

F²MC-16LX FAMILY

SUPPORT TOOL

EMULATOR SYSTEM MB2147-01

EMULATING AND DEBUGGING WITH SOFTUNE AND MB2147-01

APPLICATION NOTE

Revision History

| Date | Issue |
|------------|--|
| 2003-02-13 | V1.0; MWi |
| 2003-03-18 | V1.1; MWi – upgraded |
| 2003-04-04 | V1.2; MWi – upgraded |
| 2003-08-12 | V1.3; MWi – performance measurement added |
| 2003-09-26 | V1.4; MWi – upgraded (break points, real time memory) |
| 2004-10-12 | V1.5; MWi – upgraded (notes, Appendix added) |
| 2005-04-11 | V1.6; MWi – upgraded (RT watch view table added, Additional picture in Power on debugging description) |
| 2005-04-12 | V1.7; MWi – upgraded |
| 2006-10-24 | V1.8; MWi – Resource Register Watch added |
| 2007-10-24 | V1.9; MWi – Data break configuration added |
| 2008-01-17 | V2.0; MWi – Features since Softune version V30L34R05 added |

This document contains 49 pages.

Warranty and Disclaimer

To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH restricts its warranties and its liability for **all products delivered free of charge** (eg. software include or header files, application examples, target boards, evaluation boards, engineering samples of IC's etc.), its performance and any consequential damages, on the use of the Product in accordance with (i) the terms of the License Agreement and the Sale and Purchase Agreement under which agreements the Product has been delivered, (ii) the technical descriptions and (iii) all accompanying written materials. In addition, to the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH disclaims all warranties and liabilities for the performance of the Product and any consequential damages in cases of unauthorised decompiling and/or reverse engineering and/or disassembling. **Note, all these products are intended and must only be used in an evaluation laboratory environment.**

1. Fujitsu Microelectronics Europe GmbH warrants that the Product will perform substantially in accordance with the accompanying written materials for a period of 90 days from the date of receipt by the customer. Concerning the hardware components of the Product, Fujitsu Microelectronics Europe GmbH warrants that the Product will be free from defects in material and workmanship under use and service as specified in the accompanying written materials for a duration of 1 year from the date of receipt by the customer.
2. Should a Product turn out to be defect, Fujitsu Microelectronics Europe GmbH's entire liability and the customer's exclusive remedy shall be, at Fujitsu Microelectronics Europe GmbH's sole discretion, either return of the purchase price and the license fee, or replacement of the Product or parts thereof, if the Product is returned to Fujitsu Microelectronics Europe GmbH in original packing and without further defects resulting from the customer's use or the transport. However, this warranty is excluded if the defect has resulted from an accident not attributable to Fujitsu Microelectronics Europe GmbH, or abuse or misapplication attributable to the customer or any other third party not relating to Fujitsu Microelectronics Europe GmbH.
3. To the maximum extent permitted by applicable law Fujitsu Microelectronics Europe GmbH disclaims all other warranties, whether expressed or implied, in particular, but not limited to, warranties of merchantability and fitness for a particular purpose for which the Product is not designated.
4. To the maximum extent permitted by applicable law, Fujitsu Microelectronics Europe GmbH's and its suppliers' liability is restricted to intention and gross negligence.

NO LIABILITY FOR CONSEQUENTIAL DAMAGES

To the maximum extent permitted by applicable law, in no event shall Fujitsu Microelectronics Europe GmbH and its suppliers be liable for any damages whatsoever (including but without limitation, consequential and/or indirect damages for personal injury, assets of substantial value, loss of profits, interruption of business operation, loss of information, or any other monetary or pecuniary loss) arising from the use of the Product.

Should one of the above stipulations be or become invalid and/or unenforceable, the remaining stipulations shall stay in full effect

Contents

| | |
|---|-----------|
| REVISION HISTORY | 2 |
| WARRANTY AND DISCLAIMER | 3 |
| CONTENTS | 4 |
| 0 INTRODUCTION | 7 |
| 1 SAMPLE PROGRAM | 8 |
| 1.1 Start a new Project with Softune Workbench | 8 |
| 1.2 “Main.c” | 8 |
| 1.2.1 Source Code | 8 |
| 1.3 Compiling “Main.c” | 9 |
| 2 DEBUGGING, FIRST STEPS | 10 |
| 2.1 Setup Hardware | 10 |
| 2.2 Entering Debugger Mode | 10 |
| 2.2.1 Mixed Display | 10 |
| 2.3 Using Bookmarks | 11 |
| 2.4 Start Execution | 11 |
| 2.5 Stop Execution | 12 |
| 2.6 Reset MCU | 12 |
| 3 BREAKPOINTS AND PROGRAM STEPPING | 13 |
| 3.1 Setting Break Points | 13 |
| 3.2 Using a Break Point | 13 |
| 3.3 Setting Break Points during program execution | 15 |
| 3.4 Data Break Points | 16 |
| 3.4.1 Data Break Points at ROM Constants | 16 |
| 3.5 Important Note: | 16 |
| 4 MONITORING AND MANIPULATING | 17 |
| 4.1 Monitoring and Manipulating Processor Status | 17 |
| 4.2 Monitoring and Manipulating CPU Registers | 17 |
| 4.3 Monitoring and Manipulating C Variables | 18 |
| 4.3.1 Example of Manipulating Variables | 18 |
| 4.4 Monitoring Resource Registers | 19 |
| 4.5 Monitoring and Manipulating Memory | 19 |
| 4.5.1 Example of Manipulating Memory | 20 |

| | | |
|-----------|---|-----------|
| 4.5.2 | Realtime Memory..... | 20 |
| 4.5.3 | Memory Watch updated at runtime | 21 |
| 4.6 | Symbol view | 22 |
| 4.6.1 | Icon Reference | 22 |
| 4.7 | Local variables | 22 |
| 4.8 | Important Note: | 23 |
| 5 | TRACE..... | 24 |
| 5.1 | Trace View | 24 |
| 5.1.1 | Instruction View | 24 |
| 5.1.2 | Cycle View..... | 25 |
| 5.1.3 | Cycle View abbreviations:..... | 26 |
| 5.2 | Back Trace | 26 |
| 5.3 | Saving Trace Data..... | 27 |
| 5.4 | Important Note: | 27 |
| 6 | TIME MEASUREMENT..... | 28 |
| 6.1 | Setup Time Measurement | 28 |
| 6.2 | Important Note: | 29 |
| 7 | SEQUENCE | 30 |
| 7.1 | Setup Sequence function | 30 |
| 7.2 | Important Note: | 33 |
| 8 | COVERAGE..... | 34 |
| 8.1 | Setup Coverage | 34 |
| 8.2 | Important Note: | 35 |
| 9 | PERFORMANCE MEASUREMENT..... | 36 |
| 9.1 | Setup Performance Measurement..... | 36 |
| 9.2 | Important Note: | 37 |
| 10 | POWER-ON DEBUGGING | 38 |
| 10.1 | Preparation..... | 38 |
| 10.2 | Setting break points..... | 38 |
| 10.3 | Enter Power-On Debug Mode | 38 |
| 10.4 | Important Note: | 40 |
| 11 | TRIGGER-INPUT AND EMULATOR-OUTPUT..... | 41 |
| 11.1 | The BNC Connectors | 41 |
| 11.2 | Trigger-Input..... | 41 |
| 11.3 | Emulator-Output..... | 41 |

| | |
|--------------------------------------|-----------|
| 12 APPENDIX A | 42 |
| 12.1 Debug environment settings..... | 42 |

0 Introduction

This installation guide will help you how to debug an emulation system with the MB2147-01 Emulation Hardware for the new 0.35 μm technology with the Softune Workbench V30L28/29. For in-depth information please refer to the following manuals:

- MB2147-01 Hardware Manual (Emulator)
- MB2147-10 Hardware Manual (Adapter Board PGA256P)
- MB2147-20 Hardware Manual (Adapter Board PGA299P)
- MB2147-01 Getting Started Application Note (AN-FMEMCU-900069)
- MB2147-01 Installation Guide Application Note (AN-FMEMCU-900070)

This document describes the debugging methods of a MB90V340 system together with a Flash-CAN-100P-340 target board. Please note, that the debugging principle is the same for other evaluation systems, even those of the 0.5 μm technology (e.g. MB90V540).

1 Sample Program

SAMPLE PROGRAM FOR DEBUGGING

1.1 Start a new Project with Softune Workbench

At first choose an evaluation MCU (here: MB90V340), copy the template project of the “Softune samples” into an own folder (here: “Emulation_Test”) and start the Softune Workbench Software. Please note, that all examples are for demonstration purpose only. Fujitsu is not taking over responsibility for any reliability.

1.2 “Main.c”

The following program, based on the standard template project, is used for demonstrating emulation and debugging. Please change “Main.c” to the following:

1.2.1 Source Code

```
1  /* THIS SAMPLE CODE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS. FUJITSU */
2  /* MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY FOR ANY ERRORS OR */
3  /* ELIGIBILITY FOR ANY PURPOSES. */
4  /* (C) Fujitsu Microelectronics Europe GmbH */
5  /*-----*/
6  MAIN.C
7  - description
8  - See README.TXT for project description and disclaimer.
9
10 /*-----*/
11
12 #include "mb90340.h" /* can be any other MB90xxx MCU */
13
14 void wait (int i) /* Waiting routine */
15 {
16     volatile int j;
17
18     for (j = 0; j < i; j++);
19 }
20
21 int a, b, c; /* global variables */
22
23 void main(void)
24 {
25     InitIrqLevels();
26     __set_il(7); /* allow all levels */
27     __EI(); /* globally enable interrupts */
28
29     a = 2;
30     DDR0 = 0xFF; /* Port0 as Output */
31
32     while (1) /* Endless Loop */
33     {
34         for (b = 0; b < 10; b++)
35         {
36             c = b * a;
37             PDR0 = c; /* Write to Port0 */
38             wait (5000);
39         }
40     }
41 }
```

This program is only an example with no “great assignment“. It contains a simple wait-function (`void wait`), which needs an integer value for the wait time. The resulting delay time depends on the value itself and the clock speed of the emulation system.

The main program (`void main`) uses the global variables `a`, `b`, and `c`. At first the interrupts are enabled (although they are not used in this example), then the variable `a` is set to 2 and the Port0 of the MCU is set to “output“ (Port0 is the LED-Port of the Flash-CAN-100P-340 board).

