRU), available from the CPU manufacturer) is available, electrical characteristics don't differ much and usually the user doesn't have to pay attention. The differences may become relevant when standard integrated circuits are used and operating close to electrical limits, e.g. when input voltage level is close to specified maximum voltage for low input level ("0") or specified minimum voltage for high input level ("1") or if, for example, the target is built in the way that the maximum port input current must be considered.

2 Emulation Options

2.1 Hardware Options



☐ Clear Emulation Memory 0 HEX

☐ Run in Target mode

Emulation Memory
- ○ 256 kBytes
- ● 1 MByte

Shadow Memory
- ☑ Use for real-time access, if available
- Address: 0 * 10000 HEX
- Access ● 8-bit ○ 16-bit

In-Circuit Emulator Options dialog, Hardware page

Emulation Memory

User must define the size of emulation memory available on the emulation module. The emulator will not initialize in case of wrong setting.

This setting applies for Power Emulator only.

Clear Emulation Memory

This option allows you to force clearing (with the specified value) of emulation memory after the emulation unit is initialized.

Clearing emulation memory takes about 2 seconds per megabyte, so use it only when you want to make sure that previous emulation memory contents don't affect the current debug session.

Shadow Memory

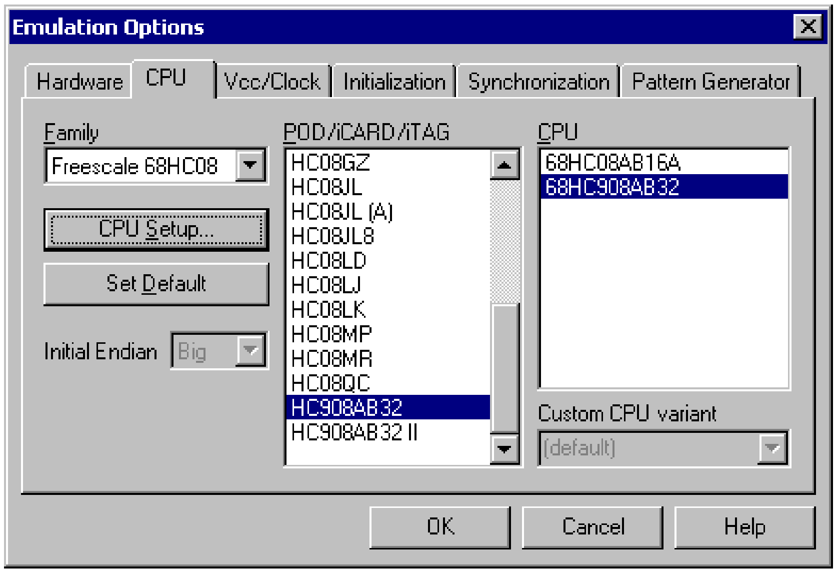
On-board shadow memory is provided that allows reading of memory without stopping the CPU or stalling it even for a single cycle. If you wish to use this memory for real-time access, check the 'Use for real-time access if available' option.

In case of Power Emulator, the 'Address' setting allows you to place a 256KB shadow memory block to the address where your program's data begin. Since HC08 family operates in 64KB memory space only, keep offset at default 0.

This setting applies for Power Emulator only. Other HC08 development systems implement real-time access using other emulator resources.

2.2 CPU Configuration

With In-Circuit emulation besides the CPU family and CPU type the emulation POD must be specified (some CPU's can be emulated with different PODs).



Family: Freescale 68HC08

Initial Endian: Big

POD/iCARD/iTAG:
HC08GZ
HC08JL
HC08JL (A)
HC08JL8
HC08LD
HC08LJ
HC08LK
HC08MP
HC08MR
HC08QC
HC908AB32
HC908AB32 II

CPU:
68HC08AB16A
68HC908AB32

Custom CPU variant: (default)

In-Circuit Emulator Options dialog, CPU Configuration page

CPU Setup

Opens the CPU Setup dialog. In this dialog, parameters like memory mapping, bank switching and advanced operation options are configured. The dialog will look different for each CPU reflecting the options available for it.

Set Default

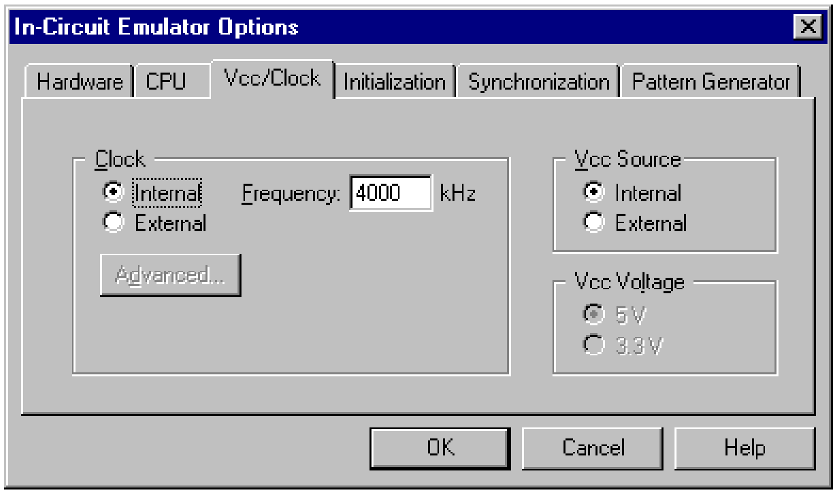
This button will set default options for currently selected CPU. These include:

- Vcc and clock source and frequency
- Advanced CPU specific options
- Memory configuration (debug areas, banks, memory mapping)

Note: Default options are also set when the Family or a POD is changed.

2.3 Power Source and Clock

The Vcc/Clock Setup page determines the CPU's power and clock source.



Clock: [✓ Internal] External — Frequency: 4000 kHz

Vcc Source: [✓ Internal] External

Vcc Voltage: [✓ 5 V] 3.3 V

In-Circuit Emulator Options dialog, Vcc/Clock Setup page

Note: When either of these settings is set to External, the corresponding line is routed directly to the CPU from the target system.

Clock Source

Clock source can be either used internal from the emulator or external from the target. It is recommended to use the internal clock when possible. When using the clock from the target, it may happen that the emulator cannot initialize any more.

It is dissuaded to use a crystal in the target as a clock source during the emulation. It is recommended that the oscillator be used instead. Normally, a crystal and two capacitors are connected to the CPU's clock inputs in the target application as stated in the CPU datasheets. A length of clock paths is critical and must be taken into consideration when designing the target. During the emulation, the distance between the crystal in the target and the CPU (on the POD) is furthermore increased, therefore the impedance may change in a manner that the crystal doesn't oscillate anymore. In such case, a standalone crystal circuit, oscillating already without the CPU must be built or an oscillator must be used.

When the clock source is set to Internal, the clock is provided by the emulator and its frequency can be set in steps of 1 kHz.

Note: The clock frequency is the frequency of the signal on the CPU's clock input pin. Any internal manipulation of it (division or multiplication) depends entirely on the emulated CPU.

If the clock source is set to external, the clock is provided by the target system. In certain applications, for instance, a 32.786kHz clock is used. Since the minimal clock the Emulator can generate is 1MHz, an external clock source must be used and the clock source set to external.

Vcc Source

This setting determines whether the emulator or the target system provides a power supply for the CPU.