Next follows the endless main loop. In this loop the variable `b` is count from 0 to 9. At each counting this variable multiplied by `a` is stored in `c`. `c` is then output to Port0.

Finally the `wait` function is called with the value 5000, which causes a delay of about 11 ms at an internal CPU clock of 16 MHz, and the main loop is performed again.

1.3 Compiling “Main.c”

To compile the project, please use “Setup Project” first. In *Project*→*Setup Project*→*C Compiler*→*Category*: *Optimize* has to be selected General-purpose Optimization Level: **None**.

Then compile the project by clicking on  Build all source files regardless of data, selecting *Project*→*Build*, or pressing “Ctrl-F8”.

Watch for error messages. If all is ok. you will get the following message:

```
Now building...
-----Configuration: Template.prj - Debug-----
Start.asm
Main.c
vectors.c
Mb90340.asm
Now linking...
<Your path>\Emulation_Test\ABS\Template.abs
Now starting load module converter...
<Your path>\Emulation_Test\ABS\Template.mhx

-----
No Error.
-----
```

2 Debugging, first steps

HOW TO ENTER DEBUGGING MODE

2.1 Setup Hardware

For the next steps you have to set up your emulation hardware. Please refer to the application notes “Installation Guide MB2147-01” (AN-FMEMCU-900070) and “Emulator System MB2147-01, Getting started” (AN-FMEMCU-900069) for details.

2.2 Entering Debugger Mode

After successful compilation of the project start the debugging mode via COM1/2, USB or LAN by double clicking on the regarding Debug-“.sup”-entry in the workspace window. After successful connection to the emulator open “Main.c” (close it first, if it is open) and then click on right mouse button and select “*Mix Display*”. Your generated code should look like the following.

2.2.1 Mixed Display

```

. . .
12: #include "mb90340.h"
13:
14: void wait (int i)
15: {
FE0088: 0802          LINK    #02
16:  volatile int j;
17:
18:  for (j = 0; j < i; j++);
FE008A: D0             MOVN   A,#0
FE008B: CBFE          MOVW  @RW3-02,A
FE008D: BBFE          MOVW  A,@RW3-02
FE008F: 767306       CMPW  A,@RW3+06
FE0092: FB05          BGE   FE0099
FE0094: 7353FE       INCW  @RW3-02
FE0097: 60F4          BRA   FE008D
19: }
FE0099: 09             UNLINK
FE009A: 66             RETP
20:
21: int a, b, c;
22:
23: void main(void)
24: {
FE009B: 0800          LINK    #00
25:  InitIrqLevels();
FE009D: 65E300FE     CALLP  \InitIrqLevels
26:  __set_il(7); /* allow all levels */
FE00A1: 1A07          MOV    ILM,#07
27:  __EI(); /* globally enable interrupts */
FE00A3: 2540          OR    CCR,#40
28:
29:  a = 2;
FE00A5: D2             MOVN   A,#2
FE00A6: 5B0401       MOVW  0104,A
30:  DDR0 = 0xFF;
FE00A9: 5410FF       MOV    I:10,#FF

```

Continued on next page

Continuation from previous page

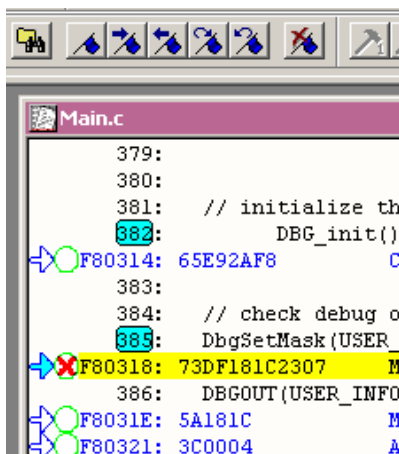
```

31:
32:     while (1)
33:     {
34:         for (b = 0; b < 10; b++)
FE00AC: D0             MOVN    A, #0
FE00AD: 5B0201          MOVW   0102,A
FE00B0: 5A0201          MOVW   A,0102
FE00B3: DA             MOVN   A,#A
FE00B4: 2B             CMPW   A
FE00B5: FB1E           BGE    FE00D5
    35:         {
    36:             c = b * a;
FE00B7: 5A0201          MOVW   A,0102
FE00BA: 783F0401        MULUW  A,0104
FE00BE: 5B0001          MOVW   0100,A
    37:             PDR0 = c;
FE00C1: 520001          MOV    A,0100
FE00C4: 5100           MOV    I:00,A
    38:             wait (5000);
FE00C6: 4A8813          MOVW   A,#1388
FE00C9: 4C             PUSHW A
FE00CA: 658800FE        CALLP \wait
FE00CE: 5D             POPW  AH
    39:         }
FE00CF: 735F0201        INCW   0102
FE00D3: 60DB           BRA    FE00B0
    40:     }
FE00D5: 60D5           BRA    FE00AC
    41: }
FE00D7: 09             UNLINK
FE00D8: 66             RETP
FE00D7: 09             UNLINK
FE00D8: 66             RETP

```


2.3 Using Bookmarks

Since Softune version V30L34R05 it is possible to set bookmarks in source code lines or debugging windows.




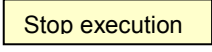
Bookmarked lines are marked with the line number in a green bubble in the debug window and completely in green in source code lines. With the bookmark arrow buttons it can be stepped through the code stopping at bookmarked lines.

2.4 Start Execution

To enter the run mode, click on  Run continuously, select Debug→Run→Go, or press “F5”.

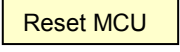
Now the program is being executed. If you are using a target system with LEDs on Port0, you will see the LEDs flicker.

2.5 Stop Execution

To stop the MCU, click on   or select Debug→Abort.

Now the system is halted, but it can be continued again by clicking on “*Run continuously*” or selecting “Go”.

2.6 Reset MCU

To reset the MCU, click on   or select Debug→Reset of MCU.

3 Breakpoints and Program Stepping

HOW TO SET BREAK POINTS AND HOW TO USE SINGLE STEPS

3.1 Setting Break Points

Each assembler line in the mixed mode display of the source code has a blue arrow and a green circle symbol:



In these line a breakpoint can be set by clicking into the circle. The symbol then turns to



3.2 Using a Break Point

Assume you want to examine the program in the example above. A good breakpoint would be just before calling the `wait`-function. Therefore set the breakpoint to Address `H'FE00C6`, where the accumulator is loaded with the content of address `H'0100` (variable `c`).

```

35:      {
36:      c = b * a;
=>( ) FE00B7: 5A0201      MOVW   A,0102
=>( ) FE00BA: 783F0401    MULUW  A,0104
=>( ) FE00BE: 5B0001      MOVW   0100,A
37:      PDR0 = c;
=>( ) FE00C1: 520001      MOV    A,0100
=>( ) FE00C4: 5100        MOV    I:00,A
38:      wait (5000);
=>(x) FE00C6: 4A8813      MOVW   A,#1388
=>( ) FE00C9: 4C          PUSHW  A
=>( ) FE00CA: 658800FE    CALLP  \wait
=>( ) FE00CE: 5D          POPW   AH
39:      }

```

Now start execution (press “F5”).


The MCU then executes the program just until your breakpoint is reached. The source window will then look like this:

```

35:      {
36:      c = b * a;
=>( ) FE00B7: 5A0201      MOVW   A,0102
=>( ) FE00BA: 783F0401    MULUW  A,0104
=>( ) FE00BE: 5B0001      MOVW   0100,A
37:      PDR0 = c;
=>( ) FE00C1: 520001      MOV    A,0100
=>( ) FE00C4: 5100        MOV    I:00,A
38:      wait (5000);
=>(x) FE00C6: 4A8813      MOVW   A,#1388
=>( ) FE00C9: 4C          PUSHW  A
=>( ) FE00CA: 658800FE    CALLP  \wait
=>( ) FE00CE: 5D          POPW   AH
39:      }

```

The yellow highlighted line is the actual line where the CPU is halt (actual program counter in code section H' FE).

Now we want to step an instruction further. Therefore click on , or select *Debug*→*Run*→*Step in*, or press “F6”. The source window will change to:

```

35:      {
36:          c = b * a;
=>( ) FE00B7: 5A0201      MOVW    A,0102
=>( ) FE00BA: 783F0401    MULUW  A,0104
=>( ) FE00BE: 5B0001      MOVW    0100,A
37:          PDR0 = c;
=>( ) FE00C1: 520001      MOV     A,0100
=>( ) FE00C4: 5100        MOV     I:00,A
38:          wait (5000);
=>(x) FE00C6: 4A8813      MOVW    A,#1388
=>( ) FE00C9: 4C          PUSHW  A
=>( ) FE00CA: 658800FE    CALLP  \wait
=>( ) FE00CE: 5D          POPW   AH
39:      }
    
```


The CPU has performed the “move-word” instruction and is waiting at the “push-word-accu” instruction.

Do two instruction steps further (press two times “F5”). Then the highlighted line will jump to the wait function:

```

14: void wait (int i)
15: {
=>( ) FE0088: 0802      LINK   #02
16: volatile int j;
17:
18: for (j = 0; j < i; j++);
=>( ) FE008A: D0        MOVN   A,#0
=>( ) FE008B: CBFE      MOVW   @RW3-02,A
=>( ) FE008D: BBFE      MOVW   A,@RW3-02
=>( ) FE008F: 767306    CMPW   A,@RW3+06
=>( ) FE0092: FB05      BGE    FE0099
=>( ) FE0094: 7353FE    INCW   @RW3-02
=>( ) FE0097: 60F4      BRA    FE008D
19: }
=>( ) FE0099: 09        UNLINK
=>( ) FE009A: 66        RETP
    
```


Now we are “trapped”. Because the function `wait` has the parameter value 5000 (specified by `main`) for counting, we would have to press about (number_of_instructions_in_wait multiplied by 5000) times “F5” until we would be back in the `main` function.


Therefore the step-out command exists:  (*Debug*→*Run*→*Step out* or “Shift-F6”). Using this stepping method, after a **further** step-in within the function, the next execution stop will be in the caller function (here: `main`) just after return from the sub function:

```

38:          wait (5000);
=> (x) FE00C6: 4A8813          MOVW    A, #1388
=> ( ) FE00C9: 4C             PUSHW  A
=> ( ) FE00CA: 658800FE       CALLP  \wait
=> ( ) FE00CE: 5D             POPW   AH
39:          }
    
```

Note, if you are using the simulator, you can't step out from the top of a function. In emulation mode the step-out command at the top of a function causes in a continued run mode.