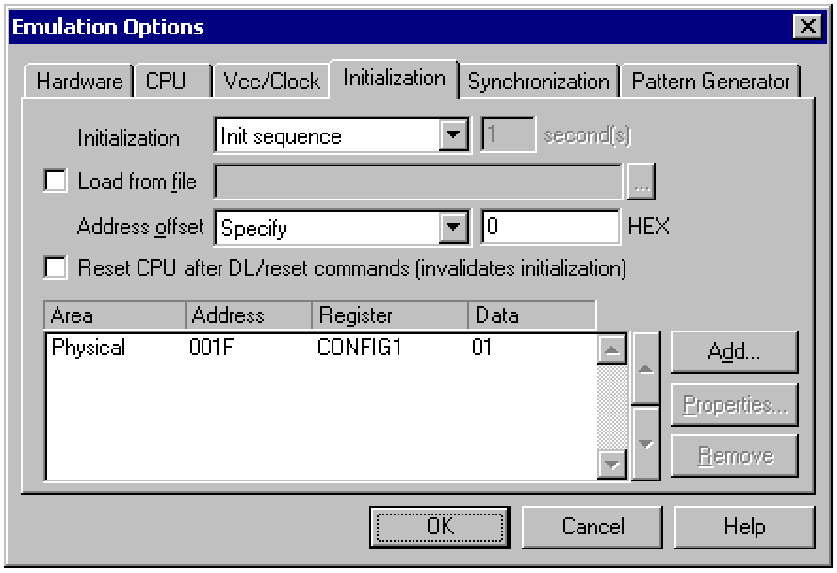
2.4 Initialization Sequence

Usually, there is no need to use initialization sequence when debugging with an In-Circuit Emulator (ICE) a single chip application. Primarily, initialization sequence is used on On-Chip Debug systems to initialize the CPU after reset to be able to download the code to the target (CPU or CPU external) memory. With an ICE system, the initialization sequence may be required for instance to enable memory access to the CPU internal EEPROM or to some external target memory, which is not accessible by default after the CPU reset. The user can also disable CPU internal COP using initialization sequence if there is a need for that, etc.

Initialization sequence is executed immediately after the CPU reset and then the code is downloaded.

The initialization sequence can be set up in two ways:

1. Set up the initialization sequence by adding necessary register writes directly in the Initialization page within winIDEA.



Initialization: Init sequence

☐ Load from file

Address offset: Specify 0 HEX

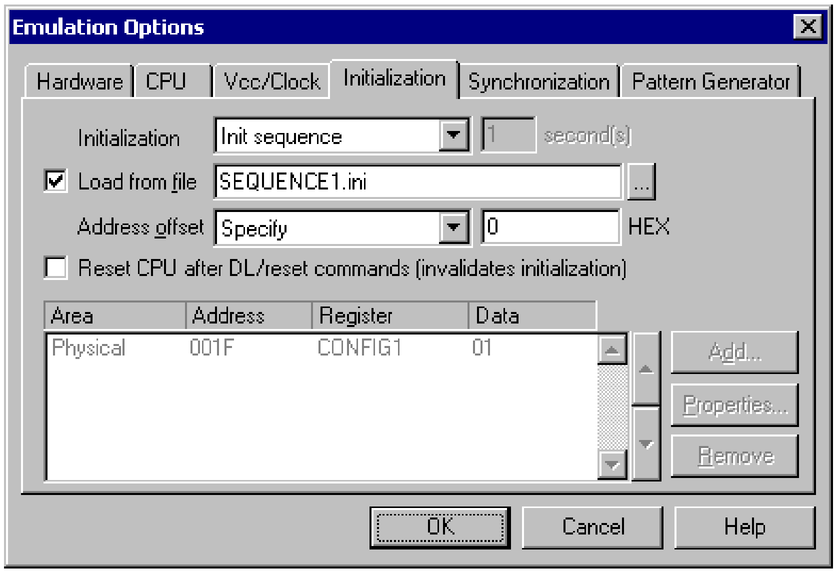
☐ Reset CPU after DL/reset commands (invalidates initialization)

Area	Address	Register	Data
Physical	001F	CONFIG1	01

2. winIDEA accepts initialization sequence as a text file with .ini extension. The file must be written according to the syntax specified in the appendix in the hardware user's guide.

   Excerpt from the sample SEQUENCE1.ini file:

```
S PTBD B 12         //comment
S PTBDD B FF
```


Emulation Options

Hardware | CPU | Vcc/Clock | Initialization | Synchronization | Pattern Generator

Initialization: Init sequence — 1 second(s)

☑ Load from file: SEQUENCE1.ini

Address offset: Specify — 0 HEX

☐ Reset CPU after DL/reset commands (invalidates initialization)

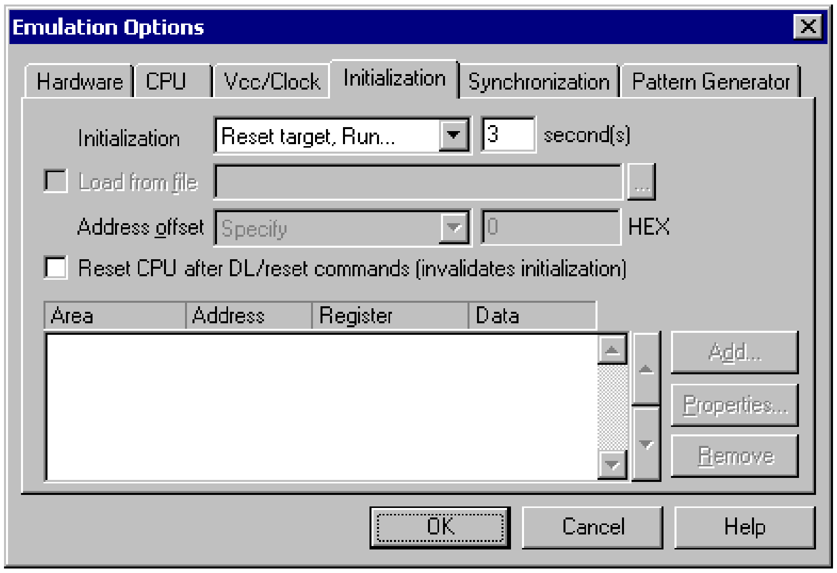
Area	Address	Register	Data
Physical	001F	CONFIG1	01

Add... Properties... Remove

OK Cancel Help

The advantage of the second method is that you can simply distribute your .ini file among different workspaces and users. Additionally, you can easily comment out some line while debugging the initialization sequence itself.

There is also a third method, which can be used too but it's not highly recommended for the start up. The user can initialize the CPU by executing part of the code in the target ROM for X seconds by using 'Reset and run for X sec' option.


Emulation Options

Hardware | CPU | Vcc/Clock | Initialization | Synchronization | Pattern Generator

Initialization: Reset target, Run... — 3 second(s)

☐ Load from file

Address offset: Specify — 0 HEX

☐ Reset CPU after DL/reset commands (invalidates initialization)

Area	Address	Register	Data

Add... Properties... Remove

OK Cancel Help

2.5 Pattern Generator

iC1000, iC2000 and iC4000 provide an 8-channel waveform programmable pattern generator capable of continuous or single shot operation at up to 10MHz-clock rate with up to 512 samples.

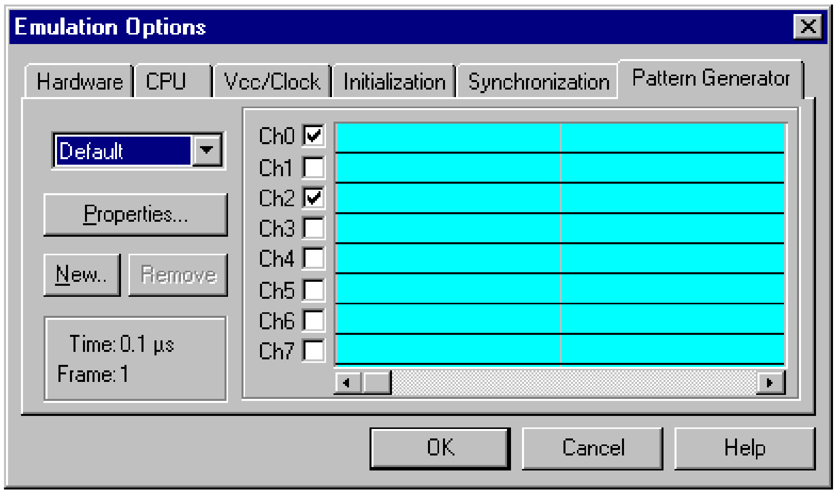
Note: when using iC4000 system, it has in certain configurations two Pattern Generators: one on the base module and one on the Power Emulator module. The Pattern Generator on the base module is active when the debugging type in the 'Hardware/Hardware/Hardware type' tab is set to 'Active' or 'On-Chip'. The Pattern Generator on the Power Emulator Module is active when 'In-circuit Emulation' is selected..