To step through a program without entering sub functions you can use the step-over command:  (Debug → Run → Step Over or "F7")

With the Execute-until-cursor-position command (, Debug → Run → Run Until Cursor or "Ctrl-F7") you can set a temporary breakpoint. Therefore click into the source window so that the cursor is flashing. Then reset the CPU and perform the command. The execution will stop at the cursor position.

Note, that the execution only stops, if the marked program code is executed. Be careful with conditional instructions using this command.

You also can jump to a specified memory location by directly clicking at the address. When turning the mouse pointer over the left side of the mix display window the pointer turns to:



Clicking then sets a "temporary break point". This means that the code is executed until the marked address is reached.

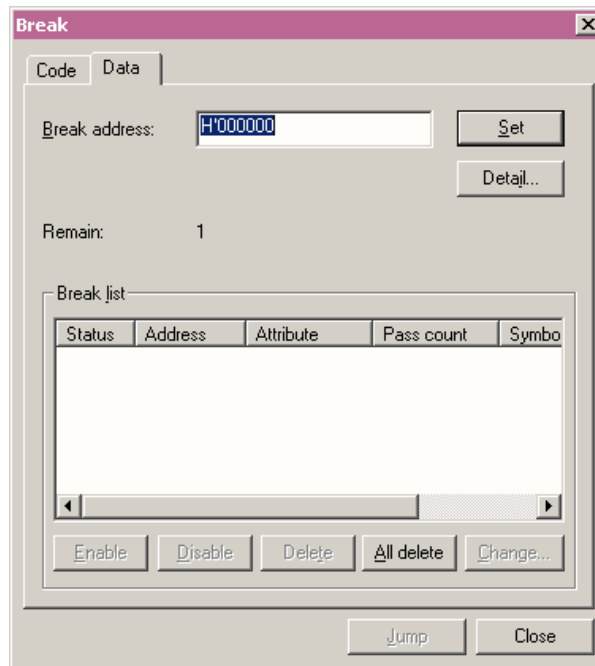
Note, that this only works, if this code line is ever reached.

3.3 Setting Break Points during program execution

With Softune version V30L31 or higher break points can be set during program execution. Therefore this function has to be enabled by: Setup → Debug environment → Execution: Setting break point while running: enable.

3.4 Data Break Points

It is also possible to set data break points. In the *Break...* dialog (right-click on mouse within source window area) it can be chosen between code and data break.



To set a breakpoint when a certain (global) variable is accessed, double click the variable in the debug source window and then choose with right-click on mouse *Break...* menu. In this case the address of the variable is automatically suggested in the field *Break address*:

Please note, that after setting always Set should be clicked to add the new break point to the list.

3.4.1 Data Break Points at ROM Constants

In the small or medium memory model (`_near`) ROM constants can't be set for a break point automatically when ROM mirror is used. The symbol reference is still located in `0xFF` bank. When breaking on a constant read access, please change address in the break configuration from `0xFFxxxx` to `0x00xxxx`.

3.5 Important Note:

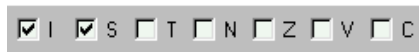
Please be aware that most debug environment settings are only accessible, if the program execution is stopped – not during execution of a program.

4 Monitoring and Manipulating

HOW TO MONITOR AND MANIPULATE CPU REGISTERS, VARIABLES, MEMORY

4.1 Monitoring and Manipulating Processor Status

The Condition Code Register (CCR) is always displayed below the workspace window.



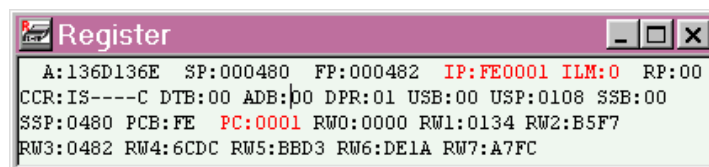
The flags are:

| Abbr. | Flag name |
|-------|--|
| I | Interrupt enable flag (1 = enable) |
| S | Stack flag (0 = User stack; 1 = System stack) |
| T | Sticky bit flag (1 = shift right instruction executed) |
| N | Negative flag (MSB = 1 in last operation) |
| Z | Zero flag (Last operation resulted in "0") |
| V | Overflow flag (Overflow at last operation) |
| C | Carry flag (Last operation caused carry) |

The value of the flags can be easily changed by clicking into the white square. A "check mark" (√) indicates that the flag is set (== 1).

4.2 Monitoring and Manipulating CPU Registers

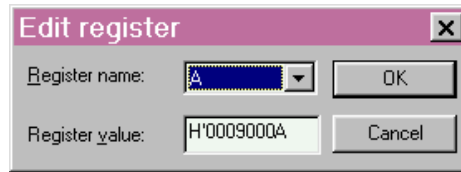
To display the CPU Registers window choose in the debugging mode: View→Register. A new window will occur and look like this:



The registers are:

| Abbr. | Register name |
|---------|---|
| A | Accumulator |
| SP | Stack Pointer (USP or SSP) |
| FP | Frame Pointer |
| IP | Instruction Pointer (PC + Code section) |
| ILM | Interrupt Level Mask (belongs to PS) |
| RP | Register Bank Pointer (belongs to PS) |
| CCR | Condition Code Register (belongs to PS) |
| DTB | Data Bank Register |
| ADB | Additional Data Bank Register |
| DPR | Direct Page Register |
| USB | User Stack Bank Register |
| USP | User Stack Pointer |
| SSB | System Stack Bank Register |
| SSP | System Stack Pointer |
| PCB | Program Bank Register |
| PC | Program Counter |
| RW0-RW7 | General Purpose Registers |

The contents of these registers can be changed by double-clicking them. A pop-up window will occur and look like the following picture:



In the dialog frame *Register value* you can enter a new value for the register. Note, that the values always are shown in hexadecimal notation by set-up default, but you can enter even decimal values (beginning with “D’”), binary values (beginning with “B’”), or octal values (beginning with “O’”).

4.3 Monitoring and Manipulating C Variables

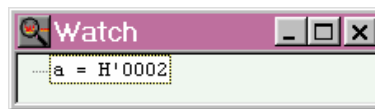
To display C variables choose in the debugging mode: *View*→*Watch*→*Watch1*. A new window *Watch* will occur.

Click in this window on the right mouse button and select *Set...*. A pop-up window occurs.



In the text frame *Variable name* type in the variable name *a* of the example program above. The *Mode* must be *C language* in this case.

The *Watch* window will then contain the variable name and value.



Note: You can change the radix of the value by right-clicking on the variable entry and choose via *Radix*: Binary, Octal, Decimal, or Hexadecimal.

To manipulate the value just double-click on the entry and enter in the pop-up window *Edit variable* a new value. The radix can be chosen via “D’”, “H’”, “B’”, or “O’”.

(Note: The variables within the *Watch* window are only updated during execution if their memory location is defined in one of the *Realtime Memory* areas, otherwise an error message is displayed during execution. See 4.4.2)

4.3.1 Example of Manipulating Variables

The following example shows how to intervene in a program.

First use the example C code “Main.c” from above and enter debugging mode. Open “Main.c” in *Mix display* mode. Now set a break point at the beginning of the main loop in *main*.

```

32:   while (1)
33:   {
34:     for (b = 0; b < 10; b++)
=> (x) FE00AC: D0          MOVN   A, #0
=> ( ) FE00AD: 5B0201     MOVW   0102, A
=> ( ) FE00B0: 5A0201     MOVW   A, 0102
=> ( ) FE00B3: DA         MOVN   A, #A
=> ( ) FE00B4: 2B         CMPW   A
=> ( ) FE00B5: FB1E      BGE    FE00D5
35:   {

```

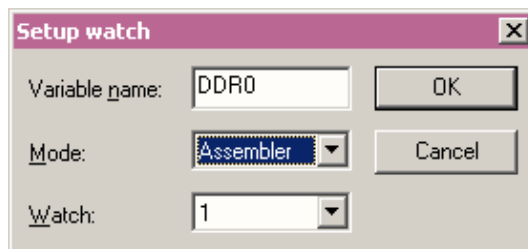
Now start execution and the debugger will halt on the break point. Next, display variables *a*, *b*, and *c* as explained in 4.3.

The value of *a* should be 2, the others 0. Change *a* for instance to 4 and re-run the program.

The CPU will halt at the break point again, and the variables will have the values: *a* = H'02, *b* = H'0A, and *c* = H'24. This shows that the variable manipulation was successful, because *c* would never get the value H'24 if there were no intervention (The "normal" maximum of *c* would be 2*9 = 18 = H'12).

4.4 Monitoring Resource Registers

Resource registers can be monitored and viewed in the watch windows. Softune automatically references their symbol. To display and change their contents, please add a symbol (e. g. *DDRO*), the click on right mouse button and choose *Set...→Mode:Assembler*.



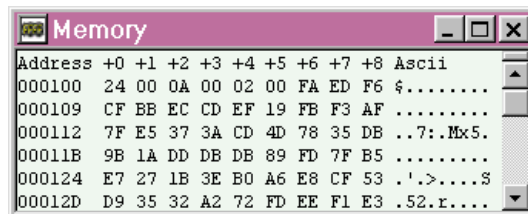
By using the default mode "Automatic", only the address of the resource register is displayed in the watch window.

4.5 Monitoring and Manipulating Memory

To display the MCU memory choose in the debugging mode: *View→Memory*. A pop-up window *Watch* will occur and ask for the start address to be displayed.

Type for instance H'100 (or just 100) for the RAM area (MB9034x).

A new window occurs which is something like a "Hex-Editor":



To change a memory content just double click on the respecting byte and a new dialog window pops up. In this window you can specify the address (default is the address of the clicked byte) and the new value. The value can be entered in hexadecimal, decimal, binary or octal format.

4.5.1 Example of Manipulating Memory

Use the example program from above compile it and enter debugging mode.

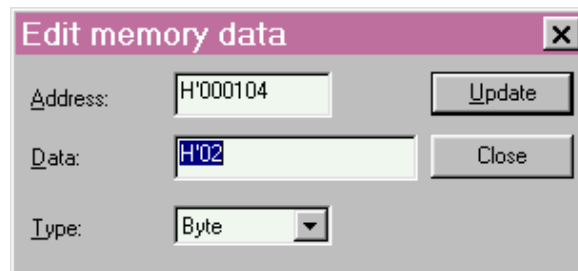
Set a break point at address H' FE00AC:

```

32:      while (1)
33:      {
34:      for (b = 0; b < 10; b++)
=> (x) FE00AC: D0          MOVN   A, #0
=> ( ) FE00AD: 5B0201     MOVW   0102, A
=> ( ) FE00B0: 5A0201     MOVW   A, 0102
=> ( ) FE00B3: DA         MOVN   A, #A
=> ( ) FE00B4: 2B         CMPW   A
=> ( ) FE00B5: FB1E      BGE    FE00D5
35:      {
    
```

Enter execution mode and the CPU stops at H' FE00AC. Now select View→Watch→Watch1 and add variable a. Then select View→Memory and enter start address H'100.

You will find the C variable a at the address H' 0104. Assume you want to change its value. Therefore double click on H' 0104. The following dialog window will pop up:

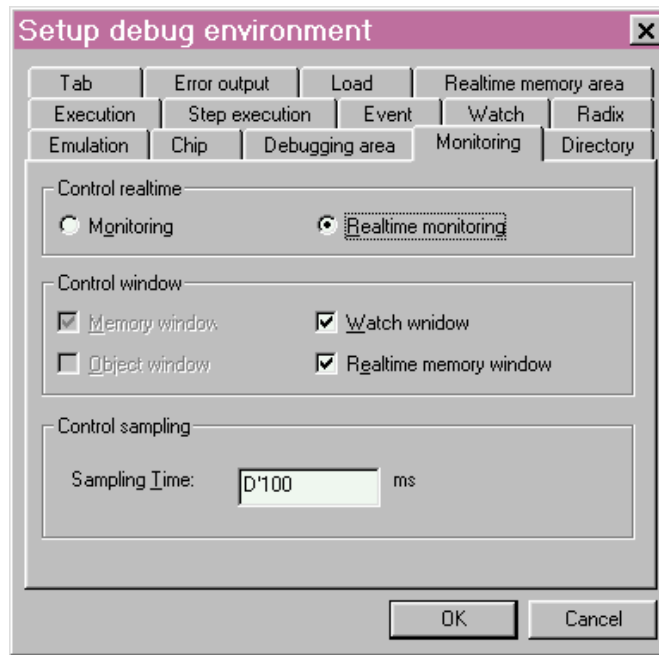


Now type 4 in the Data text field and watch the variable window. You will see that the value of a will change immediately to H' 0004.

4.5.2 Realtime Memory

There is another way to view memory. Choose View→Realtime memory. A Setup window will occur (if not, click the right button of the new opened window and select Setup Area...). In the Setup debug environment window select Realtime memory area. Enter “100” for the Start address for Area number: 1. Click on Set and then OK. The new Realtime memory window looks much like the Memory window, but is limited to 256 Bytes (per area), but highlights any memory changes in red.

To see the memory in “real time” during execution, choose Setup→Debug environment→Debug environment. Then select the Monitoring folder and enter “D' 100” at Sampling Time: (using USB connection to the PC) and click on check box Realtime memory window:



If you then execute the sample program the Realtime Memory window is updated all 100 ms, and you will see the addresses H' 0100 and H' 0102 changing its values. The changes are highlighted in red.

Note, memory manipulation is not possible in the *Realtime memory* window.

4.5.3 Memory Watch updated at runtime

Another way to watch the memory and variables updated at runtime is to do this via the memory and watch window. Therefore the coverage function has to be disabled. Otherwise you will get the error message, that the MCU is busy.

To disable the coverage function choose: *Setup* → *Debug environment* → *Debug environment* → *Chip: Coverage: Disable*. Also in the Debug environment settings at the *Monitoring* “folder”, *Control realtime: Monitoring* has to be selected.

Now all the memory in bank 0 can be watched during program execution.

Note that this feature is supported with Softune version V30L31 or higher.


The following table shows, which settings support memory and variable watch, which is updated at runtime:

| Debug Mode | Native Mode | RT memory set ¹ | Coverage | updated Watch at runtime |
|------------|-------------|----------------------------|----------|--------------------------|
| X | | | | X ² |
| | X | | | X |
| X | | X | | X |
| | X | X | | X |
| X | | | X | |
| | X | | X | |
| X | | X | X | X |
| | X | X | X | X |

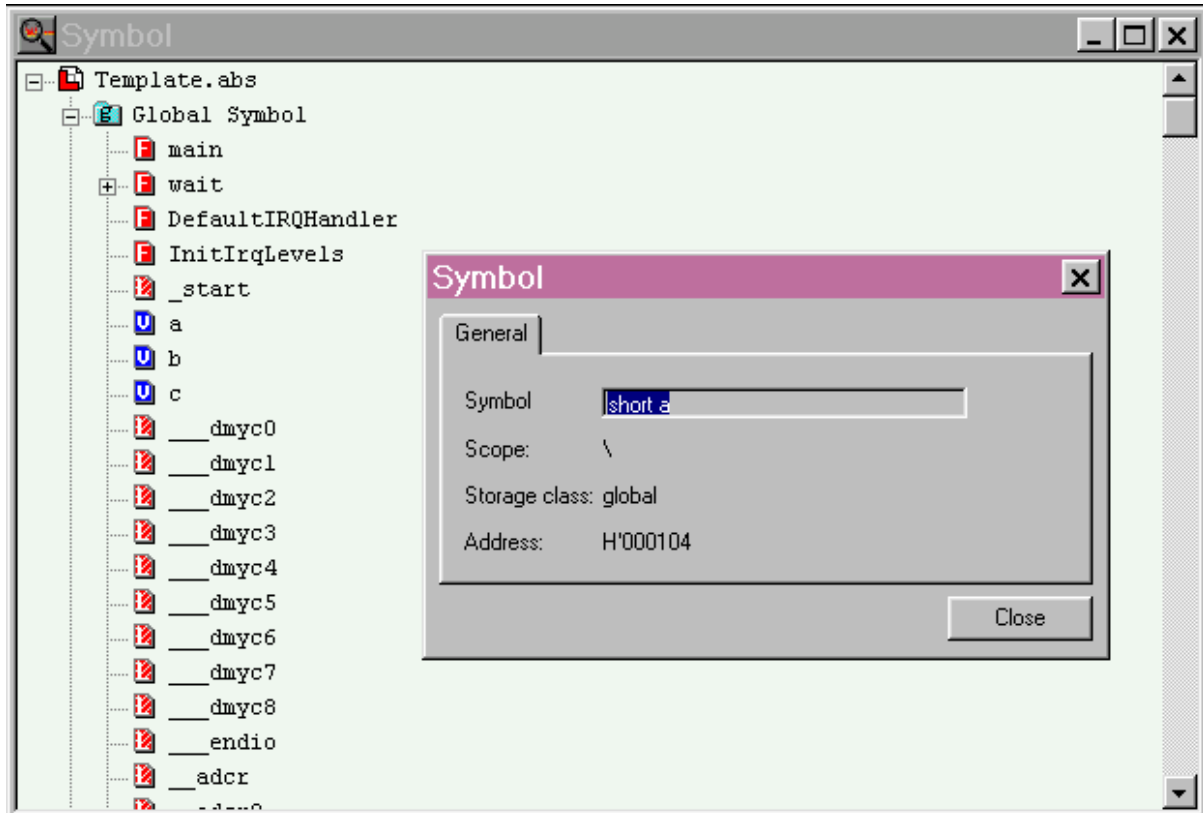
¹ means that the variable to be watched is located in the area defined in the Realtime Memory settings.

² supported in Softune version V30L32 or higher

4.6 Symbol view




If you want to know where a variable is located in the memory you can choose View→Symbol. Then unfold the sub list *Project_name.abs/Global Symbol*. Click with the right mouse button to the variable () you want to get information about and select *Property...*

The Symbol sub window shows then the address:



4.6.1 Icon Reference

The following icons are used:

| Icon | Description |
|---|-------------|
|  | Function |
|  | Label |
|  | Variable |

4.7 Local variables

Local variables of functions can be displayed via View→Local. A new window will open. Note, that this window only shows contents if the debugger is in stop mode (e.g. breakpoint reached) and the actual function has local variables.

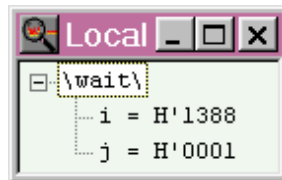
To demonstrate the *Local* view window open “*Main.c*” in *mixed display* and set a breakpoint at address H'FF0097 within the sub function *wait*.

```

14: void wait (int i)
15: {
=>( )FE0088: 0802          LINK      #02
16: volatile int j;
17:
18: for (j = 0; j < i; j++);
=>( )FE008A: D0           MOVN    A, #0
=>( )FE008B: CBFE          MOVW   @RW3-02,A
=>( )FE008D: BBFE          MOVW   A,@RW3-02
=>( )FE008F: 767306       CMPW   A,@RW3+06
=>( )FE0092: FB05          BGE    FE0099
=>( )FE0094: 7353FE       INCW   @RW3-02
=>(x)FE0097: 60F4          BRA    FE008D
19: }
=>( )FE0099: 09           UNLINK
=>( )FE009A: 66           RETP

```

Now start execution and look at the *Local* view window when the CPU stops at the breakpoint. The window will show the local variables *i* and *j* of the function *wait*:



4.8 Important Note:

Please be aware that most debug environment settings are only accessible, if the program execution is stopped – not during execution of a program.

5 Trace

HOW TO USE THE TRACE BUFFER

5.1 Trace View

Use the example program from above and set a break point (e.g. at address H' FE00AC) and enter execution mode. After the CPU has stopped, run the program again (to fill the trace buffer) and then choose View→Trace. A new window with the content of the trace buffer will open.