You can configure any number of patterns using 'New…' and 'Remove' buttons. The currently selected pattern is displayed in the combo box as indicated in the above figure.

State of a disabled channel can be configured either to high or low.

Every individual channel can be enabled or disabled by configuring the check box next to its name. When a channel is disabled you can still configure its state, which remains unchanged throughout its period.

Waveforms are configured easily by clicking and moving the mouse cursor on the desired channel and position.



Default

Ch0 ✓
Ch1
Ch2 ✓
Ch3
Ch4
Ch5
Ch6
Ch7

Time: 0.1 µs
Frame: 1

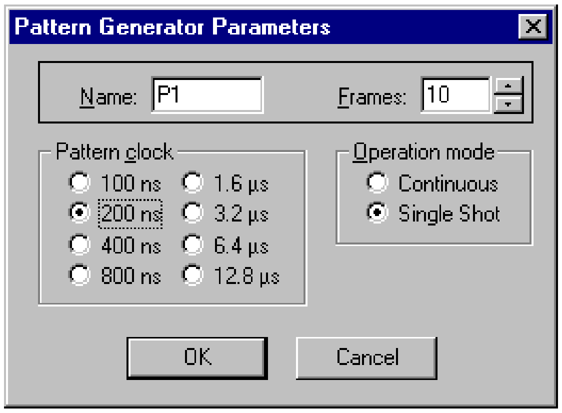
In-Circuit Emulator Options dialog, iC1000/iC2000/iC4000 Pattern Generator page

Properties

This button opens a dialog where parameters for the current pattern can be configured.

Pattern Generator Parameters

Parameters of a pattern are valid for all of its eight channels. This means that all channels are of the same length and all use the same clock.



Name: P1
Frames: 10

Pattern clock: 100 ns [✓ 200 ns] 400 ns 800 ns 1.6 µs 3.2 µs 6.4 µs 12.8 µs

Operation mode: Continuous [✓ Single Shot]

iC1000/iC2000/iC4000 Pattern Generator Parameters dialog

Name

Defines the name of the current pattern

Frames

Defines number of frames used in the pattern. Frames multiplied by pattern clock define the period of the pattern. The number of frames is limited to 512.

Pattern clock

Defines the clock rate by, which the waveform progresses.

Operation mode

Defines whether the pattern is to run continuously or to execute only a single shot on demand. In any case, pattern operation is controlled from the Hardware menu by selecting the 'Run Pattern' command.

When continuous mode is selected, the 'Run Pattern' command will either stop pattern execution (at the last frame), or resume it.

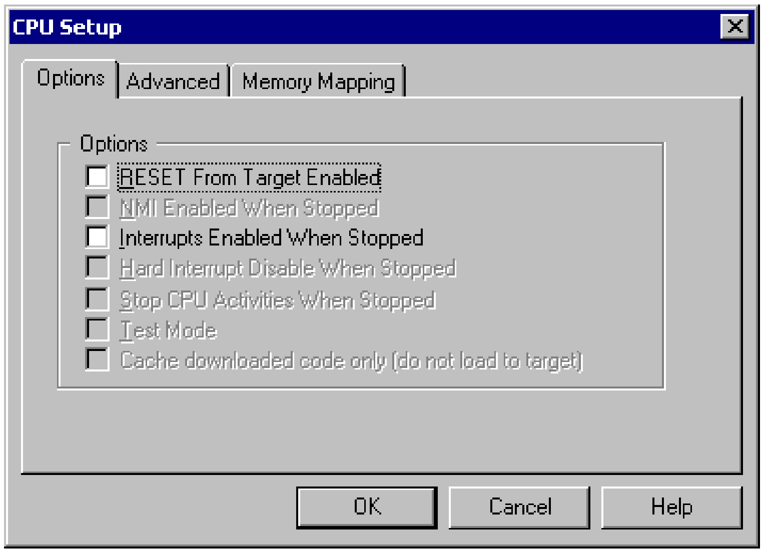
In single shot mode selecting the 'Run Pattern' executes a single pattern shot.

Note: Pattern generator operation can be controlled by an external device through the TRIG/CLKEN pin on the pattern generator connector. Refer to the Hardware User’s Manual for more information.

3 Setting CPU options

3.1 CPU Options

The CPU Setup, Options page provides some emulation settings, common to most CPU families and all emulation modes. Settings that are not valid for currently selected CPU or emulation mode are disabled. If none of these settings is valid, this page is not shown.



Options
- [ ] RESET From Target Enabled
- [ ] NMI Enabled When Stopped
- [ ] Interrupts Enabled When Stopped
- [ ] Hard Interrupt Disable When Stopped
- [ ] Stop CPU Activities When Stopped
- [ ] Test Mode
- [ ] Cache downloaded code only (do not load to target)

CPU Setup, Options page

RESET From Target Enabled

When checked, the target reset line is sensed, which can then reset the CPU while the CPU is running.

Interrupts Enabled When Stopped

Source step debug command is considered as a stop mode from the debugger's standpoint although it is implemented by setting breakpoints (hidden to the user) on adequate addresses and running the program up to them. This particular option refers to the stop mode, but it really impacts source step behaviour only. When the program is stopped, the CPU enters in so-called BDM mode in which the CPU cannot service any interrupt. Any pending interrupts are serviced after the program is resumed.

When this option is checked, I (Interrupt Mask) flag in the CCR register is not modified by the emulator. It means that when the user program is stopped or stepped through the sources, I flag is not affected by the emulator. For instance, if interrupts are enabled and there is an active interrupt request, it will be serviced during source step.

When this option is unchecked, the interrupts are disabled in stop mode. After the user program is stopped, the emulator, hidden from the user, memorizes the current I (Interrupt Mask) flag state and sets it, which result in disabled interrupts. When the user program is resumed, the emulator first restores original I flag and then resumes the program.

There is no problem when the 'Run' command is being used, but a problem can occur under certain conditions when a single step command is being used.

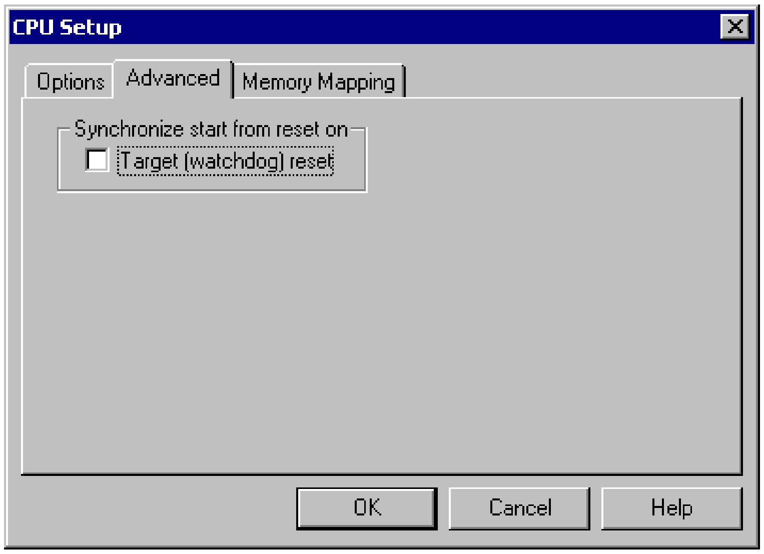
While in stop and executing a single step in the disassembly window there are no problems. During single step in the disassembly window the emulator itself detects any instruction that changes the state of I flag and handles it correctly. For example, interrupts are active and the program is stopped. The emulator memorizes the I flag state and disables interrupts. Now the user executes single steps in the disassembly window and, for example, once the SWI instruction (software interrupt) is stepped. At this moment, the CPU pushes the content of the CCR register to the stack, where the Interrupt Enable flag is stored and jumps to the address where the interrupt vector points to. Before the user's program was stopped (from running), the interrupts were active (I flag cleared) and after the program was stopped, they were disabled (I flag set) by the emulator. Therefore an incorrect I flag value (CCR register) is now pushed to the stack. Since the emulator can detect such an instruction it modifies the stack with the proper I flag value. If this would not be done, the program execution would be changed after RETI instruction is executed on software interrupt routine exit. Interrupts in the user's program would now be disabled and not enabled as before while the program was running.