5.1.1 Instruction View

| Frame no. | address | mnemonic | time stamp |
|--------------|----------|-----------------------------|------------|
| -00062 | : FE0099 | UNLINK | 375 |
| -00059 | : 000482 | internal read access. 048A | 200 |
| -00056 | : 000186 | internal write access. 048A | 175 |
| -00055 | : FE009A | RETP | 75 |
| -00054 | : 000484 | internal read access. 00CE | 50 |
| -00051 | : 000486 | internal read access. 00FE | 200 |
| -00045 | : FE00CE | POPW AH | 375 |
| -00044 | : 000488 | internal read access. 1388 | 50 |
| main.c\$39 } | | | |
| -00041 | : FE00CF | INCW 0102 | 200 |
| -00037 | : 000102 | internal read access. 0009 | 250 |
| -00034 | : 000102 | internal write access. 000A | 175 |
| -00031 | : FE00D3 | BRA FE00B0 | 200 |
| -00025 | : FE00B0 | MOVW A,0102 | 375 |
| -00022 | : 000102 | internal read access. 000A | 175 |
| -00019 | : FE00B3 | MOVN A,#A | 200 |
| -00018 | : FE00B4 | CMPW A | 50 |
| -00017 | : FE00B5 | BGE FE00D5 | 75 |
| main.c\$40 } | | | |
| -00011 | : FE00D5 | BRA FE00AC | 375 |
| -00004 | : 000100 | internal read access. 24 | 425 |
| 00000 | : ** | THOLD ** | 250 |

This is the instruction view, which shows in the first column the frame number (bus cycle). Note, that the MB2147-01 Emulator can hold up to 65535 frames. The second column shows the internal bus address. The third column shows either the disassembled machine code or a data transfer (inclusive data value).

Note that the last executed frame has the number 0 and all previous frames negative numbers.

The time stamp shows the execution time of the instruction including all internal cycles in nano seconds (billionth seconds).

Please also note, that in this view the execution frame of the instructions is shown, not their fetch frame.

The read access from H' 000100 (Frame -00004) is an emulator internal read access and should be ignored for debugging.

The next illustration shows the relationship between execution (trace window) and the compiled source code:

| Source Code: | Trace view: |
|---|---|
| <pre> 31: 32: while (l) 33: { 34: for (b = 0; b < 10; b++) FE00AC: D0 MOVN A, #0 FE00AD: 5B0201 MOVW 0102,A FE00B0: 5A0201 MOVW A,0102 FE00B3: DA MOVN A,#A FE00B4: 2B CMPW A FE00B5: FB1E BGE FE00D5 35: { 36: c = b * a; FE00B7: 5A0201 MOVW A,0102 FE00BA: 783F0401 MULUW A,0104 FE00BE: 5B0001 MOVW 0100,A 37: PDR0 = c; FE00C1: 520001 MOV A,0100 FE00C4: 5100 MOV I:00,A 38: wait (5000); FE00C6: 4A8813 MOVW A,#1388 FE00C9: 4C PUSHW A FE00CA: 658800FE CALLP \wait FE00CE: 5D POPW AH 39: } FE00CF: 735F0201 INCW 0102 FE00D3: 60DB BRA FE00B0 40: } FE00D5: 60D5 BRA FE00AC 41: } FE00D7: 09 UNLINK FE00D8: 66 RETP </pre> | <pre> -00062 : FE0099 UNLINK 375 -00059 : 000482 internal read access. 048A 175 -00056 : 000186 internal write access. 048A 200 -00055 : FE009A RETP 75 -00054 : 000484 internal read access. 00CE 50 -00051 : 000486 internal read access. 00FE 200 -00045 : FE00CE POPW AH 375 -00044 : 000488 internal read access. 1388 50 main.c\$39 -00041 : FE00CF INCW 0102 200 -00037 : 000102 internal read access. 000 250 -00034 : 000102 internal write access. 175 -00031 : FE00D3 BRA FE00B0 200 -00025 : FE00B0 MOVW A,0102 375 -00022 : 000102 internal read access. 000A 175 -00019 : FE00B3 MOVN A,#A 200 -00018 : FE00B4 CMPW A 50 -00017 : FE00B5 BGE FE00D5 75 main.c\$40 -00011 : FE00D5 BRA FE00AC 375 -00004 : 000100 internal read access. 24 425 00000 : ** THOLD ** 250A </pre> |

The last 4 frames relating to the code in the trace view are linked with red lines to the program code. The last instruction which was executed is “BRA FE00AC” which directly jumps to the break point (red and bold highlighted in source code picture).

Note, you can jump to any frame by right clicking into the trace window using *Jump...*

Also a Trace Back function is provided to see the executed instructions in history.

5.1.2 Cycle View

To view the trace buffer by cycle click with the right mouse button into the trace window and choose *Cycle*. For the example above you will see the following:

| Frame no. | address | data | a-status | d-status | Qst | dfg | event | time stamp |
|-----------|----------|------|----------|----------|-----|-----|-------|------------|
| -00014 | : FE00D5 | ---- | ICF | ----- | FLH | & | | 75 |
| -00013 | : FE00D5 | 60 | --- | EXECUTE | --- | @ | | 50 |
| -00012 | : FE00D6 | ---- | ICF | ----- | --- | & | | 75 |
| -00011 | : FE00D6 | 09D5 | --- | EXECUTE | 1by | @ | | 50 |
| -00010 | : FE00D8 | ---- | ICF | ----- | --- | & | | 75 |
| -00009 | : FE00D8 | 0866 | --- | EXECUTE | --- | @ | | 50 |
| -00008 | : FE00AC | ---- | ICF | ----- | FLH | & | | 75 |
| -00007 | : FE00AC | 5BD0 | --- | EXECUTE | --- | @ | | 50 |
| -00006 | : FE00AE | ---- | ICF | ----- | --- | & | | 75 |
| -00005 | : FE00AE | 0102 | --- | EXECUTE | --- | @ | | 50 |
| -00004 | : 000100 | ---- | IRA | ----- | --- | & | | 75 |
| -00003 | : 000100 | 00 | --- | EXECUTE | --- | @ | | 50 |
| -00002 | : 000100 | 12 | --- | EXECUTE | --- | @ | | 75 |
| -00001 | : ----- | ---- | --- | EXECUTE | --- | | | 50 |
| 00000 | : ----- | ---- | --- | THOLD | --- | | | 75 |

In this view each executed machine cycle is displayed.

Note that in some cases the emulator will stop one to two instructions *behind* the break point. The reason of this is, that the CPU instruction fetch queue has to be flushed first working independent from the break control.

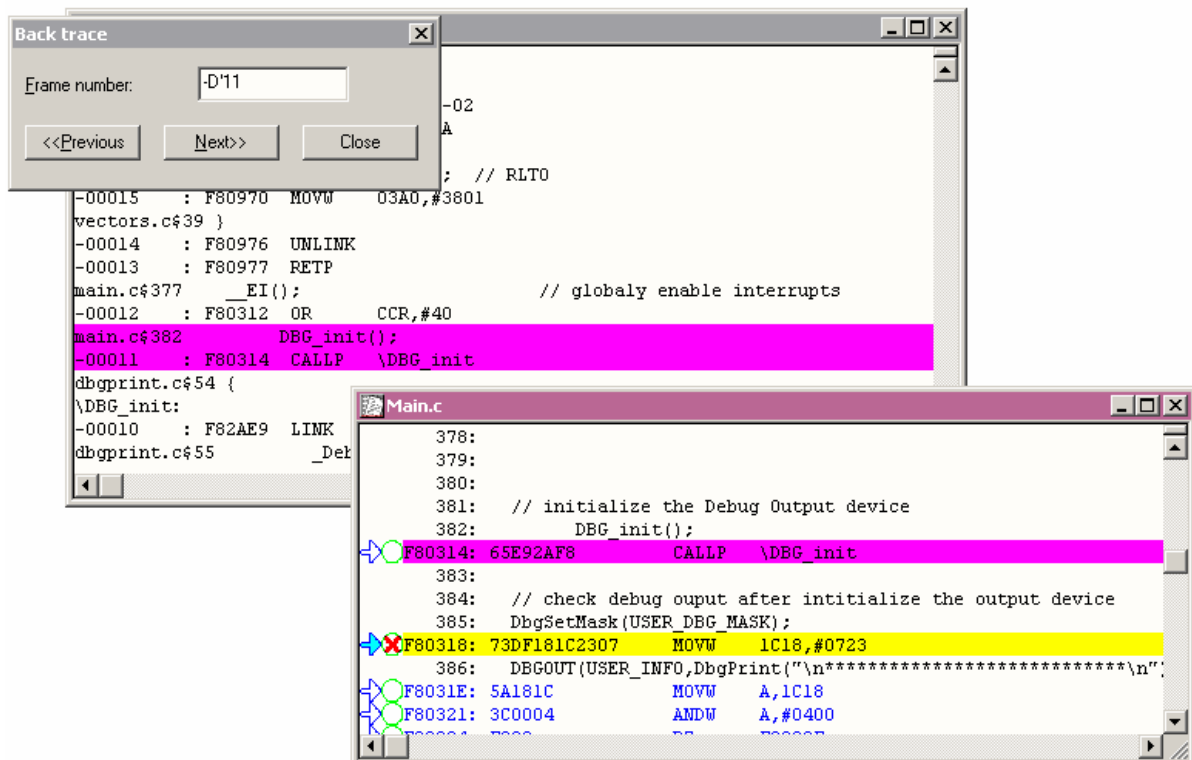
The minimum sample time of the emulator is 25ns. Because of each cycle is 62.5ns at 16 MHz CPU speed, the cycles are displayed as 50ns and 75ns. The deviation disappears when two cycles considered together: $62.5ns + 62.5ns = 50ns + 75ns = 125ns$.

5.1.3 Cycle View abbreviations:

| | | |
|-------------------------------|-----|---------------------------------|
| a-status Address Status | ICF | Instruction code fetch |
| | IRA | Internal read access |
| | IWA | Internal write access |
| | ERA | External read access |
| | EWA | External write access |
| Q-st Queue Status | FLH | Flushed Instruction Queue |
| | 1by | 1 Byte in Instruction Queue |
| | 2by | 2 Bytes in Instruction Queue |
| | 3by | 3 Bytes in Instruction Queue |
| Dfg Data Flag | & | address is available on the bus |
| | @ | data is available on the bus |

5.2 Back Trace

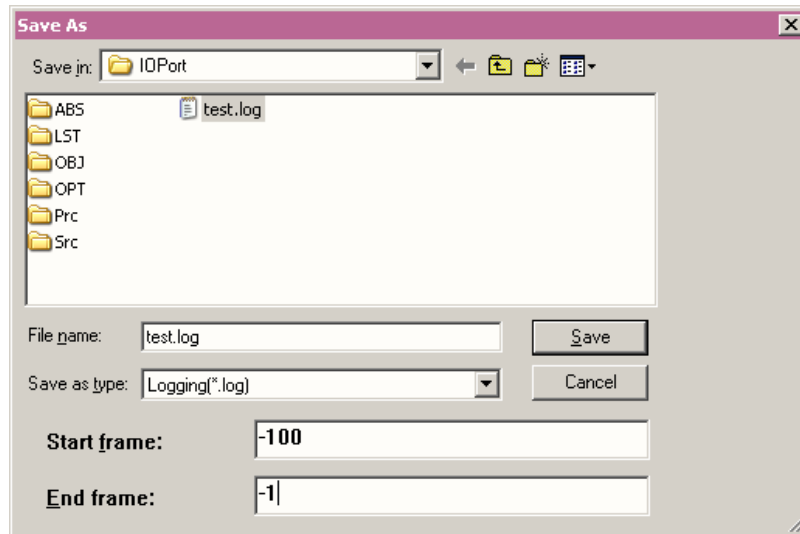
With the back trace functionality it is possible to step back and forward in the trace data. Together with the actual trace frame, the corresponding line in the source code module is highlighted (in violet):



The back tracing can be started by right-mouse click in the trace window. This functionality is available with Softune version V30L34R05 or higher.