But when using step in the source window (source step) the above problem becomes relevant and the user should never forget it. The source step is actually executed with RUN command with prior setting breakpoints on the required source lines. If for instance SWI (software interrupt) occurs during the source step execution, the CCR value with disabled interrupts will be pushed to the stack and after returning from the software interrupt routine (RETI) the same value is popped up from the stack. When the user resumes his program, interrupts are disabled and not enabled, as they were, before the program was stopped.

During the source step the emulator cannot detect instructions that changes the state of I flag as it is the case with single step in the disassembly window.

3.2 Advanced Options

Note: Advanced Options are not available on all HC08 PODs.



Freescale HC08 Advanced Options

Synchronize Start From Reset On

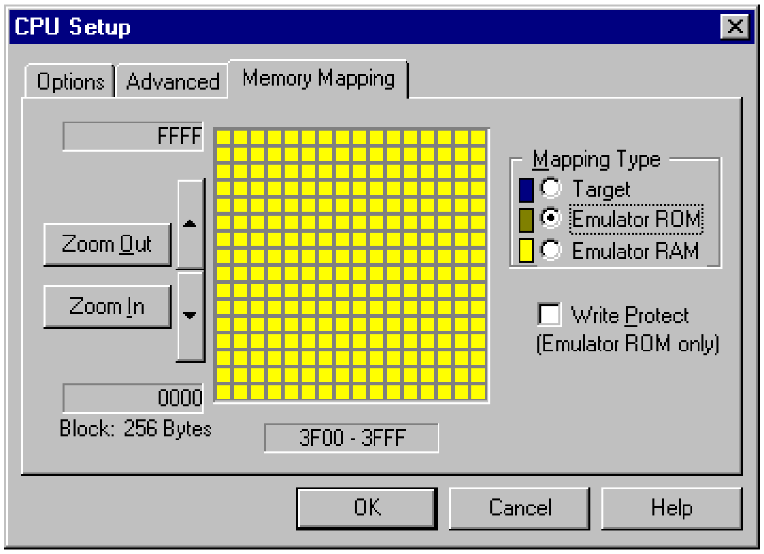
The target reset signal resets the CPU immediately. However, when the target reset line becomes inactive, the CPU reset line is belayed for few hundred milliseconds by the emulator.

If there is an active external watchdog, the CPU restart must be synchronized with the external watchdog. Then this option and 'RESET From Target Enabled' option must be checked. The watchdog timer event allows reset synchronization on the rising edge of external watchdog (target) reset. Note that the external watchdog must be a periodic signal (while forcing the CPU to a reset state). After the CPU starts, the external watchdog must be refreshed by the application, which ensures the target reset line not to be active.

3.3 Memory Mapping

Note: Memory Mapping is not available for development system based on HC08 Active POD and certain HC08 Power PODs.

The mapping page displays currently configured memory mapping.



FFFF
0000
Block: 256 Bytes
3F00 - 3FFF

Mapping Type
Target
[✓ Emulator ROM]
Emulator RAM

Write Protect (Emulator ROM only)

CPU Setup dialog, Mapping page

Gray blocks in mapping configuration area indicate memory ranges that are either outside the CPU's range (bank systems) or aren't covered by emulation memory.

Colored blocks define current mapping of the covered area:

- dark blue for target
- brown for Emulator ROM - CPU can read from it but not write to it.
- yellow for Emulator RAM - read and write access
- cyan for blocks with mixed mapping - use zoom to view where exactly such blocks map.

To change the mapping type of a block, select desired Mapping Type and click on the block that you wish to map to the select type.

Note: Clicking on a block with mixed mapping, clears all underlying mapping configuration and sets mapping for the entire block to selected mapping.

To configure and view mapping at higher resolution:

- click the 'Zoom In' button
- position the mouse cursor over the block that you wish to zoom in; the mouse cursor will change to indicate zoom mode.
- click on the block.

The mapping configuration area always shows a grid of 256 blocks. In the bottom left corner the current block size is displayed and current ranges are visible to the left of the mapping configuration area. You can zoom in and out and scroll the current range to reach the desired address and resolution. Minimum memory mapping resolution is 2 bytes.

In general you should configure the mapping for HC08 family (if available) as follows:

- Map CPU Internal ROM or Flash area as Emulator ROM.
- Map CPU Internal RAM area as Emulator RAM.

There is no need for 'Target' mapping since all CPUs are single chip CPUs.

Write Protect

Prevents the memory, mapped to the Emulator ROM, from being written to. If this option is checked, a write to the Emulator ROM area by the user application yields a warning message. This option applies to the Emulator ROM type of memory only.

4 Debugging Interrupt Routines

An interrupt routine can only be debugged when the interrupt source for this routine has been disabled; otherwise you will keep reentering the routine and thus run out of system stack.

For example, there is an interrupt routine with 10 source lines. Let's assume that the interrupt routine is called periodically by a free running timer. A breakpoint is set on the first source line in the interrupt routine. Program execution stops at the breakpoint. Now source step is executed. Source step is actually executed using RUN command with prior setting of breakpoint on adequate source line. In this particular case, while source step is executed, the CPU executes the code and before the source step actually completes, a new interrupt call is generated by the timer. Consequentially new values are pushed onto the stack and the CPU stops on breakpoint again. If you repeat source steps in such interrupt routine new values are pushed to the stack and you can easily run out of stack.

An interrupt source can be disabled in two ways:

- Disable the interrupt process when the program is stopped (stop mode) – refer to chapter 3.1 describing 'Interrupts Enabled When Stopped' option. Stop mode is entered whenever the CPU is stopped and the emulator remains in stop mode until the Run command is executed. (During Step, Step over, etc. commands, stop mode persists).
- Do not place a breakpoint on any instruction in the interrupt routine where interrupts are not yet disabled. You should also not step over any instruction that re-enables the interrupt, but run the program only up to that point.

5 Memory Access

HC08 development tools feature standard monitor memory access, which require user program to be stopped and real-time memory access based on shadow memory, which allows reading the memory while the application is running.

Real-Time Memory Access

Real-time memory access is available on iC1000, HC08 Active POD and PowerEmulator unit with shadow memory. Data area can be read in real-time only.

iC1000 and HC08 Active POD don't feature shadow memory.

In case of iC1000, Analyzer hardware resources are used instead, which result in few limitations. When real-time access is required, Analyzer must be used in the Trace or Profiler mode. In any other mode, real-time access cannot be used. Additionally, real-time access doesn't work during Trigger/Profiler configuration.

In case of HC08 Active POD, Coverage hardware resources are used instead which yield few limitations. When real-time access is required, Execution Coverage and Access Coverage cannot be used. If Coverage is started accidentally, real-time access will be inhibited till next CPU reset.

Real-time write memory access is not possible due to shadow memory use. Monitor access must be used to write to the memory.

Monitor Access

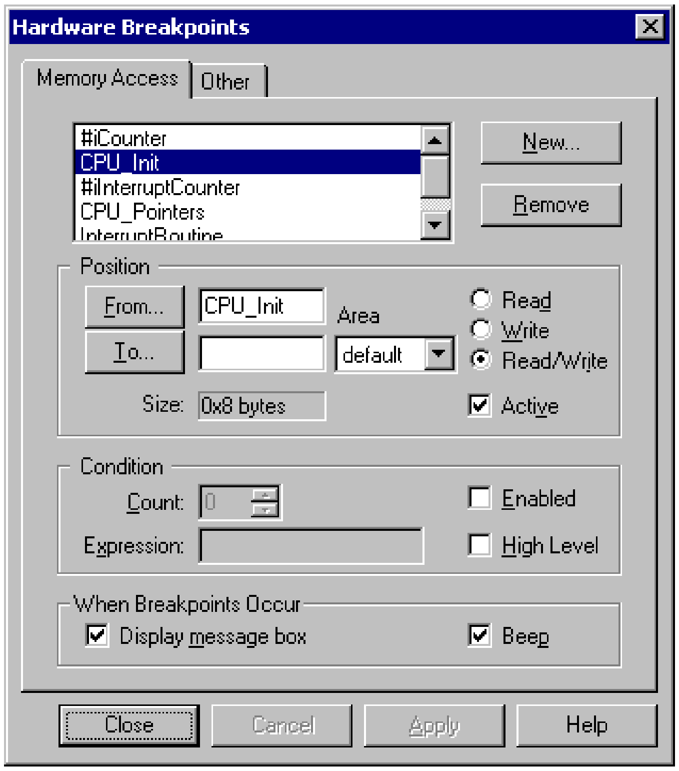
When monitor access to the CPU's memory is requested, the emulator stops the CPU and instructs it to read the requested number of bytes.

Since all accesses are performed using the CPU, all memory available to the CPU can be accessed. The drawback to this method is that memory cannot be accessed while the CPU is running. Stopping the CPU, accessing memory and running the CPU is an option, which, however, affects the real time execution considerably.

The time the CPU is stopped for is relative and cannot be exactly determined. The software has full control over it. It stops the CPU, updates all required windows and sets the CPU back to running. Therefore the time depends on the communication type used, PC's frequency, CPU's clock, number of updated memory locations (memory window, SFR window, watches, variables window), etc.

6 Access Breakpoints

The access breakpoints dialog is open by clicking Hardware breakpoints button in the Breakpoints dialog.

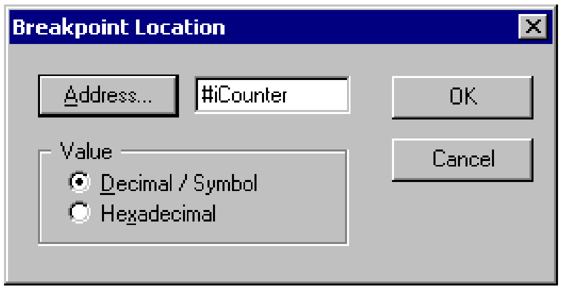


#iCounter
CPU_Init
#iInterruptCounter
CPU_Pointers
InterruptRoutine

From: CPU_Init
Area: default
Read / Write / [✓ Read/Write]
Size: 0x8 bytes
[✓] Active
Count: 0
Expression:
[ ] Enabled
[ ] High Level
[✓] Display message box
[✓] Beep

HC08 Hardware Breakpoints dialog

By pressing the 'New…' button, a new breakpoint is specified.



#iCounter

Value
[✓ Decimal / Symbol] Hexadecimal

By pressing the ‘Address…’ button, the symbol browser is invoked to specify the breakpoint address. Its value can also be entered in decimal or hexadecimal, and its type must be specified accordingly in the ‘Value’ option.

The breakpoint can also be specified as an access to a range, specified in the ‘From’ and ‘To’ positions and its access can be specified – whether it is read, write or read/write access that triggers the breakpoint. To enable the breakpoint, also the ‘Active’ checkbox must be checked.

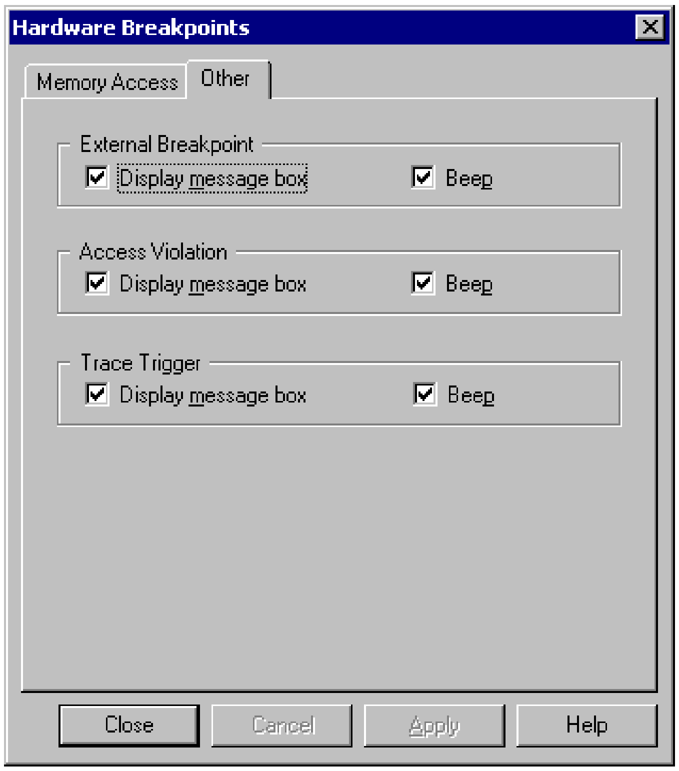
The breakpoint trigger condition can also be combined to a certain count of occurences (for example, the third occurrence of the condition triggers the breakpoint) and an expression (any valid C expression) can be specified to calculate the trigger condition. For this, the ‘Enabled’ checkbox in the ‘Condition’ options must be checked. If ‘High Level’ is checked, the trigger will evaluate the condition when the next source line is reached; if unchecked, the evaluation will occur immediatelly.

When a breakpoint occurs, two options can be selected – whether a message box will pop up and whether a beep will be produced.

6.1 Other break events

On a number of conditions, also a message box can be popped up and/or a beep produced.

These settings are available for HC08 Power POD based development systems only.



External Breakpoint
[✓] Display message box [✓] Beep

Access Violation
[✓] Display message box [✓] Beep

Trace Trigger
[✓] Display message box [✓] Beep

External Breakpoint

When an external breakpoint is triggered on the pin on the POD, the selected actions are performed.

Access Violation

When a write to a read-only memory is performed, the selected actions are performed.

Note: this option is only available for PODs that have the internal mapping already specified on the POD.

Trace Trigger

When a trace trigger event occurs, the selected actions are performed.

7 Emulation Notes

7.1 Internal RAM

During the emulation the internal RAM of the HC08 CPU on the POD is disabled internally, thereby adequate memory area must be mapped as Emulator RAM by the user.

7.2 Internal CPU Flash

Internal FLASH is overlaid by the emulation memory and disabled during the emulation and cannot be used in any way.

7.3 Setting Execution Breakpoints

It is recommended to set breakpoints in the source window where breakpoints are set on source lines. After modifying and rebuilding the project, the breakpoints are still set on the same source line, even though the physical address may have changed. It is dissuaded to set breakpoints on physical addresses because after modifying the project and linking it, a previously set breakpoint may not be legal/correct any more.

7.4 COP Use

The internal COP must be either disabled in the CONFIG-1 register (if the CPU has such option) or serviced by the user's program, otherwise the emulation fails. Refer to the CPU datasheet for more details on disabling COP in the CONFIG-1 register (write once). Note that COP is usually enabled after reset.

While the user’s program is stopped, the emulator updates the COP counter.

When COP timer expires, it generates resets. The CPU reset is bi-directional signal. For the emulator to be able to recognize this reset (or any other), 'Reset from target enabled' option must be checked. When COP reset (or any other) occurs, the emulator detects it and starts the CPU in the emulation mode and set it to run.

If the 'Reset from target enabled' option is not checked and COP reset occurs, the CPU is reset, but not started in the emulation mode since the emulator doesn't detect reset. The emulator recognizes only that the CPU doesn’t run/respond and displays HALTED status.

COP servicing by the user's program

Writing any value to location 0xFFFF before overflow occurs, clears the COP counter and prevents reset. A user must be careful since reset vector and COP register are located at the same address.

There is no problem if a ROM is located at the reset vector. Writing any value to 0xFFFF clears the COP counter and does not mangle the reset vector. Typically, a reset vector is covered by the emulation memory, therefore, it is advised to allocate the ROM type at reset vector location.

If a RAM (e.g. Emulator RAM) is located at 0xFFFF, the user has to write a low byte value of reset vector in the COP register. Otherwise, reset vector is mangled and the program counter has incorrect value after next reset. The most secure way is that the user first reads from 0xFFFF and then writes back the same value to clear the COP counter and to retain the original reset vector.

COP update routine should be placed in the main program and not in an interrupt subroutine. Such an interrupt subroutine could keep the COP from generating a reset even while the main program is not working properly.