5.3 Saving Trace Data

Since Softune version V30L34R05 it is possible to save the trace data beginning with a starting frame and ending with an end frame. Please enter negative numbers due to negative trace frames.



The save dialog is available by right-clicking in the trace window and choosing *Save file ...*.

5.4 Important Note:

Please be aware that most debug environment settings are only accessible, if the program execution is stopped – not during execution of a program.

6 Time Measurement

HOW TO USE TIME MEASUREMENT FUNCTION

6.1 Setup Time Measurement

Use the example from above and enter debugging mode. Open “Main.c” and use normal display (not *mixed display*). Set three breakpoints as shown below:

```

23: void main(void)
=>(x) 24: {
=>( ) 25:     InitIrqLevels();
=>( ) 26:     __set_il(7);           /* allow all levels */
=>( ) 27:     __EI();              /* globally enable interrupts */
28:
=>( ) 29:     a = 2;
=>( ) 30:     DDR0 = 0xFF;
31:
=>( ) 32:     while (1)
33:     {
=>( ) 34:         for (b = 0; b < 10; b++)
35:         {
=>( ) 36:             c = b * a;
=>( ) 37:             PDR0 = c;
=>(x) 38:             wait (5000);
=>(x) 39:         }
=>( ) 40:     }
=>( ) 41: }
    
```

Choose Debug→Time Measurement... and click on *Clear* if there are any time entries not equal to zero. Click on *Close* then.

Run the program. The CPU will stop at the beginning of the `main` function (1st breakpoint). This first stop initializes the time measurement counter.

Next, go on with execution. The CPU stops at the call to the `wait` function. Next, run the program till the third breakpoint.

Now choose Debug→Time Measurement.... You will find the following information:

| | |
|---|---|
| <i>From Initialize: 0h00m00s015ms704us525ns[Time]</i> | Time from start of <code>main</code> (1 st breakpoint) to return from <code>wait</code> (3 rd breakpoint) |
| <i>From Last Executed: 0h00m011ms568us125ns[Time]</i> | Time from call to <code>wait</code> to return from <code>wait</code> (2 nd breakpoint to 3 rd breakpoint) |
| <i>From Initialize: 193604[Cycle]</i> | see above |
| <i>From Last Executed 185094[Cycle]</i> | see above |

Note, the function `wait` needs about 11ms as stated in chapter 1.2.1.

6.2 Important Note:

Please be aware that most debug environment settings are only accessible, if the program execution is stopped – not during execution of a program.

7 Sequence

HOW TO USE THE SEQUENCE FUNCTION

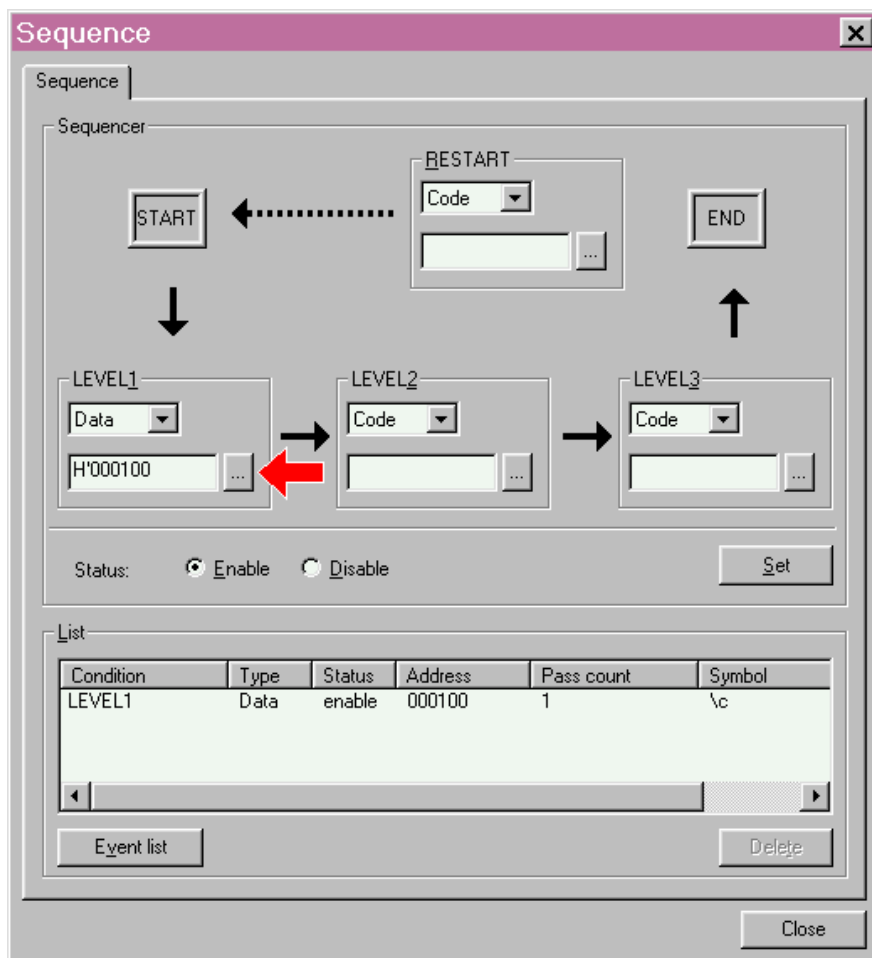
7.1 Setup Sequence function


The sequence function can be used to set complex breakpoints or a restart condition. Two kinds of sequential conditions can be set:

- Code: The execution (program counter register) meets a defined address (similar to “usual” breakpoints)
- Data: Defined data is written to or read from a defined address

Up to three sequential conditions (levels) can be set.

To use the sequence function enter debugging mode and choose *Debug*→*Sequence*. The sequence window will open and look like this:



Click on the pointed button ( , marked with a red arrow above) to open the sequence detail window:

Fill out in all text boxes the contents shown in the picture above. Note, that the Data and Data mask text boxes are only available if either “Data agreement” or “Data not” is selected.

The entries mean, that a level-1 break condition is reached, if for the first time (Pass count = 1) the Word “H’ 0010” is written to the address “H’ 000100”, which corresponds to the variable `c` of the example program.

Now click the OK button. Then click on **S**et in the sequence window! If you forget this point, Softune will not accept the entries and they will get lost.

Now start execution and the CPU will stop afterwards. The *mixed display* will look like this:

```

=> ( ) FE00BE: 5B0001      MOVW    0100,A
          37:          PDR0 = c;
=> ( ) FE00C1: 520001      MOV     A,0100
=> ( ) FE00C4: 5100        MOV     I:00,A
          38:          wait (5000);
=> ( ) FE00C6: 4A8813      MOVW    A,#1388
=> ( ) FE00C9: 4C          PUSHW  A
=> ( ) FE00CA: 658800FE      CALLP  \wait
=> ( ) FE00CE: 5D          POPW   AH
          39:          }
=> ( ) FE00CF: 735F0201      INCW   0102
=> ( ) FE00D3: 60DB        BRA     FE00B0
          40:          }
=> ( ) FE00D5: 60D5        BRA     FE00AC
          41:          }
=> ( ) FE00D7: 09          UNLINK
=> ( ) FE00D8: 66          RETP
    
```

Why this? The last executed instruction was “MOV I : 00 , A” and this is sure not an access to the address H’ 000100.

To understand this, let’s have a look at the *trace window*. Therefor choose View→Trace. In the trace window select via the right mouse button *I*nstruction. The following cycle list will be shown:

| Frame no. | address | mnemonic | time stamp |
|------------|----------|------------------------|------------|
| main.c\$39 | } | | |
| -00072 | : FE00CF | INCW 0102 | 175 |
| -00068 | : 000102 | internal read access. | 0007 250 |
| -00065 | : 000102 | internal write access. | 0008 200 |
| -00062 | : FE00D3 | BRA FE00B0 | 175 |
| -00056 | : FE00B0 | MOVW A,0102 | 375 |
| -00053 | : 000102 | internal read access. | 0008 200 |
| -00050 | : FE00B3 | MOVN A,#A | 175 |
| -00049 | : FE00B4 | CMPW A | 75 |
| -00048 | : FE00B5 | BGE FE00D5 | 50 |
| main.c\$36 | | c = b * a; | |
| -00045 | : FE00B7 | MOVW A,0102 | 200 |
| -00043 | : 000102 | internal read access. | 0008 125 |
| -00040 | : FE00BA | MULW A,0104 | 175 |
| -00035 | : 000104 | internal read access. | 0002 325 |
| -00022 | : FE00BE | MOVW 0100,A | 800 |
| -00019 | : 000100 | internal write access. | 0010 200 |
| main.c\$37 | | PDR0 = c; | |
| -00016 | : FE00C1 | MOV A,0100 | 175 |
| -00014 | : 000100 | internal read access. | 10 125 |
| -00011 | : FE00C4 | MOV I:00,A | 200 |
| -00009 | : 000000 | internal write access. | 10 125 |
| -00004 | : 000100 | internal read access. | 10 300 |
| 00000 | : ** | THOLD ** | 250 |

The determining C code instruction was `main.c$36`, frame `-0019`. The execution stops at the end of the next C code instruction. Please note, that the instruction fetch queue works independent from the sequencer control as well as the break control does. All instructions have to be flushed first before the break is performed.

In the Cycle view the event counter of the sequence is displayed (here highlighted in bold). In this view it is shown, that at the time when the event condition is met, there are 3 Bytes in the queue (3by).

| Frame no. | address | data | a-status | d-status | Qst | dfg | event | time stamp |
|-----------|----------|------|----------|----------|-----|-----|-------------------|------------|
| -00022 | : ----- | ---- | --- | EXECUTE | 4by | | | 50 |
| -00021 | : FE00C2 | ---- | ICF | ----- | --- | & | | 75 |
| -00020 | : FE00C2 | 0100 | --- | EXECUTE | --- | @ | | 50 |
| -00019 | : 000100 | ---- | IWA | ----- | --- | & | | 75 |
| -00018 | : 000100 | 0010 | --- | EXECUTE | --- | @ | | 50 |
| -00017 | : 000100 | 0010 | --- | EXECUTE | --- | @ | | 75 |
| -00016 | : FE00C4 | ---- | ICF | ----- | 3by | & | D 00000001 | 50 |
| -00015 | : FE00C4 | 0051 | --- | EXECUTE | --- | @ | | 75 |
| -00014 | : 000100 | ---- | IRA | ----- | --- | & | | 50 |
| -00013 | : 000100 | 00 | --- | EXECUTE | --- | @ | | 75 |
| -00012 | : 000100 | 10 | --- | EXECUTE | --- | @ | | 50 |
| -00011 | : FE00C6 | ---- | ICF | ----- | 2by | & | | 75 |
| -00010 | : FE00C6 | 884A | --- | EXECUTE | --- | @ | | 50 |
| -00009 | : 000000 | ---- | IWA | ----- | --- | & | | 75 |
| -00008 | : 000000 | 10 | --- | EXECUTE | --- | @ | | 50 |
| -00007 | : 000000 | 10 | --- | EXECUTE | --- | @ | | 75 |
| -00006 | : FE00C8 | ---- | ICF | ----- | --- | & | | 50 |
| -00005 | : FE00C8 | 4C13 | --- | EXECUTE | --- | @ | | 75 |
| -00004 | : 000100 | ---- | IRA | ----- | --- | & | | 50 |
| -00003 | : 000100 | 00 | --- | EXECUTE | --- | @ | | 75 |
| -00002 | : 000100 | 10 | --- | EXECUTE | --- | @ | | 50 |
| -00001 | : ----- | ---- | --- | EXECUTE | --- | | | 75 |
| 00000 | : ----- | ---- | --- | THOLD | --- | | | 50 |

7.2 Important Note:

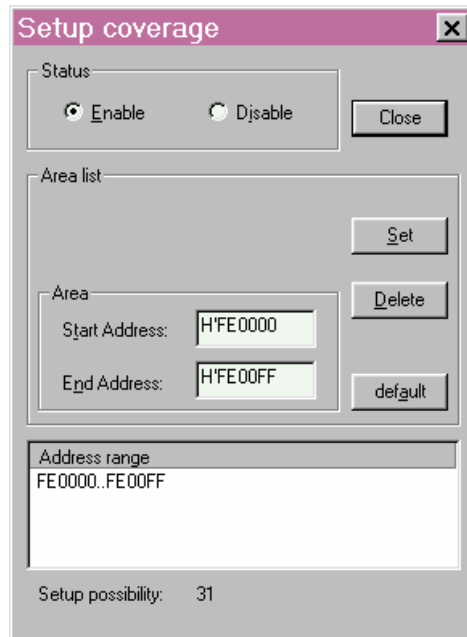
Please be aware that most debug environment settings are only accessible, if the program execution is stopped – not during execution of a program.

8 Coverage

HOW TO USE THE COVERAGE MEASUREMENT FUNCTION

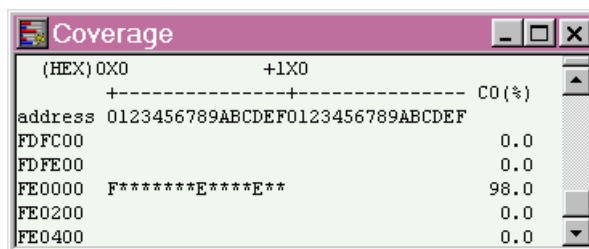
8.1 Setup Coverage

With this function it is possible to check which parts of the code were executed. To use the coverage measurement first choose *View*→*Coverage* and then click with the right mouse button into the *coverage* window and select *Setup...* . For the example program above type in the following address area:



Start Address: H' FE0000 and *End Address:* H' FE00FF. Then click on *Set* and *Close*.

Then execute the program for several seconds and halt it with *Stop execution*. Right-click into the *coverage* window and select *Refresh*. Then scroll to address H' FE0000. The following entries should occur (for the *16 address unit view*):



An asterisk “*” shows, that all 16 addresses were executed. Otherwise a hexadecimal entry is shown for the number of executed addresses or a dot “.” if not.

To see the code coverage in detail select *1 address unit view*:

| address | +00 | +01 | +02 | +03 | +04 | +05 | +06 | +07 | +08 | +09 | +0A | +0B | +0C | +0D | +0E | +0F | CO (%) |
|---------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|--------|
| FE0000 | . | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 93.8 |
| FE0010 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE0020 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE0030 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE0040 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE0050 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE0060 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE0070 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE0080 | - | - | - | - | - | . | . | - | - | - | - | - | - | - | - | - | 87.5 |
| FE0090 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE00A0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE00B0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE00C0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE00D0 | - | - | - | - | - | . | . | - | - | - | - | - | - | - | - | - | 87.5 |
| FE00E0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE00F0 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 100.0 |
| FE0100 | | | | | | | | | | | | | | | | | 0.0 |
| FE0110 | | | | | | | | | | | | | | | | | 0.0 |
| FE0120 | | | | | | | | | | | | | | | | | 0.0 |

A “-“ indicates an executed address, a “.” indicates no access.

In this example the addresses H’ FE0000, H’ FE0086-87, H’ FE00D7-D8 are not executed. This is o.k., because:

- H’ FE0000: Start.asm-NOP, not executed after Reset.
- H’ FE0086-87: BRA STARTUP\end of “Start.asm”
- H’ FE00D7-D8: UNLINK/RETP of “Main.c” not executed because of endless loop

The addresses above H’ FE00D8 are the interrupt vectors which are always covered using `InitIrqLevels(); (CALLP \InitIrqLevels)`.

Note that the coverage feature only can be used if it is selected by: Setup→Debug environment→Chip: Coverage: Enable. Also note that in this case real time memory displaying is only possible within the two selected real time memory areas.

8.2 Important Note:

Please be aware that most debug environment settings are only accessible, if the program execution is stopped – not during execution of a program.

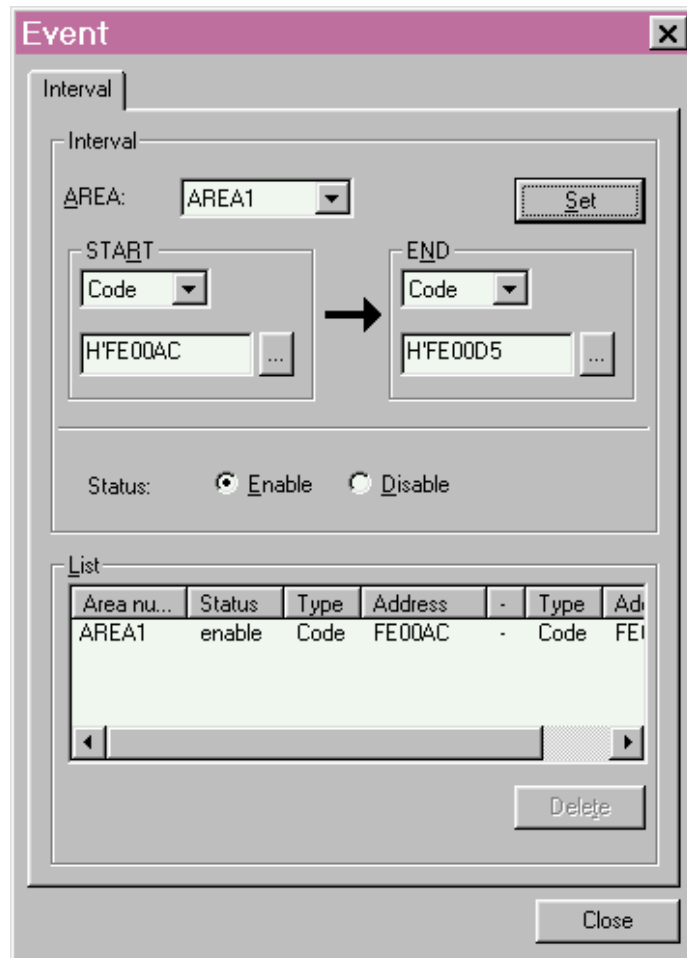
9 Performance Measurement

HOW TO USE THE PERFORMANCE MEASUREMENT FUNCTION

9.1 Setup Performance Measurement

Enter debugging mode and select via Setup→Debug environment→Debug environment ... the tab event. Then select in Event mode Performance and click on OK.

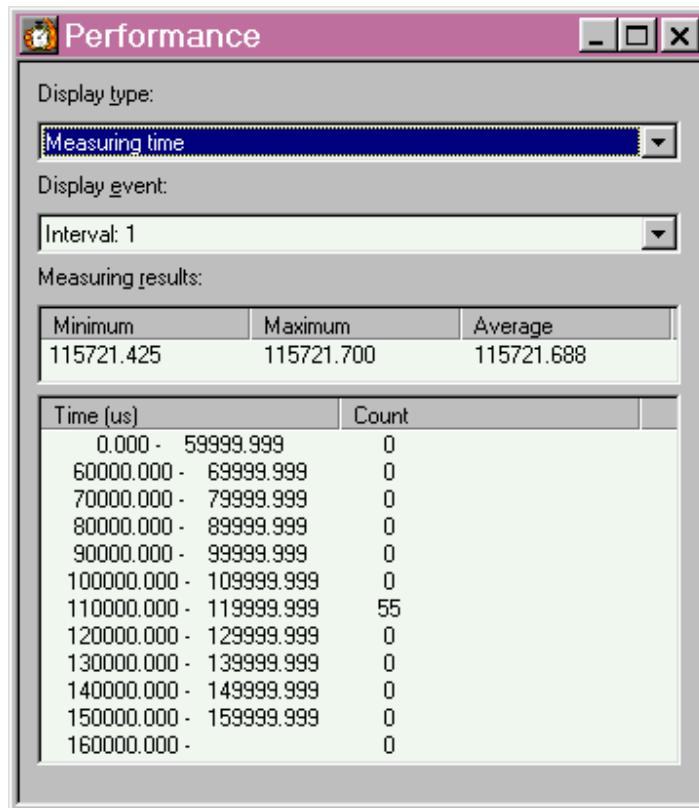
Then select Debug→Event. The following window will open:



Enter a start and an end address, e. g. FE00AC and FE00D5 (the begin and the end of the endless loop of the example code from above), and click on Set.

In the list below the new event interval for AREA1 will be displayed.

Now run the program for several seconds and then select View→Performance. The following window will open:



This analyze shows that the interval was executed 55 times with a minimum time of 0.115721425 s, a maximum of 0.1157217 s and a mean execution time of 0.115721688 s. Additionally the event count can be displayed. Up to 4 intervals can be analyzed.

9.2 Important Note:

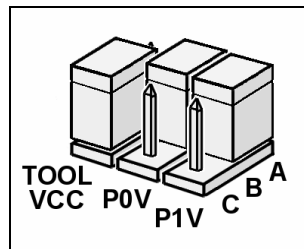
Please be aware that most debug environment settings are only accessible, if the program execution is stopped – not during execution of a program.

10 Power-On Debugging

HOW TO USE THE POWER-ON DEBUG FUNCTION

10.1 Preparation

The Jumper *TOOL VCC* (S1) of the adapter board has to be set to B-C position, so that the Evaluation MCU is powered by the emulator.



For Power-On debugging first enter the debug mode. Then check via *Debug*→*Run* that *Power On Debug* is **not** selected at this time.

Now run the project for some seconds and then stop the MCU.

10.2 Setting break points

After stopping the code, you can set break points. For the example above use the address `H'FE009B` – the entry point of the function *main*.

```

23: void main(void)
24: {
=> (x) FE009B: 0800          LINK    #00
25:     InitIrqLevels();

```

Because of, that this address is executed only at start up (after reset), the code execution can be continued without trapping into the break point.

Be careful setting break points for power on debugging. The power-on debug function can only be performed when the MCU is in *Run Mode*.

10.3 Enter Power-On Debug Mode

Now select *Power On Debug* via *Debug*→*Run*. After this, enter the low power voltage (e. g. 2 Volts), and continue the code execution.

Now turn off and on the power of your target board. The MCU will do a cold start, perform a reset sequence, and run till the set break point:

```

23: void main(void)
24: {
=> (x) FE009B: 0800          LINK    #00
25:     InitIrqLevels();

```

Now choose View→Trace and Cycle View and move to frame no. -08356. You will see the following:

| Frame no. | address | data | a-status | d-status | Qst | dfg | event | time stamp |
|-----------|----------|------|----------|----------|-----|-----|-------|---------------|
| -08356 | : ----- | ---- | --- | RESET | FLH | | | 1s240,582,300 |
| -08355 | : FFFFDC | ---- | ICF | ----- | --- | & | | 500 |
| -08354 | : FFFFDC | 0001 | --- | EXECUTE | --- | @ | | 500 |
| -08353 | : FFFFDC | 0001 | --- | EXECUTE | --- | @ | | 500 |
| -08352 | : FFFFDC | 0001 | --- | EXECUTE | --- | @ | | 500 |
| -08351 | : FFFFDC | 0001 | --- | EXECUTE | --- | @ | | 500 |
| -08350 | : FFFFDE | ---- | ICF | ----- | --- | & | | 500 |
| -08349 | : FFFFDE | 00FE | --- | EXECUTE | --- | @ | | 500 |
| -08348 | : FFFFDE | 00FE | --- | EXECUTE | --- | @ | | 500 |
| -08347 | : FFFFDE | 00FE | --- | EXECUTE | --- | @ | | 500 |
| -08346 | : FFFFDE | 00FE | --- | EXECUTE | --- | @ | | 500 |
| -08345 | : FFFFE0 | ---- | ICF | ----- | --- | & | | 500 |
| -08344 | : FFFFE0 | 0000 | --- | EXECUTE | --- | @ | | 500 |

The time displayed in frame -08356 is the time in which the power was turned off. In this time the emulator holds the MCU at *reset*.

After turning on the power at frame no -08355 the reset vector is fetched (address H' FFFFDC). Note, that the cycle time is 500ns which is related to 2 MHz, if a 4 MHz crystal is used.

This time changes to 50/75 ns (62.5ns) in frame no. -00054 in which the PLL factor was set in *start.asm*.

```

730: ;=====
731: ; 6.10 Wait for PLL to stabilise
732: ;=====
733:
734: #if CLOCKSPD > MAINCLOCK && CLOCKWAIT == ON
735: no_PLL_yet:
736:     BBS I:CKSCR:6,no_PLL_yet ; check MCM and wait for
=>( )FE007E: 6CA6A1FC     BBS I:A1:6,STARTUP\no_PLL_yet
737:                                     ; PLL to stabilize
738: #endif

```

| Frame no. | address | data | a-status | d-status | Qst | dfg | event | time stamp |
|-----------|----------|------|----------|----------|-----|-----|-------|------------|
| -00061 | : FE0084 | FE00 | --- | EXECUTE | --- | @ | | 500 |
| -00060 | : FE007E | ---- | ECF | ----- | FLH | & | | 500 |
| -00059 | : FE007E | A66C | --- | EXECUTE | --- | @ | | 500 |
| -00058 | : FE007E | A66C | --- | WAIT | --- | @ | | 500 |
| -00057 | : FE007E | A66C | --- | WAIT | --- | @ | | 500 |
| -00056 | : FE007E | A66C | --- | WAIT | --- | @ | | 500 |
| -00055 | : FE007E | A66C | --- | EXECUTE | --- | @ | | 300 |
| -00054 | : FE0080 | ---- | ECF | ----- | --- | & | | 75 |
| -00053 | : FE0080 | FCA1 | --- | EXECUTE | 2by | @ | | 50 |
| -00052 | : FE0080 | FCA1 | --- | WAIT | --- | @ | | 75 |
| -00051 | : FE0080 | FCA1 | --- | WAIT | --- | @ | | 50 |
| -00050 | : FE0080 | FCA1 | --- | WAIT | --- | @ | | 75 |
| -00049 | : FE0080 | FCA1 | --- | EXECUTE | --- | @ | | 50 |
| -00048 | : FE0082 | ---- | ECF | ----- | --- | & | | 75 |
| -00047 | : FE0082 | 9B65 | --- | EXECUTE | --- | @ | | 50 |

The tracing ends with the prefetch of `CALLP \InitIrqLevels`, the return address stack pushing, and the read access to address H' 000100 which should all be ignored.

Note that power-on debugging only works, if the used evaluation chip supports a power-on debug pin.

Note for Softune Version <V30L32:

If the code is once halted by break point, continuation will work, but *not* the power on debug function. To use it again you have to start a new debug session.

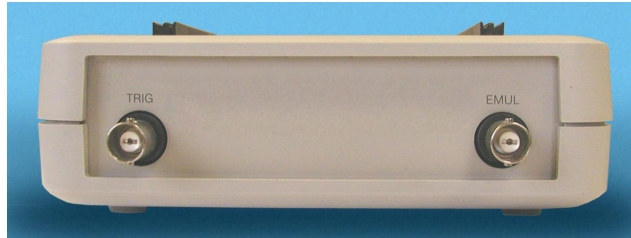
10.4 Important Note:

Please be aware that most debug environment settings are only accessible, if the program execution is stopped – not during execution of a program.

11 Trigger-Input and Emulator-Output

HOW TO USE THE TWO BNC CONNECTORS

11.1 The BNC Connectors



The MB2147-01 emulator has two BNC connectors. The left one is the “Trig”-Input and the right one the “Emul”-Output.

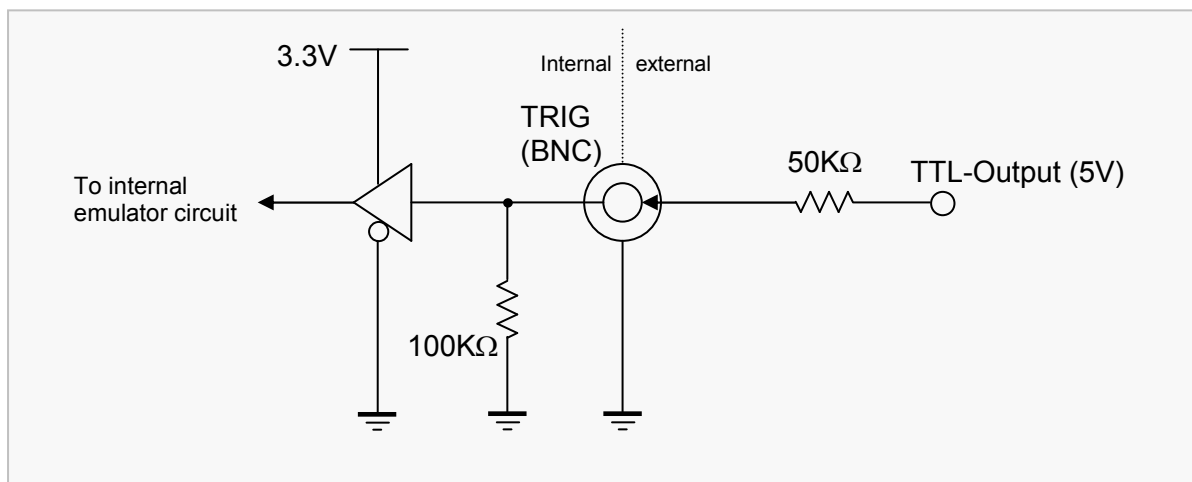
11.2 Trigger-Input

With this input an execution can be stopped. The Trigger-Input is a hardware “break point”.

A logical “high” (= 3.3V) on this input stops the execution in the debugging mode. Note, that because of internal latches and different clock speeds of the emulator and the MCU the termination is not immediately. The break slip is in a range of dozens to hundreds machine clock cycles.

The execution can be resumed after a triggered break.

Because of the 3.3V input and an internal 100K pull down resistor, it is recommended to use a serial 50K resistor, if a 5V signal is used:



11.3 Emulator-Output

The BNC-Output “EMUL” goes logical “high” (= 3.3V) if a program is executed and is “low” (= 0V) if the program is stopped or a break point has occurred. This signal can be used for controlling external hardware.

12 Appendix A

OVERVIEW OF DEBUG ENVIRONMENT

12.1 Debug environment settings