7.5 STOP mode

When using the 68HC08GP20 CPU and when a STOP instruction is executed the CPU should enter the STOP mode. Note that first STOP instruction must be enabled and then everything works fine. STOP bit in the CONFIG1 register must be set to enable the STOP instruction. Nevertheless, if the STOP instruction is not enabled the CPU treats it as an illegal instruction and the CPU generates reset for a few clock cycles. If the emulator doesn't service the reset correctly (the CPU requires a power-on reset), the CPU on the POD doesn't start any more in the CTM mode (emulation mode) and the emulator must be switched off and on. The "Reset From Target Enabled" option must be checked in the 'Hardware/In-Circuit Emulation/CPU Setup/Options' tab. Now, the emulator will detect reset from the CPU when executing STOP instruction (illegal instruction) and service it properly.

7.6 Miscellaneous

Remove all emulator breakpoints when performing any kind of checksum since they may impact the checksum result.

8 Trace

HC08 development systems feature a powerful trace, which is implemented externally to the CPU. HC08 Active PODs feature a so called Active Trace, which additionally offers following features comparing to the HC08 Power PODs:

- Enlarged trace buffer
- Duration Tracker
- Watchdog trigger
- Pre/Post Qualifier
- Q between B and C
- Data Change

Refer to a separate document describing Active Trace features and use.

9 Execution Coverage

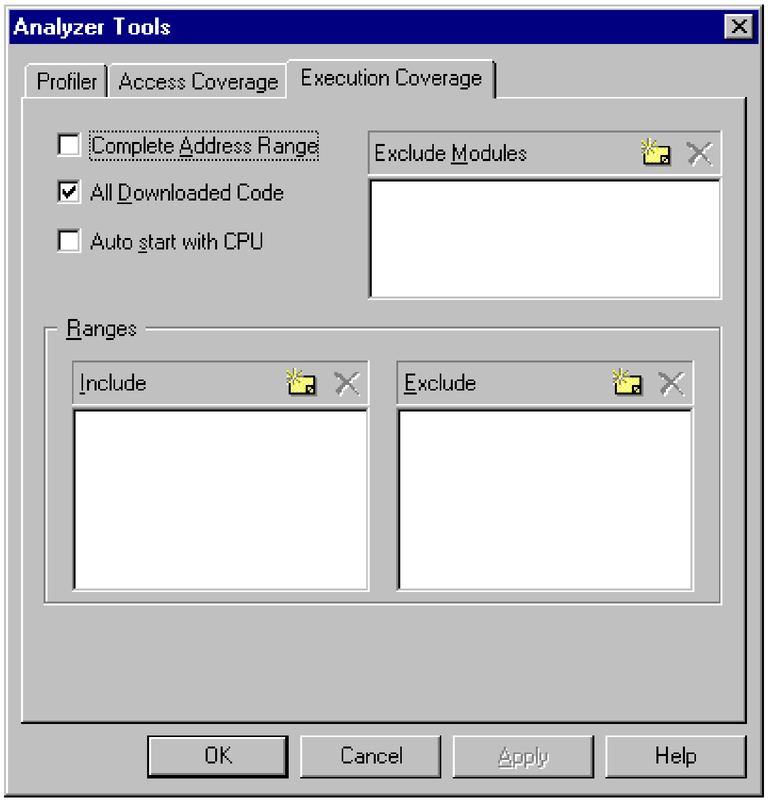
Execution coverage records all addresses being executed, which allows the user to detect the code or memory areas not executed. It can be used to detect a so called "dead code", the code that was never executed. Such code represents undesired overhead when assigning code memory resources.

Execution coverage covers complete CPU address space. It can run infinite time, which means in practice that the application can run for days and then the execution coverage results can be analyzed.

- First, make sure that 'Active' Trace is selected in the Hardware/Analyzer Setup dialog.

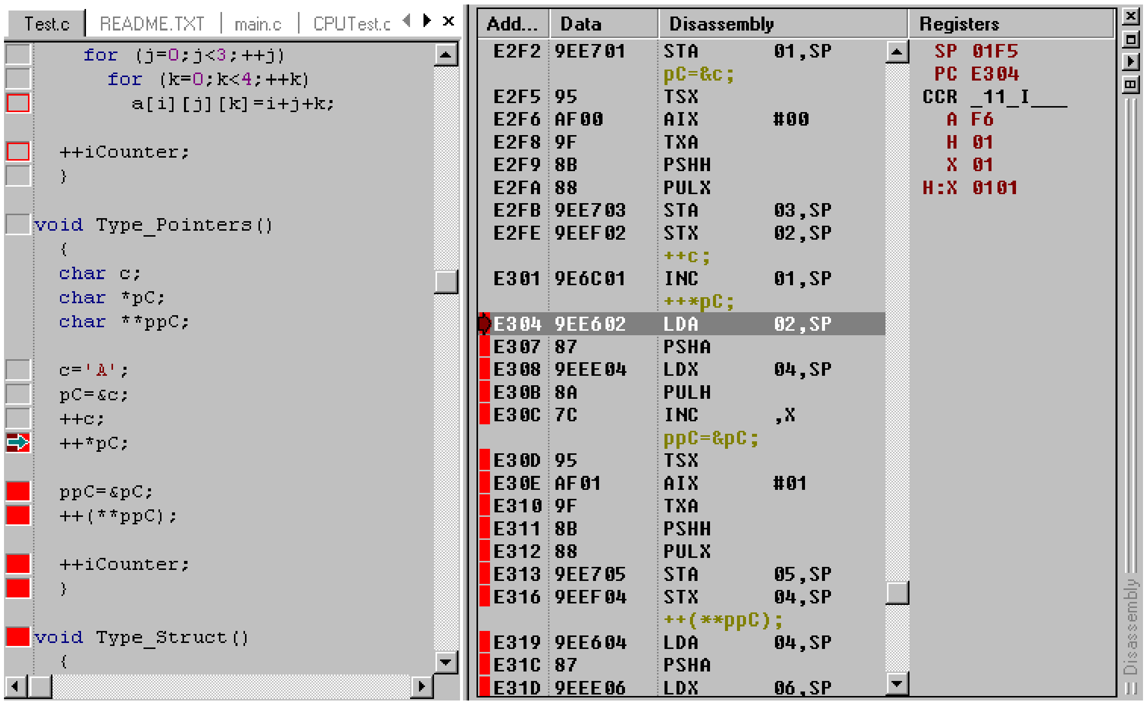
- Next, select 'Execution Coverage' window from the View menu and configure Execution Coverage settings. Normally, 'All Downloaded Code' option has to be checked only. The debugger extracts all the necessary information like addresses belonging to each C/C++ function from the debug info, which is included in the download file and configure hardware accordingly.

Refer to software user's guide for more details on configuring Execution Coverage and its use.


Complete Address Range [ ]
All Downloaded Code [✓]
Auto start with CPU [ ]

Execution Coverage is configured. Reset the application, start Execution Coverage and then run the application. When it's assumed that the complete application code was executed, stop the Execution Coverage and inspect the results.

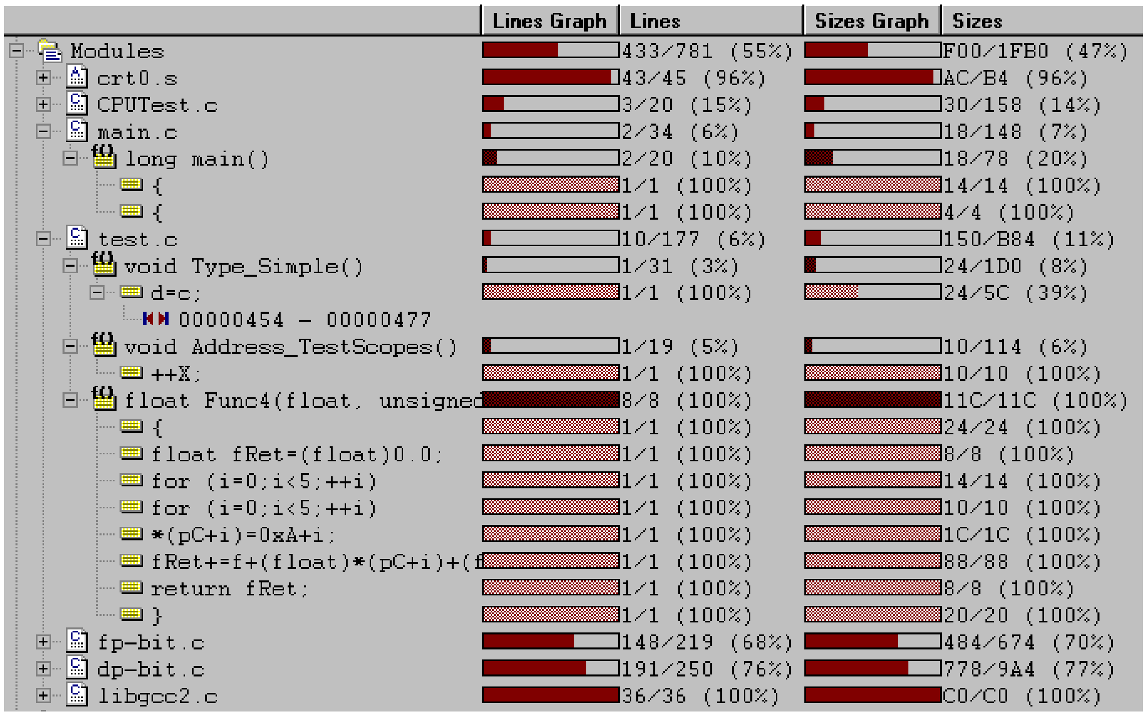
Red boxes on the left in the source window and disassembly window depict not executed code. Execution Coverage window shows which code was executed/not executed for selected modules.



```
    for (j=0;j<3;++j)
      for (k=0;k<4;++k)
        a[i][j][k]=i+j+k;

  ++iCounter;
  }

void Type_Pointers()
  {
  char c;
  char *pC;
  char **ppC;

  c='A';
  pC=&c;
  ++c;
  ++*pC;

  ppC=&pC;
  ++(**ppC);

  ++iCounter;
  }

void Type_Struct()
  {
```

```
Add... Data    Disassembly              Registers
E2F2  9EE701  STA        01,SP          SP  01F5
              pC=&c;                    PC  E304
E2F5  95      TSX                       CCR _11_I___
E2F6  AF00    AIX        #00              A  F6
E2F8  9F      TXA                         H  01
E2F9  8B      PSHH                        X  01
E2FA  88      PULX                      H:X  0101
E2FB  9EE703  STA        03,SP
E2FE  9EEF02  STX        02,SP
              ++c;
E301  9E6C01  INC        01,SP
              ++*pC;
E304  9EE602  LDA        02,SP
E307  87      PSHA
E308  9EEE04  LDX        04,SP
E30B  8A      PULH
E30C  7C      INC        ,X
              ppC=&pC;
E30D  95      TSX
E30E  AF01    AIX        #01
E310  9F      TXA
E311  8B      PSHH
E312  88      PULX
E313  9EE705  STA        05,SP
E316  9EEF04  STX        04,SP
              ++(**ppC);
E319  9EE604  LDA        04,SP
E31C  87      PSHA
E31D  9EEE06  LDX        06,SP
```

Source & Dissassembly Window: Execution Coverage results



```
                                        Lines Graph  Lines           Sizes Graph  Sizes
Modules                                              433/781 (55%)                F00/1FB0 (47%)
  crt0.s                                             43/45 (96%)                  AC/B4 (96%)
  CPUTest.c                                          3/20 (15%)                   30/158 (14%)
  main.c                                             2/34 (6%)                    18/148 (7%)
    long main()                                      2/20 (10%)                   18/78 (20%)
      {                                              1/1 (100%)                   14/14 (100%)
      {                                              1/1 (100%)                   4/4 (100%)
  test.c                                             10/177 (6%)                  150/B84 (11%)
    void Type_Simple()                               1/31 (3%)                    24/1D0 (8%)
      d=c;                                           1/1 (100%)                   24/5C (39%)
        00000454 - 00000477
    void Address_TestScopes()                        1/19 (5%)                    10/114 (6%)
      ++X;                                           1/1 (100%)                   10/10 (100%)
    float Func4(float, unsigned                      8/8 (100%)                   11C/11C (100%)
      {                                              1/1 (100%)                   24/24 (100%)
      float fRet=(float)0.0;                         1/1 (100%)                   8/8 (100%)
      for (i=0;i<5;++i)                              1/1 (100%)                   14/14 (100%)
      for (i=0;i<5;++i)                              1/1 (100%)                   10/10 (100%)
      *(pC+i)=0xA+i;                                 1/1 (100%)                   1C/1C (100%)
      fRet+=f+(float)*(pC+i)+(f                      1/1 (100%)                   88/88 (100%)
      return fRet;                                   1/1 (100%)                   8/8 (100%)
      }                                              1/1 (100%)                   20/20 (100%)
  fp-bit.c                                           148/219 (68%)                484/674 (70%)
  dp-bit.c                                           191/250 (76%)                778/9A4 (77%)
  libgcc2.c                                          36/36 (100%)                 C0/C0 (100%)
```

Execution Coverage Window results

10 Execution Profiler

Profiler records executed function entry and exit points and then run time-analysis over the collected information. As a result it gives details on how much time (minimum, maximum, average) has the CPU spent in the particular function. Available information allows the user to optimize those parts of code, which are most time consuming or time critical.

The debug download file must contain accurate debug information when using Profiler to analyze C/C++ application. Normally Profiler extracts all the necessary information from the debug information and becomes useless if configured for wrong function entry and exit points.

- First, make sure that 'Active' Trace is selected in the Hardware/Analyzer Setup dialog.

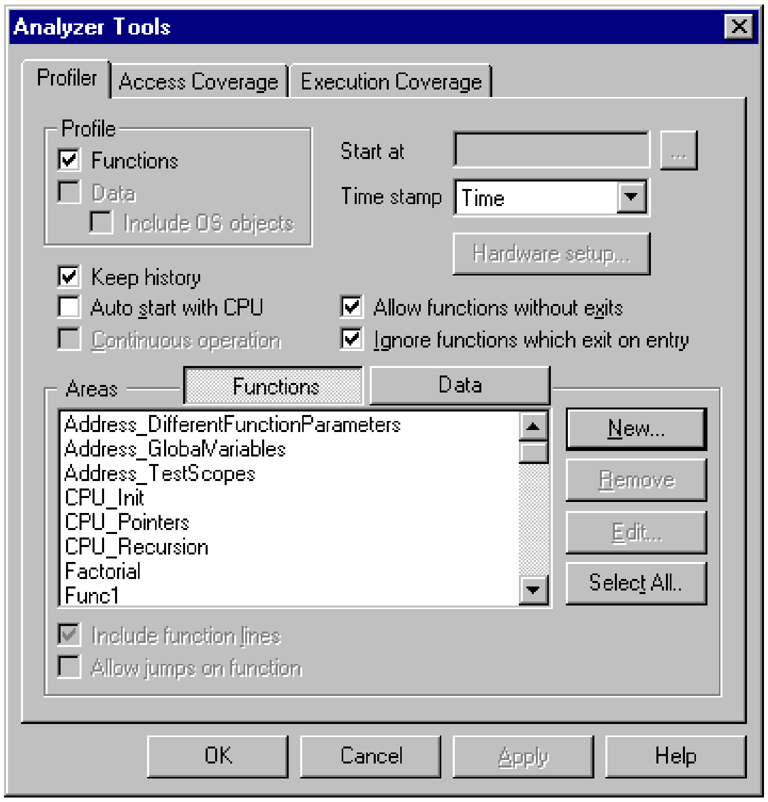
- Next, select 'Profiler' window from the View menu and configure Profiler settings. Select 'Functions' option in the 'Profile' field.

- Make sure that 'Keep history' option is checked if Code Execution view is going to be used during results analysis.

- Finally, profiled C/C++ functions are selected by pressing 'New…' button. It's recommended that 'All C Functions' is selected for the beginning. Additionally, 'Include lines' can be checked which will yield in time analysis of each source line belonging to the function.

The debugger extracts all the necessary information from the debug info, which is included in the download file and configure the hardware accordingly.

Refer to software user's guide for more details on configuring Profiler and its use.



Analyzer Tools

Profiler | Access Coverage | Execution Coverage

Profile
- [✓] Functions
- [ ] Data
- [ ] Include OS objects

Start at:

Time stamp: Time

Hardware setup...

- [✓] Keep history
- [ ] Auto start with CPU
- [ ] Continuous operation
- [✓] Allow functions without exits
- [✓] Ignore functions which exit on entry

Areas: Functions | Data

Address_DifferentFunctionParameters
Address_GlobalVariables
Address_TestScopes
CPU_Init
CPU_Pointers
CPU_Recursion
Factorial
Func1

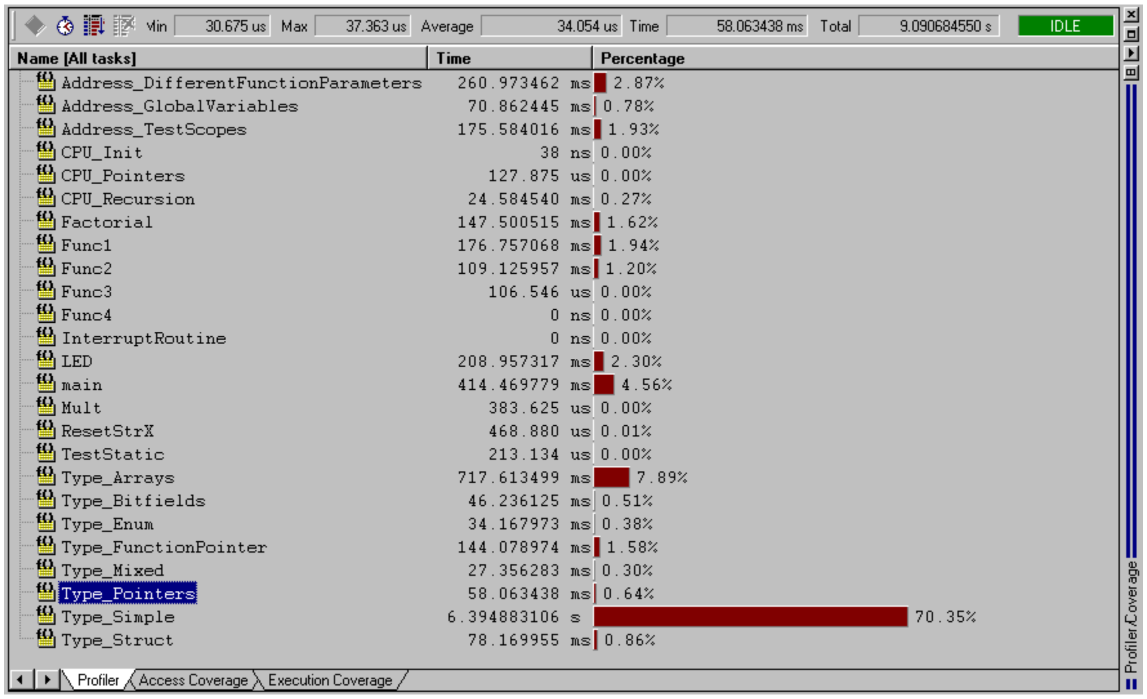
New... | Remove | Edit... | Select All..

- [✓] Include function lines
- [ ] Allow jumps on function

OK | Cancel | Apply | Help

Profiler configuration settings

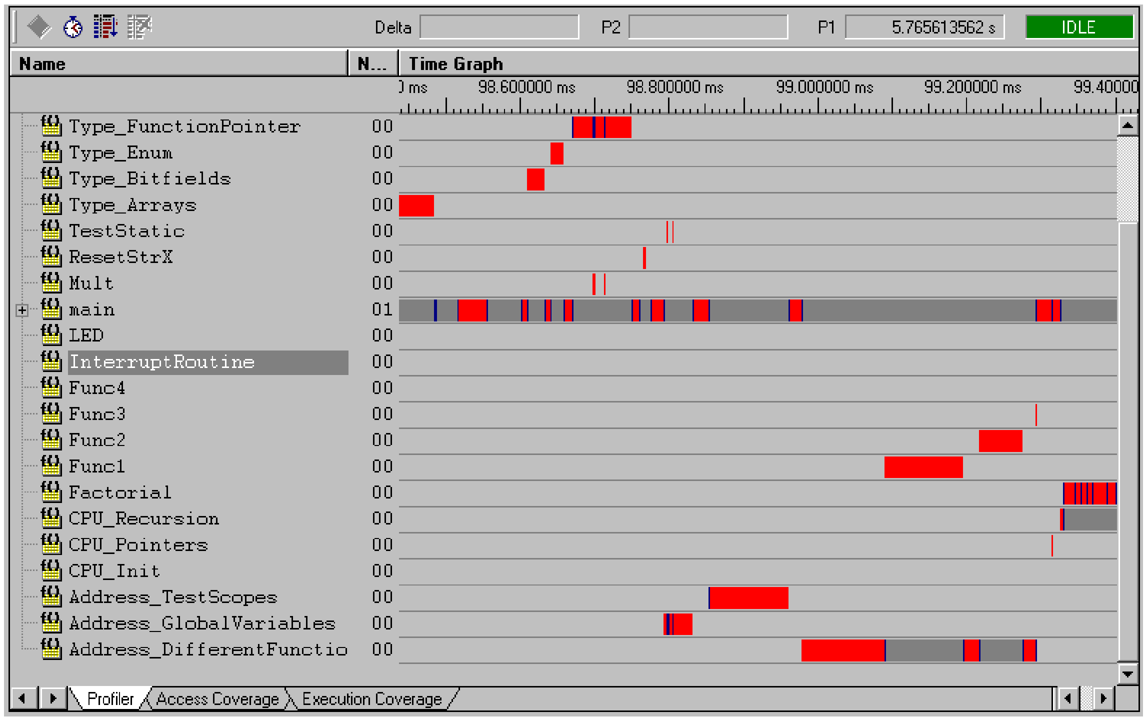
Profiler is configured. Reset the application, start Profiler and then run the application. The Profiler will stoop collecting information on a user demand or after the trace buffer becomes full. Then the recorded information is analyzed and profiler results displayed.



Min 30.675 us Max 37.363 us Average 34.054 us Time 58.063438 ms Total 9.090684550 s IDLE

Name [All tasks]	Time	Percentage
Address_DifferentFunctionParameters	260.973462 ms	2.87%
Address_GlobalVariables	70.862445 ms	0.78%
Address_TestScopes	175.584016 ms	1.93%
CPU_Init	38 ns	0.00%
CPU_Pointers	127.875 us	0.00%
CPU_Recursion	24.584540 ms	0.27%
Factorial	147.500515 ms	1.62%
Func1	176.757068 ms	1.94%
Func2	109.125957 ms	1.20%
Func3	106.546 us	0.00%
Func4	0 ns	0.00%
InterruptRoutine	0 ns	0.00%
LED	208.957317 ms	2.30%
main	414.469779 ms	4.56%
Mult	383.625 us	0.00%
ResetStrX	468.880 us	0.01%
TestStatic	213.134 us	0.00%
Type_Arrays	717.613499 ms	7.89%
Type_Bitfields	46.236125 ms	0.51%
Type_Enum	34.167973 ms	0.38%
Type_FunctionPointer	144.078974 ms	1.58%
Type_Mixed	27.356283 ms	0.30%
Type_Pointers	58.063438 ms	0.64%
Type_Simple	6.394883106 s	70.35%
Type_Struct	78.169955 ms	0.86%

Profiler results – Code Statistics view



Delta P2 P1 5.765613562 s IDLE

Name	N...
Type_FunctionPointer	00
Type_Enum	00
Type_Bitfields	00
Type_Arrays	00
TestStatic	00
ResetStrX	00
Mult	00
main	01
LED	00
InterruptRoutine	00
Func4	00
Func3	00
Func2	00
Func1	00
Factorial	00
CPU_Recursion	00
CPU_Pointers	00
CPU_Init	00
Address_TestScopes	00
Address_GlobalVariables	00
Address_DifferentFunctio	00

Profiler results – Code Statistics view

Notes: