AN205430

FR, All Series, Simulation With SOFTUNE Workbench V6

This application note will help you how to debug the application with Softune Workbench V6 Simulator. The simulator debugger simulates the MCU operations (executing instructions, memory space, I/O ports, interrupts, reset, etc.) with software to evaluate a program.

Contents

1 Introduction....2
2 Sample Program....3
2.1 Start new project with Softune Workbench....3
2.2 “Main.c”....3
2.3 Compiling “Main.c”....5
3 Debugging, first steps....6
3.1 Entering Debugger Mode....6
3.2 Using Bookmarks....6
3.3 Start Execution....7
3.4 Stop Execution....7
3.5 Reset MCU....7
4 Monitoring and Manipulating....8
4.1 Monitoring and Manipulating Processor Status....8
4.2 Monitoring and Manipulating CPU Registers....8
4.3 Monitoring and Manipulating Assembly Variables....10
4.4 Monitoring and Manipulating C Variables....11
4.5 Monitoring and Manipulating Memory....12
4.6 Symbol view....14
4.7 Local variables....14
5 Breakpoints....15
5.1 Setting Break points....15
5.2 Position of Break point....21
6 Trace....23
6.1 Trace Window....23
6.2 Trace View....25
6.3 Trace Jump....26
6.4 Back Trace....28
6.5 Search Trace....29
6.6 Trace Setup....31
6.7 Saving Trace Data....31
7 Memory Map....32
7.1 Memory Map Setup....32
8 Time Measurement....33
9 Call Stack....35
10 Function Call....36
11 Vector....38
11.1 Display and setting vectors....38
11.2 Jump....38
12 Interrupt Simulation....40
13 IO port Simulation....42
14 Debug environment setup procedure....46
14.1 Step Execution....46
14.2 Watch....47
14.3 Radix....48
14.4 Monitoring....49
14.5 Directory....50
14.6 Tab....50
14.7 Error output....51
14.8 Load....52
15 Additional Information....52
Document History....53
Worldwide Sales and Design Support....54

1 Introduction

This application note will help you how to debug the application with Softune Workbench V6 Simulator. The simulator debugger simulates the MCU operations (executing instructions, memory space, I/O ports, interrupts, reset, etc.) with software to evaluate a program.

It is used to evaluate an uncompleted system, the operation of single units, etc.

There are 2 types of simulator debuggers.

- Normal simulator debugger (normal)
- High-speed simulator debugger (fast)

This high-speed simulator provides substantial reductions in simulation time due to a dramatic review of normal simulator's processing methods.

The high-speed simulator debugger requires much more RAM space on the host PC than that of normal simulator debugger.

The required RAM size depends largely on your program size. For the required available RAM space, see the table below:

Basic Use	Fs911s.exe	20MB
Code size of target program	Per 64kB	6MB
Data size of target program	Per 64kB	1.5MB

For example if we consider target program size for CODE to be XX (KB) and for DATA to be YY (KB)

Then, Required RAM space (MB) = 20 + (XX / 64) * 6 + (YY / 64) * 1.5

However, RAM space larger than the above may be needed depending on program allocation. Allocate memory space consecutive areas should be reserved as much as possible.

Assuming program with 1 MB of CODE and DATA sizes

Required RAM space (MB) = 20 + (1024 / 64) * 6 + (1024 / 64) * 1 / 5 = 140MB

The simulator debugger simulates the MCU operations (instruction operations, memory space, interrupts, reset, low power-save mode, etc.) Peripheral I/Os, such as a timer, DMAC and serial I/O, other than the CPU core of the actual chip are not supported as peripheral resources. I/O space to which peripheral I/Os are connected is treated as memory space. There is a method for simulating interrupts like timer interrupts, and data input to memory like I/O ports.

2 Sample Program

Sample program for Debugging

2.1 Start new project with Softune Workbench

At first choose an evaluation MCU (here: MB91F467D), copy the template project of the “Softune samples” into an own folder (here: “Emulation_Test”) and start the Softune Workbench Software.

2.2 “Main.c”

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

```
/*                              SAMPLE CODE                                   */
/*----------------------------------------------------------------------------*/

/*************************@INCLUDE_START************************/
#include "mb91467d.h"
#include "vectors.h"
/*************************@INCLUDE_END************************/

/*********************@GLOBAL_VARIABLES_START*******************/
unsigned char Glob_var1;
unsigned short Glob_var2;
unsigned int Glob_var3;

/*********************@GLOBAL_VARIABLES_END*******************/

/******************@FUNCTION_DECLARATION_START****************/

void wait (unsigned short count)
{
      for(;count>0;count--)
      __asm(" NOP");
}

void initFRT0 (void)
{
      TCCS0 = 0x20;
}

void main(void)
{

      unsigned char Local_var1 = 0;
      unsigned short Local_var2 = 0;
      unsigned int Local_var3 = 0;


    __EI();                     /* enable interrupts */
    __set_il(31);               /* allow all levels */
    InitIrqLevels();            /* init interrupts */

    PORTEN = 0x3;               /* enable I/O Ports */
```

```
    While (1)                    /* endless loop */
    {
       HWWD_CL = 0;

       Glob_var2++;
       Glob_var3++;

       Local_var2++;
       Local_var3++;

       Local_var1 = PDR00;
             __asm(" NOP");
       PDR01 = Local_var1;


       wait(5000);
    }
}


/*******************@FUNCTION_DECLARATION_END*****************/


__interrupt void FRT0IRQHandler (void)
{
       TCCS0_IVF = 0;
       Glob_var1++;
}
```

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.

2.3 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 or 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----------

vectors.c

Start91460.asm

mb91467d.asm

MAIN.c

Now linking...

<Your path>\Simulation_Test\ABS\Template.abs

Now starting load module converter...

<Your path>\ Simulation_Test\ABS\Template.mhx

No Error.

3 Debugging, first steps

How to enter Debugging Mode

3.1 Entering Debugger Mode

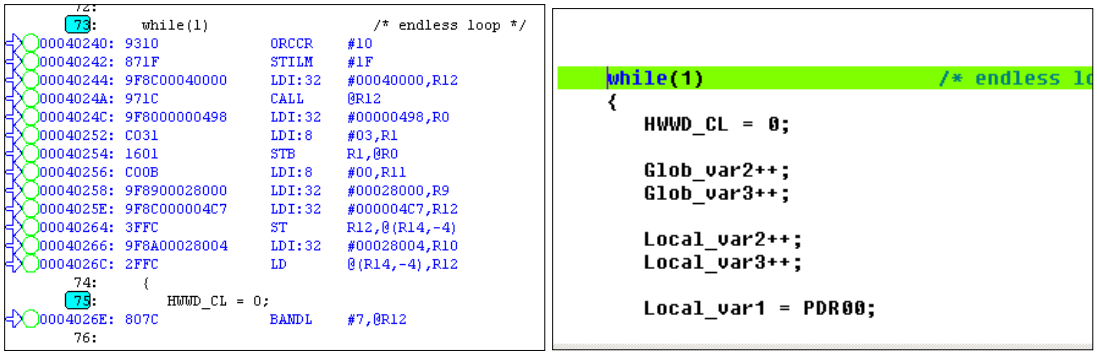
After successful compilation of the project start the debugging mode via simulation by double clicking on the Debug-"Simulator.sup"-entry in the workspace window. After successful connection to the target, reset MCU, open "Main.c" (close it first, if it is open) and then click on right mouse button and select "*Mix Display*". Your generated code should look like the following.

```
        6:   MAIN.C
        7:   - description
        8:   - See README.TXT for project description and disclaimer.
        9:
       10:   06.10.06  1.01   UMa    changed includes
       11: ------------------------------------------------------------------------*/
       12:
       13:
       14: /*************************@INCLUDE_START************************/
       15: #include "mb91467d.h"
       16: #include "vectors.h"
       17: /*************************@INCLUDE_END************************/
       18:
       19: /*********************@GLOBAL_VARIABLES_START*******************/
       20: unsigned char Glob_var1;
       21: unsigned short Glob_var2;
       22: unsigned int Glob_var3;
       23:
       24: /*********************@GLOBAL_VARIABLES_END*******************/
       25:
       26:
       27: /******************@FUNCTION_DECLARATION_START*****************/
       28:
       29: void wait (unsigned short count)
00040210: 1781             ST       RP,@-R15
00040212: 0F01             ENTER    #004
00040214: 1708             ST       R8,@-R15
00040216: 8B48             MOV      R4,R8
       30: {
       31:
       32:         for(;count>0;count--)
00040218: A808             CMP      #0,R8
0004021A: EC05             BLE      00040226
       33:         __asm(" NOP");
0004021C: 9FA0             NOP
0004021E: A1F8             ADDN2    #FFFFFFFF,R8
00040220: 97B8             EXTUH    R8
00040222: A808             CMP      #0,R8
00040224: EDFB             BGT      0004021C
       34:
```

3.2 Using Bookmarks

Since Softune version V60L06, 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.



3.3 Start Execution

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

Now the program is being executed. If you are using a target system with LEDs on Port00, you will see the LEDs flicker if FLASH-CAN-100P-340 Target board is used.

3.4 Stop Execution

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

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

3.5 Reset MCU

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

Note: This works only if the application is stopped (e.g. breakpoint, etc.)

4 Monitoring and Manipulating

How to monitor and manipulate CPU registers, variables and memory

4.1 Monitoring and Manipulating Processor Status

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

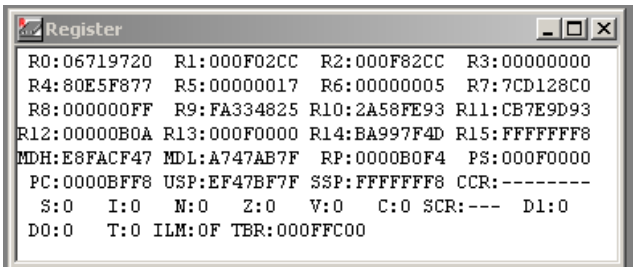
The flags are:

Abbr.	Flag Names
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:

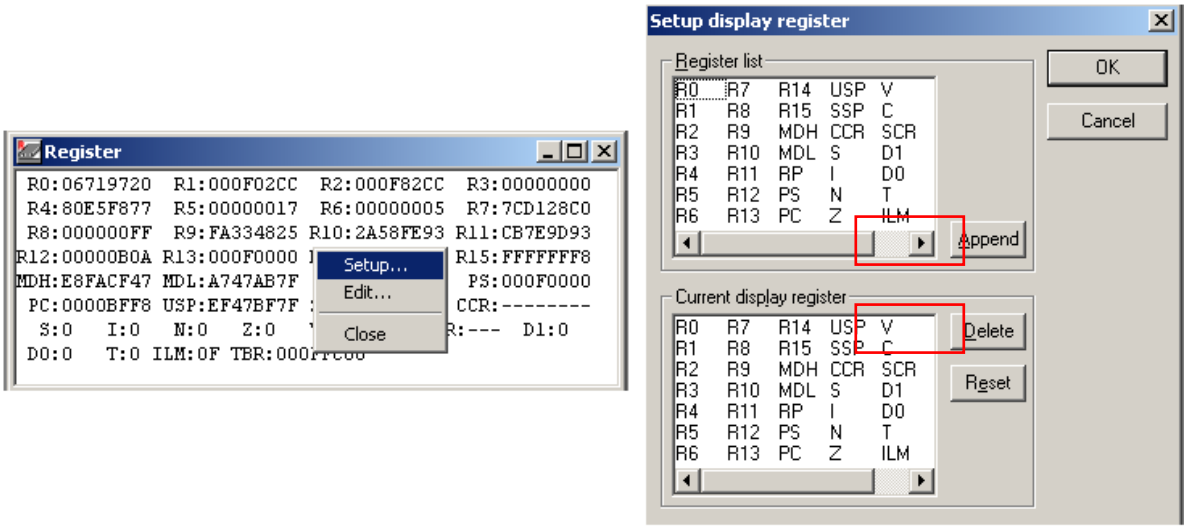
Abbr.	Flag Names
R0-R12	General purpose registers
R13	Virtual Accumulator
R14	Frame pointer
R15	Stack pointer
PC	Program counter
PS	Program status
TBR	Time Base register
RP	Return pointer
SSP	System stack pointer
USP	User stack pointer
MDH, MDL	Multiply-Divide register
TBR	Interrupt vector Table base register

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

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

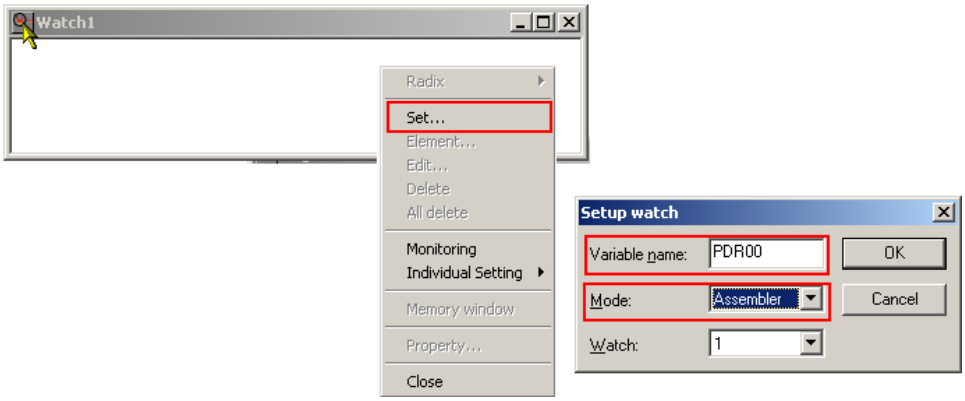
(beginning with “O’”).

Registers can be added or removed by right clicking on *Register* window and selecting *setup…*



4.3 Monitoring and Manipulating Assembly Variables

To display assembly 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 *Setup watch* will appear.



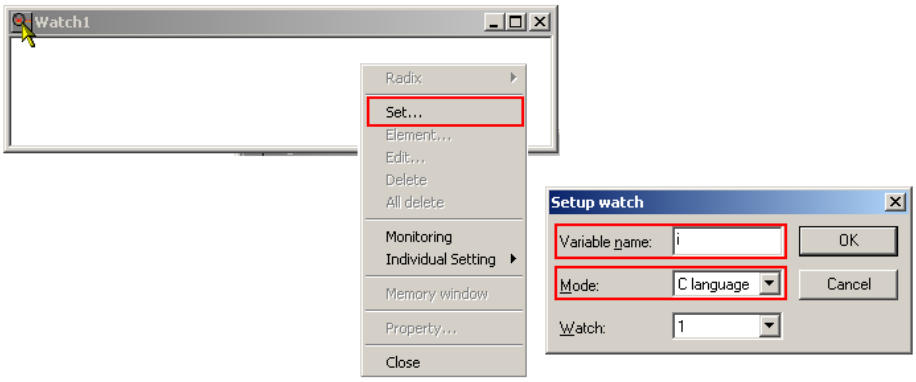
Under *Variable name* option, one can enter the variable name of the assembly program. The Mode must be *Assembler* in this case. The Watch window will then contain the variable name and value. If we select other than this then watch window will show variables memory location and not the data contained at that location.

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’”.

4.4 Monitoring and Manipulating C Variables

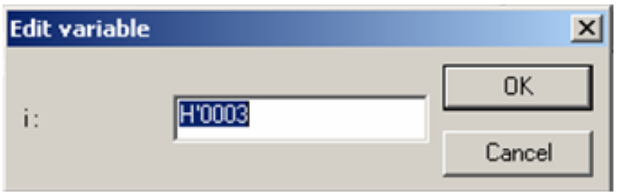
To display C variables choose in the debugging mode: *View -> Watch -> Watch1*. A new window *Watch* will occur. Click in this window on the right mouse button and select *Set…* A pop-up window *Setup watch* will appear.



Under *Variable name* option one can enter the variable name of the C program. The Mode must be *C language or Automatic* 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.

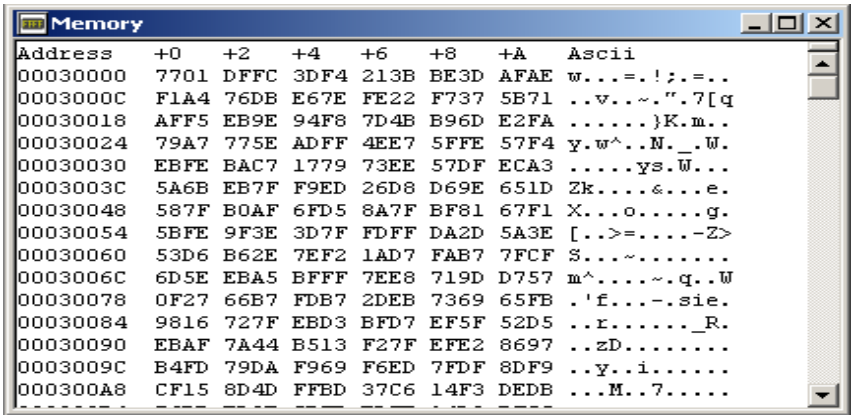
To manipulate the value just double-click on the entry and enter in the pop-up window *Edit variable* a new value. The radix can be chosen via “D’”, “H’”, “B’”, or “O’”.



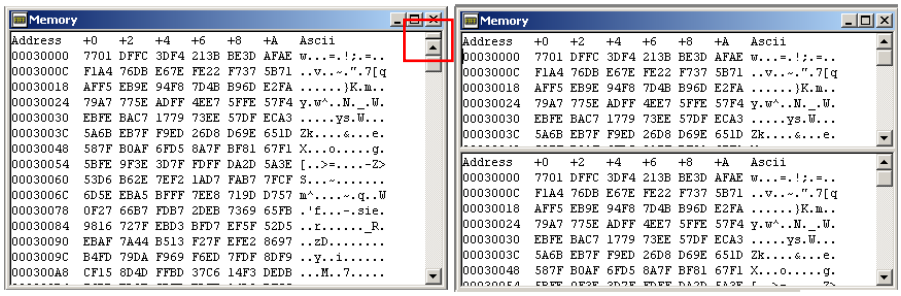
As explained in previous chapter to view memory content of Special Function Register, one should select *Assembler* under *Mode* option.

4.5 Monitoring and Manipulating Memory

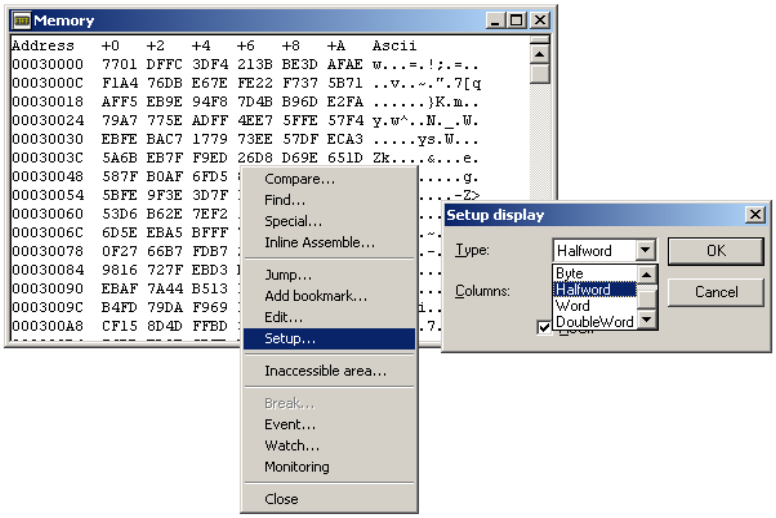
To display the MCU memory choose in the debugging mode: *View -> Memory*. A pop-up window *Memory* will occur and ask for the start address to be displayed. Type for instance H'2530 (or just 2530) for the RAM area. Following window occur which is something like a "Hex-Editor":



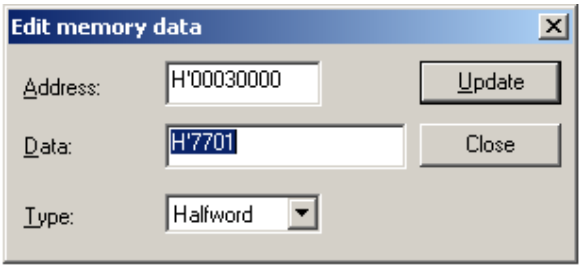
Memory window can be split. Move pointer near to top right corner, pointer will change to  ; drag it down with right mouse button pressed to split.



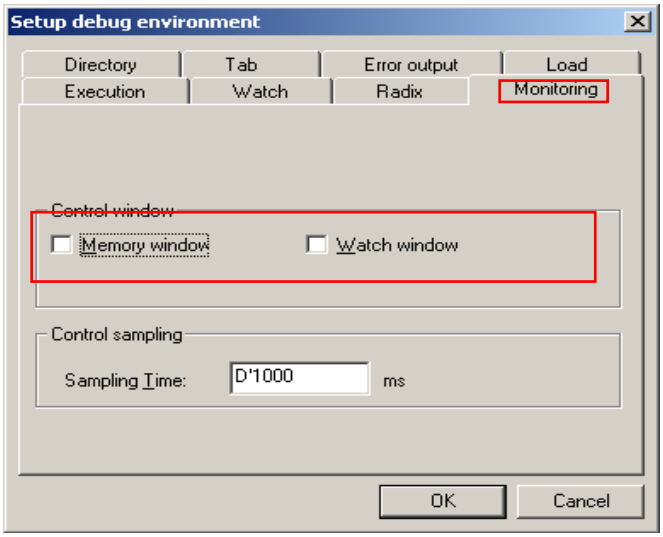
Also view can be set up to see data in bit, byte, word or long. Right click on memory window, click on setup. On a *Setup Display* dialog box select bit, byte, word or long from drop down menu.



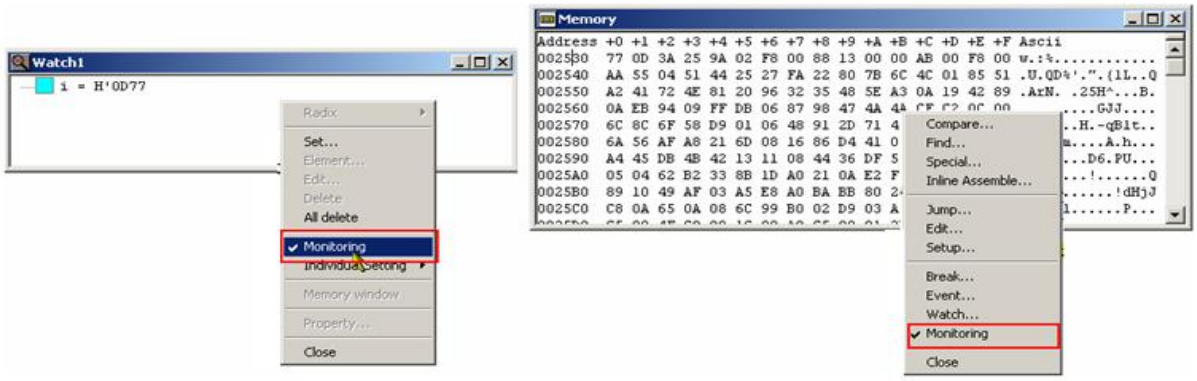
To change a memory content just double click on the respecting byte and *Edit memory data* dialog window pops up. In this window one can specify the address (default is the address of the clicked byte) and the new value. The value can be entered in hexadecimal, decimal, binary or octal format.



To see the memory in “real time” during execution, choose *Setup -> Debug environment -> Debug environment...* Then select the *Monitoring* tab and enter “D’1000” at *Sampling Time* and under *Control window* option click on check box *Memory window* and *Watch window*



Now, when the program is executed, *Watch* and *Memory* window is updated at 1000 ms, and one can see the addresses H’2530 in *Memory* window and variable ‘i’ in *Watch* window changing its values. Alternately one can select the same functionality by right clicking on *Watch/Memory* window, and on popup menu clicking on *Monitoring*

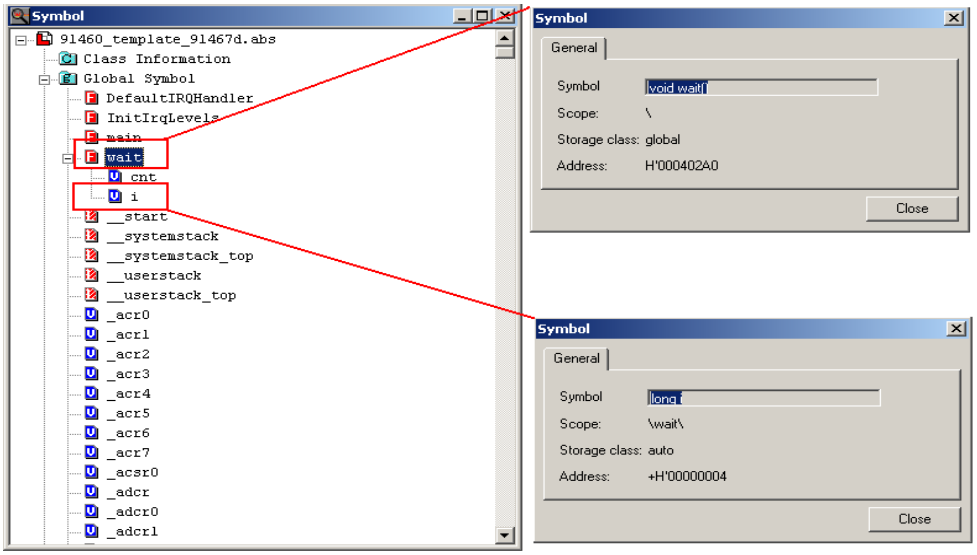


4.6 Symbol view

If one want to know where a variable is located in the memory then one 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:



Icon Reference

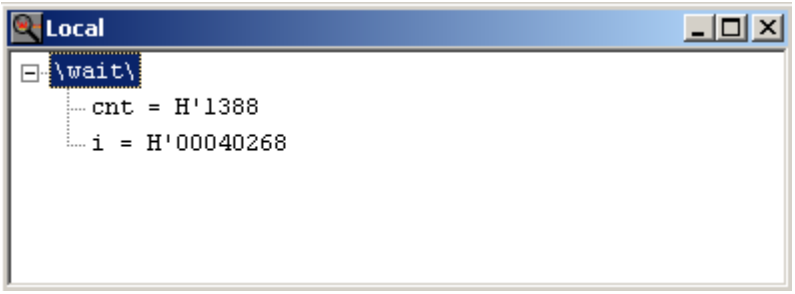
The following icons are used:

Icons	Flag Names
	Function
	Variable
	Label

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.



5 Breakpoints

How to set Break Points

Two types of break points are possible. Code break point and Data break point.

Code Break Point:

When code break point is set, program execution stops when the PC passes the break address (when instruction at that address is executed).

Code break point can be set with *pass count* from 1 to 65535 and *address mask*.

Pass count: It is the count of times PC needs to pass the set address before program execution stops at the break point.

Data Break Point:

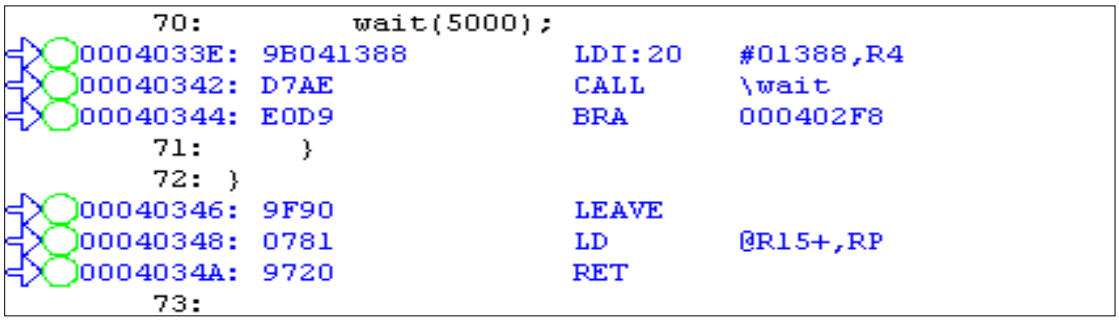
When Data break point is set, program execution stops when the data at break address is accessed.

As explained in hardware code breakpoint it is possible to set data break point with *pass count*.

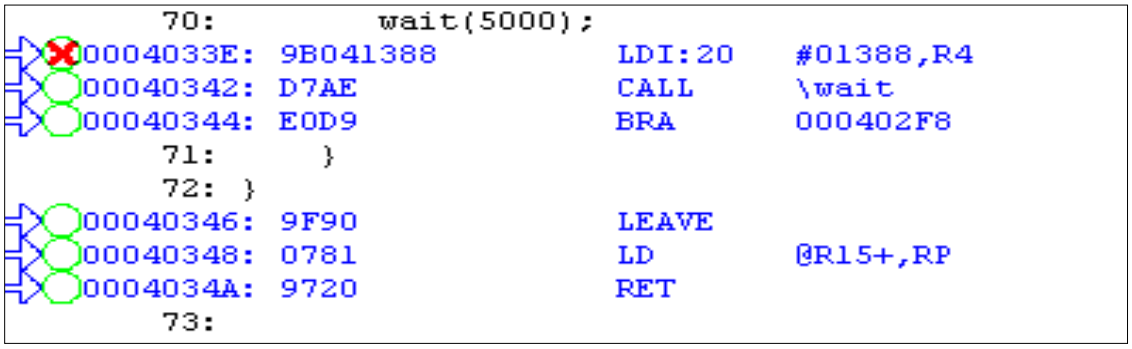
5.1 Setting Break points

5.1.1 Setting code break point through editor window

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



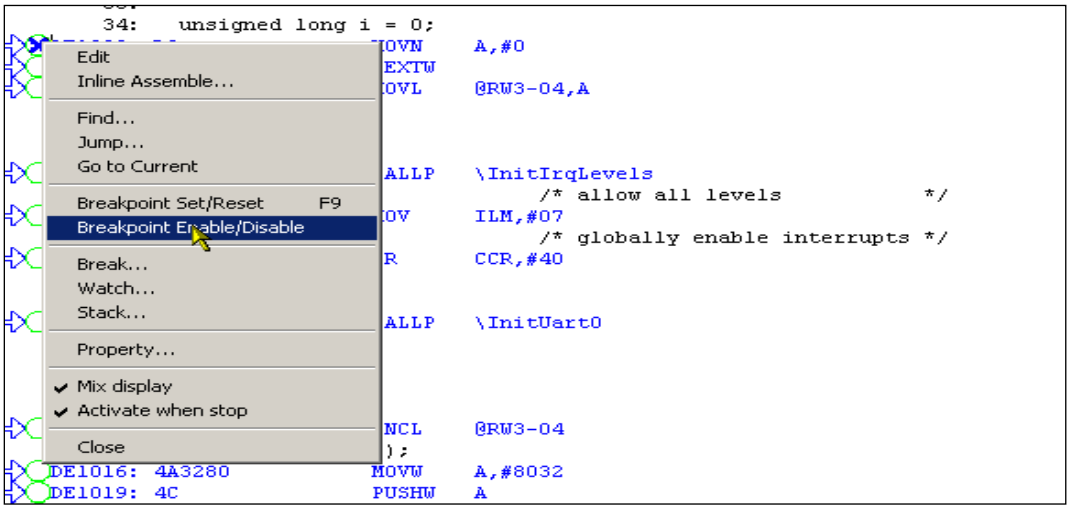
In these lines clicking into the circle can set a breakpoint or right click into the circle, click on Break Point Set/Reset. The symbol then turns to



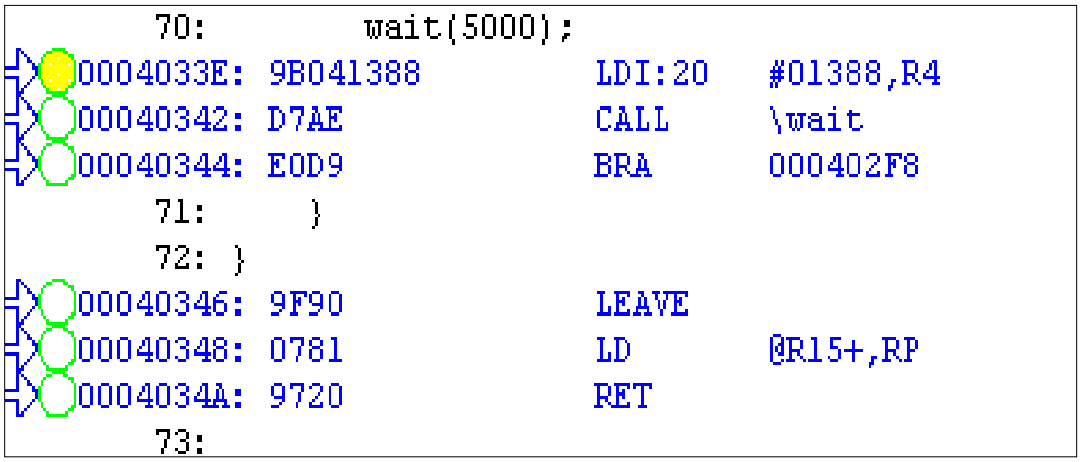
If you now start the execution, the CPU will halt on this break point. The actual line gets a yellow background color.

Clicking to this circle again or right clicking into the circle and clicking on Break Point Set/Reset, releases the break point.

Alternately, break point can be disabled by right clicking and on popup menu clicking Break Point enable/Disable.



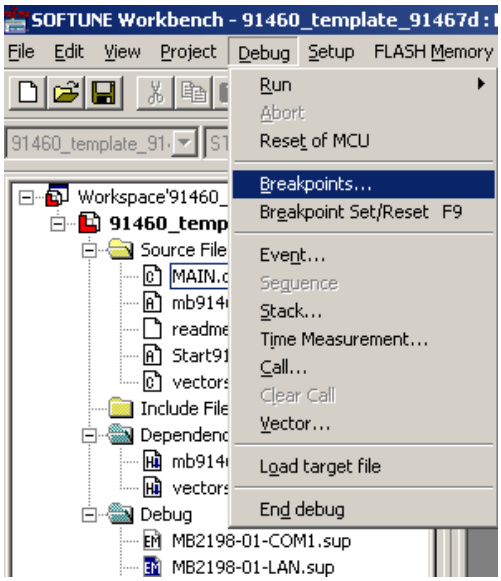
The symbol will change to



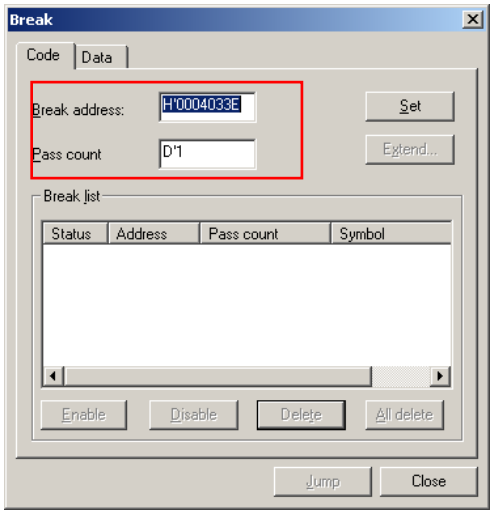
5.1.2 Setting code break point using Dialog box

A code break point can also be set using Dialog box

Open break point dialog box by clicking on *Breakpoints…* on Debug menu.

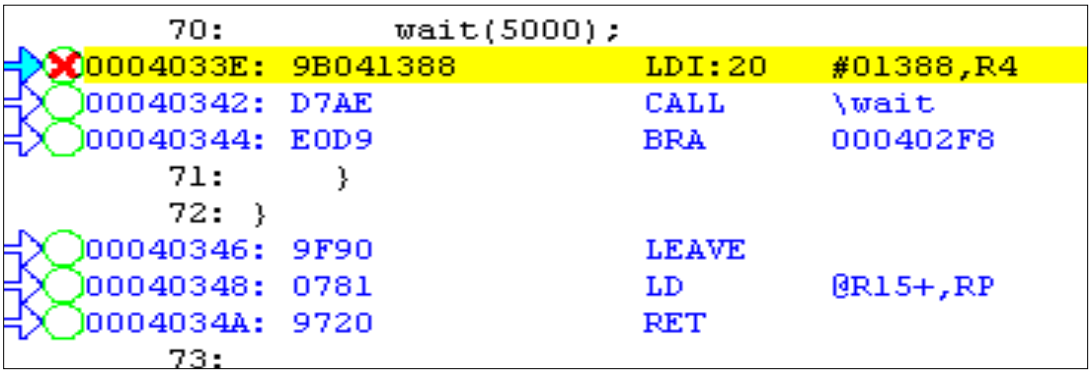


Type desired break address



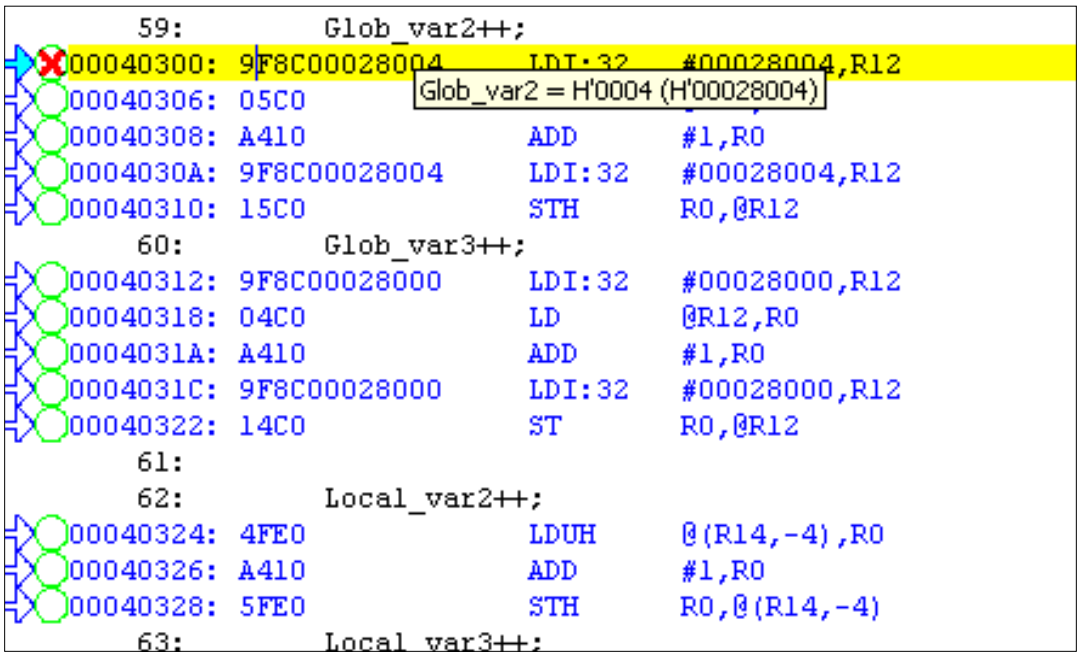
Click on *set,* to set breakpoint at required address

If you now start the execution, the CPU will halt on this break point. The actual line gets a yellow background color.

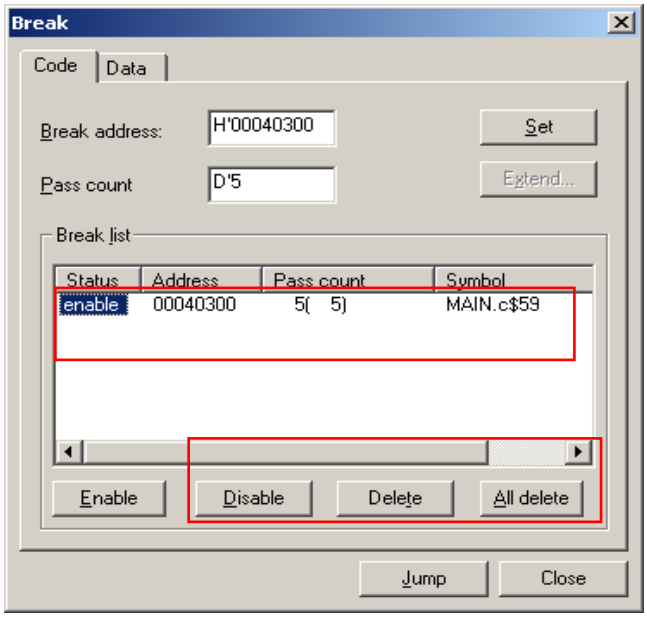


Similarly when we select Pass count, program will halt executing at code break point when program counter executes instruction at the address selected in the *break address* field, as many numbers of times as that is selected by *Pass count*.

For example, if we set code break point at *Break Address* H'40300 and select Pass count equal to D'5. When program is run, it halts due to hardware code break point when value of variable 'Glob_var2' equal to D'4

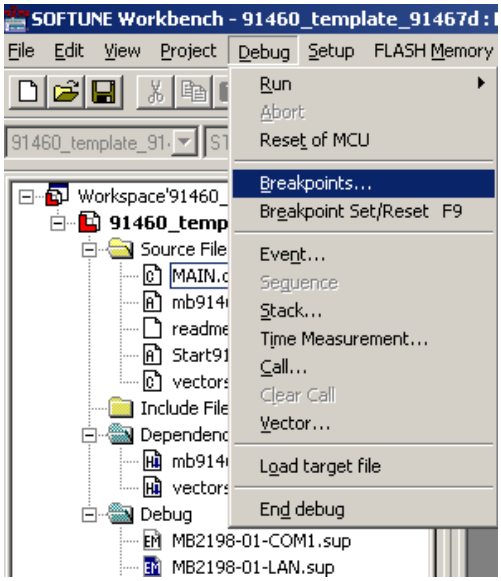


Breakpoint can be released by selecting breakpoint from *Break list,* and clicking on *Delete* button, alternately it can also be disabled by clicking *Disable* button

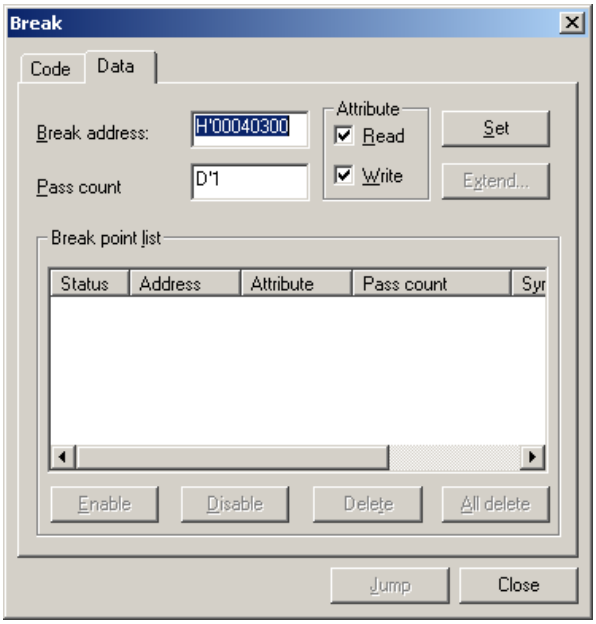


5.1.3 Data Break point

As shown below, *Break* dialog box is opened by clicking on *breakpoints…* option on *Debug* pull down menu.

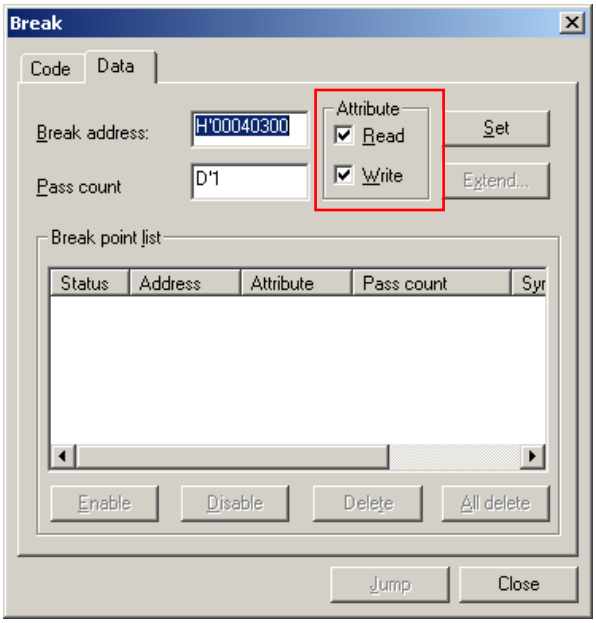


On *Data* tab, clicking on Details button will open *Data break point details dialog* box.



Pass count feature work same as that of code break point.

In *Data break point* dialog box, under *attribute* option it can be specified if *read* or *write* access at address should cause a break.

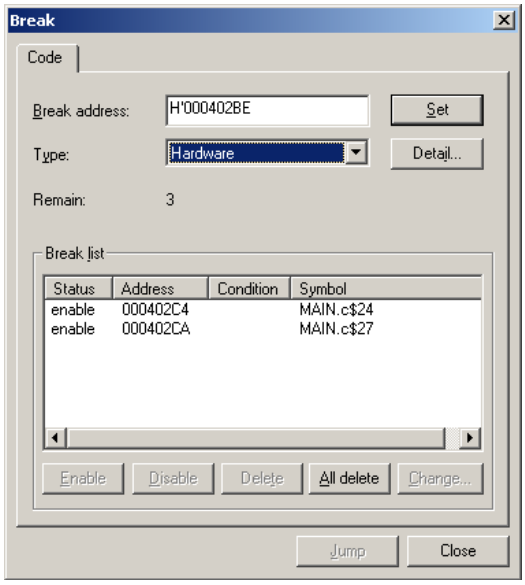


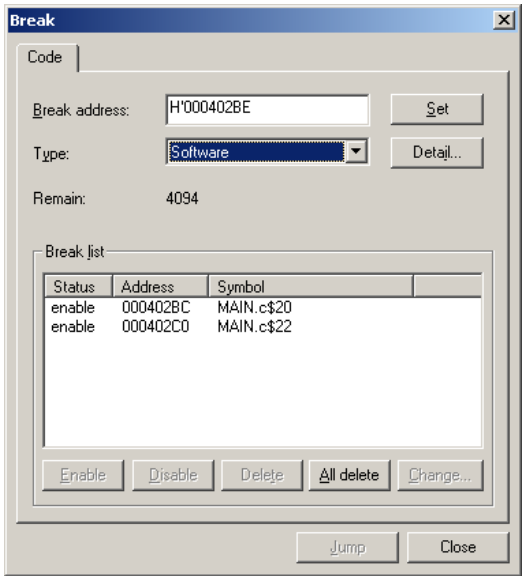
5.2 Position of Break point

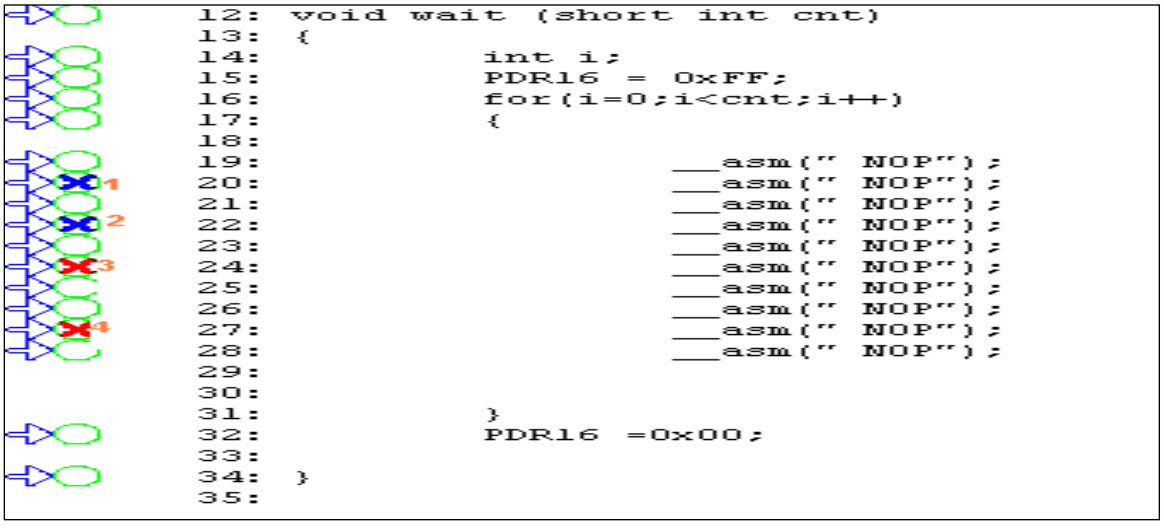
When break points are set to some location and after that code is modified and build again, position of previously set break point will be decided as per following rule

- Break point will be set to the same source code line number
- If it is not possible to meet above condition, it will be set to the same address location

For example,

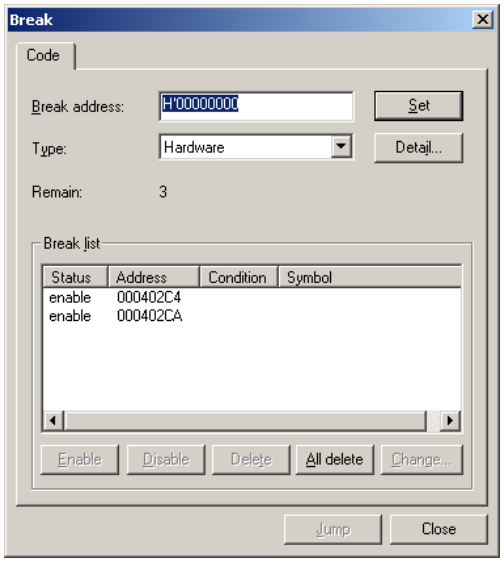


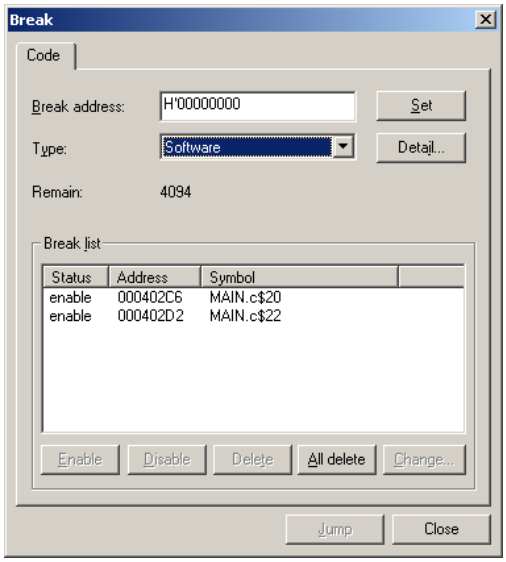


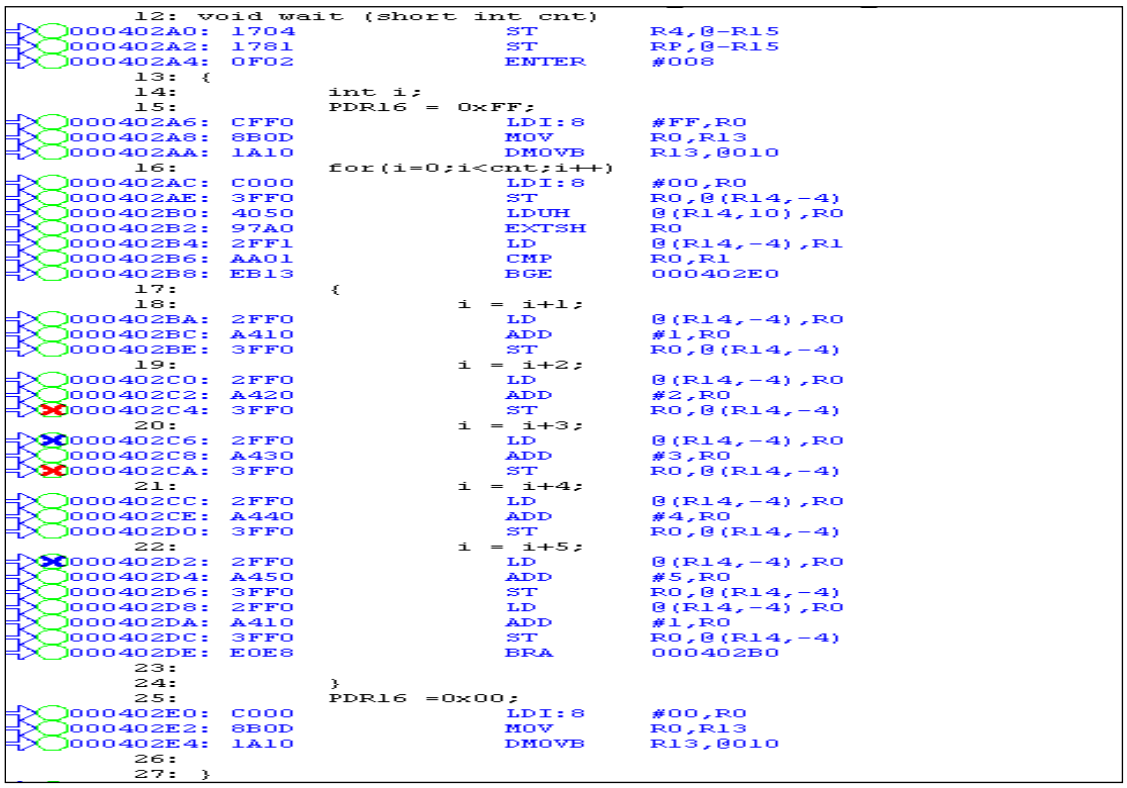


Let us change the code, replace couple of `__asm( "NOP");` instruction with some other instruction in *wait* function.

And let us assume that Breakpoint position changes as shown below







For Software breakpoint 1, this was at source code line no. 20, since after build there is valid instruction at that line; breakpoint is maintained at same source code line main.c$20

For Software breakpoint 2, this was at source code line no. 22, since after build there is valid instruction at that line; breakpoint is maintained at same source code line main.c$22

For Hardware breakpoint 3, this was at source code line no. 24, since after build there is no valid instruction at that line; breakpoint is maintained at same memory location H' 000402C4

For Hardware breakpoint 4, this was at source code line no. 27, since after build there is no valid instruction at that line; breakpoint is maintained at same memory location H' 000402CA

6 Trace

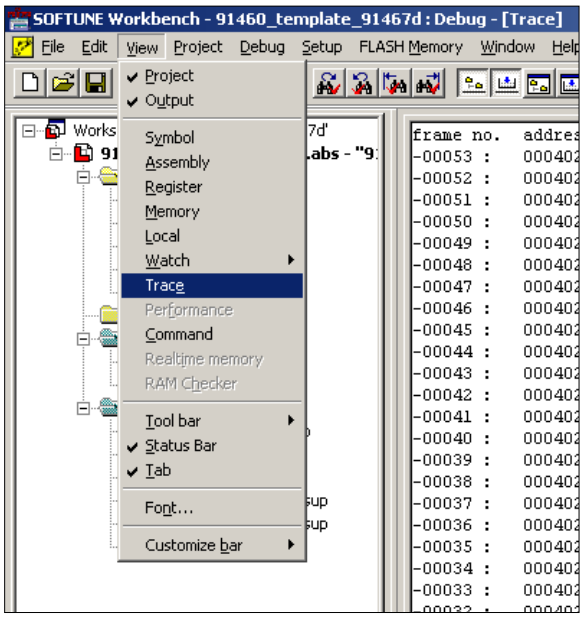
How to use the trace buffer

Trace function displays addresses and instructions executed so far. While execution of a program, the address, data and status information can be sampled and stored in the trace buffer. This function is called real-time trace.

In-depth analysis of a program execution history can be performed using the data recorded by real-time trace.

6.1 Trace Window

Trace window can be opened by clicking *Trace,* on *View* pull down menu.




```
-00013 :   00040370  LD       @R15+,RP
-00012 :   00040372  LDM1     (R12,R13)
-00011 :   00040374  LD       @R15+,R0
-00010 :   00040376  RETI
-00009 :   000402A6  LDUH     @(R14,10),R0
-00008 :   000402A8  CMP      #0,R0
-00007 :   000402AA  BLE      000402B6
-00006 :   000402B6  LEAVE
-00005 :   000402B8  LD       @R15+,RP
-00004 :   000402BA  ADDSP    #4
-00003 :   000402BC  RET
-00002 :   00040344  BRA      000402F8
-00001 :   000402F8  LDI:32   #000004C7,R0
 00000 :   000402FE  BANDL    #7,@R0
```

In a Trace result display, the first column shows the frame number.

The second column shows the internal bus address.

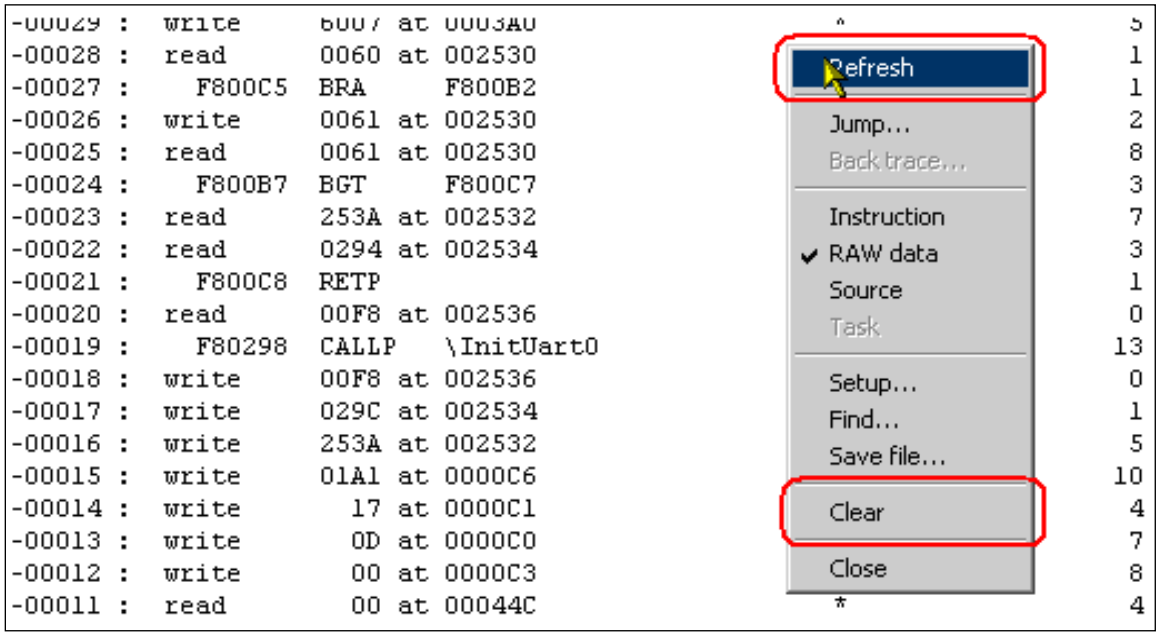
The third column shows either the disassembled machine code (mnemonic) or a data transfer (inclusive data value).

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

In this view the executed instructions are shown, not fetched instructions.

Every time, after program run, to refresh a trace window right click on Trace Window, on a popup menu click on *Refresh*.

To clear previous data, right click on Trace Window, on a popup menu click on *Clear*.

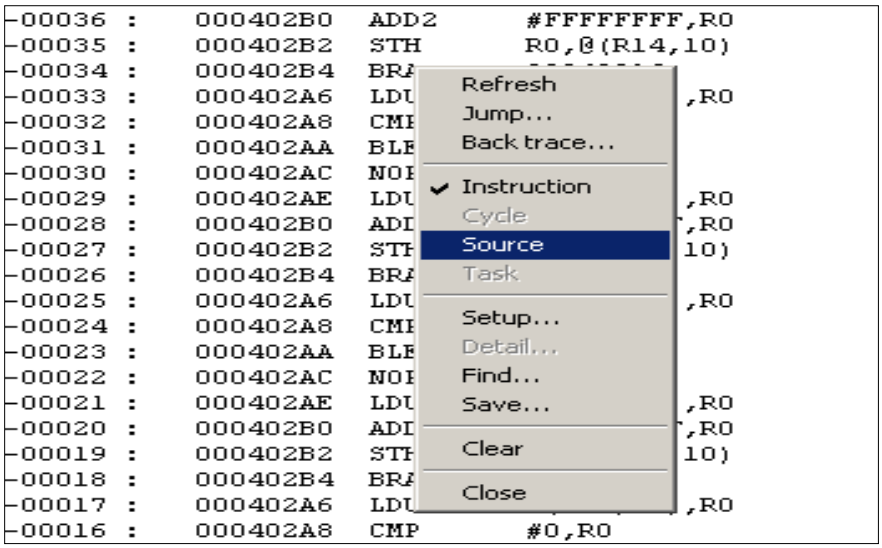


6.2 Trace View

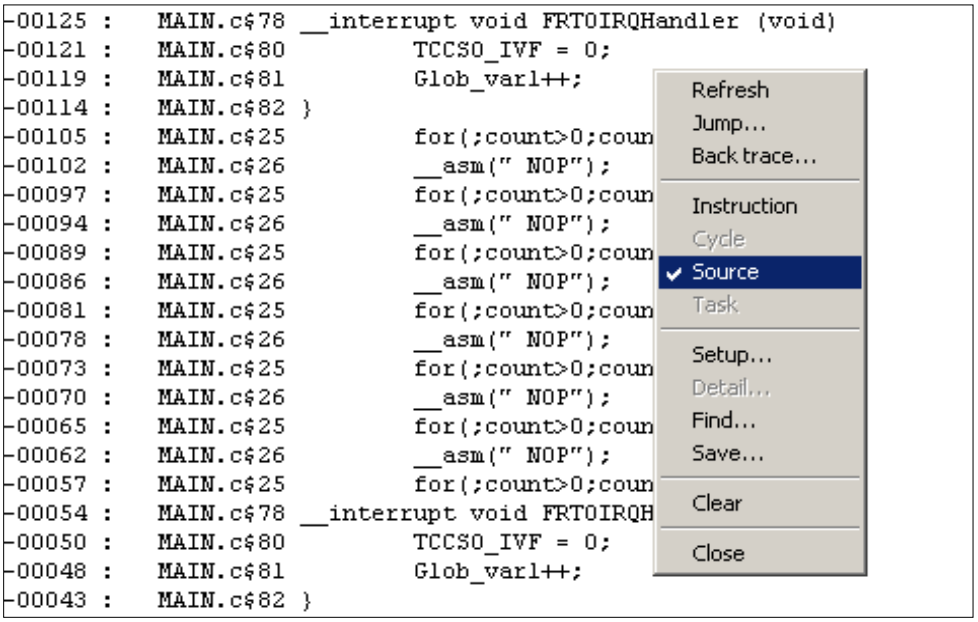
Trace view can be changed among Instruction or source view.

For Instruction view, right click on Trace window, on a popup menu click on *Instruction*.

Instruction view show instruction cycles using assembler mnemonics

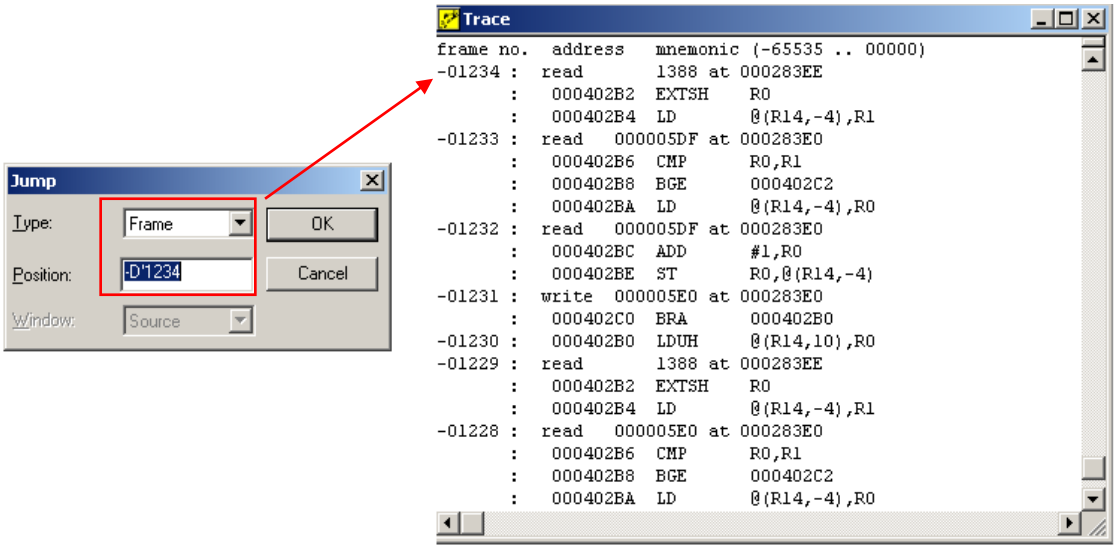


For Source view, right click on Trace window, on a popup menu click on *Source*. Source view shows program source code (e.g. C - language)

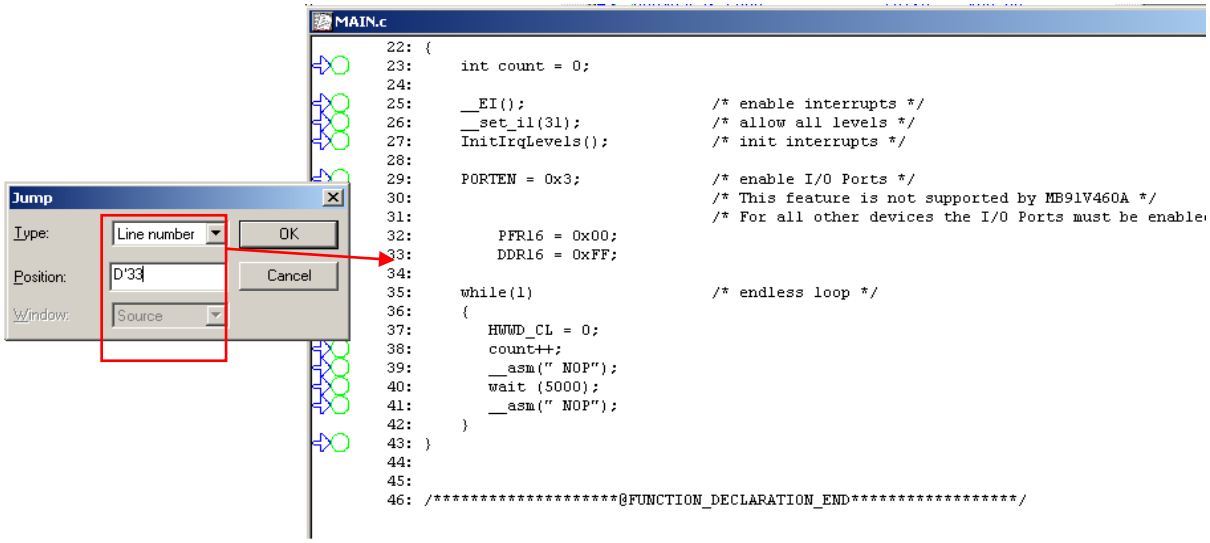


6.3 Trace Jump

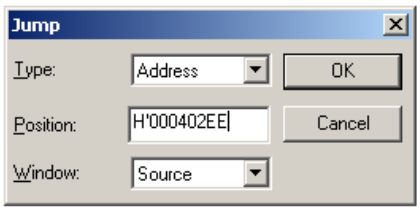
To move cursor to particular frame in trace window, right click on *trace* window, click on *Jump…,* on *Jump* dialog box, select *frame* under *Type* option and desired frame number under *Position* option and click *OK*

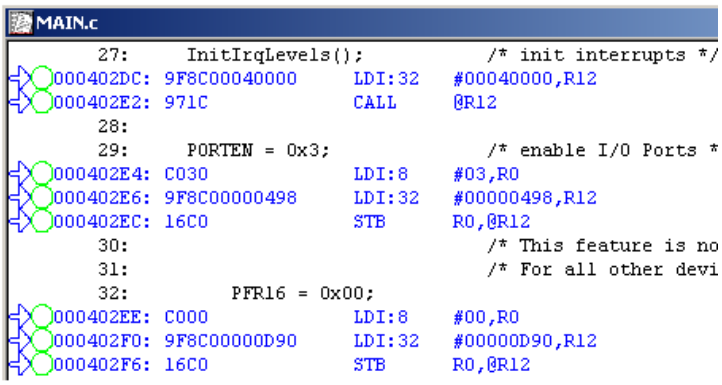


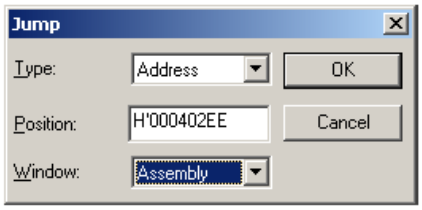
To view source code at particular line number, right click on trace window, click on *Jump…*, on *Jump* dialog box, select *Line number* under *Type* option and source line number under *Position* option, click *OK,* in editor window one can view source code at selected line number

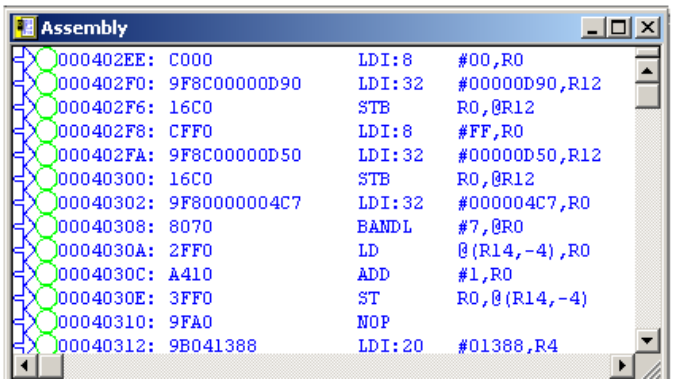


To view assembly instruction at particular memory location, right click on trace window, click on *Jump…*, on *Jump* dialog box select *Address* under *Type* option and memory location under *Position* option and then select either *Source* or *Assembly* under *Window* option.

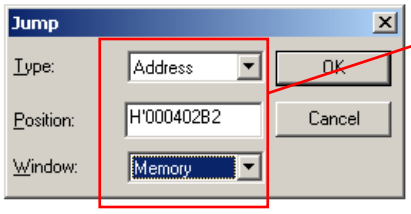


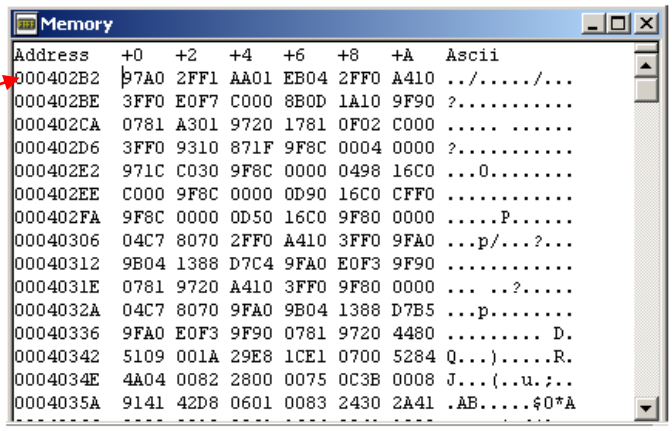






To view data at particular memory location, right click on trace window, click on *Jump…*, on *Jump* dialog box select *Address* under *Type* option and memory location under *Position* option and then select memory under *window* option.





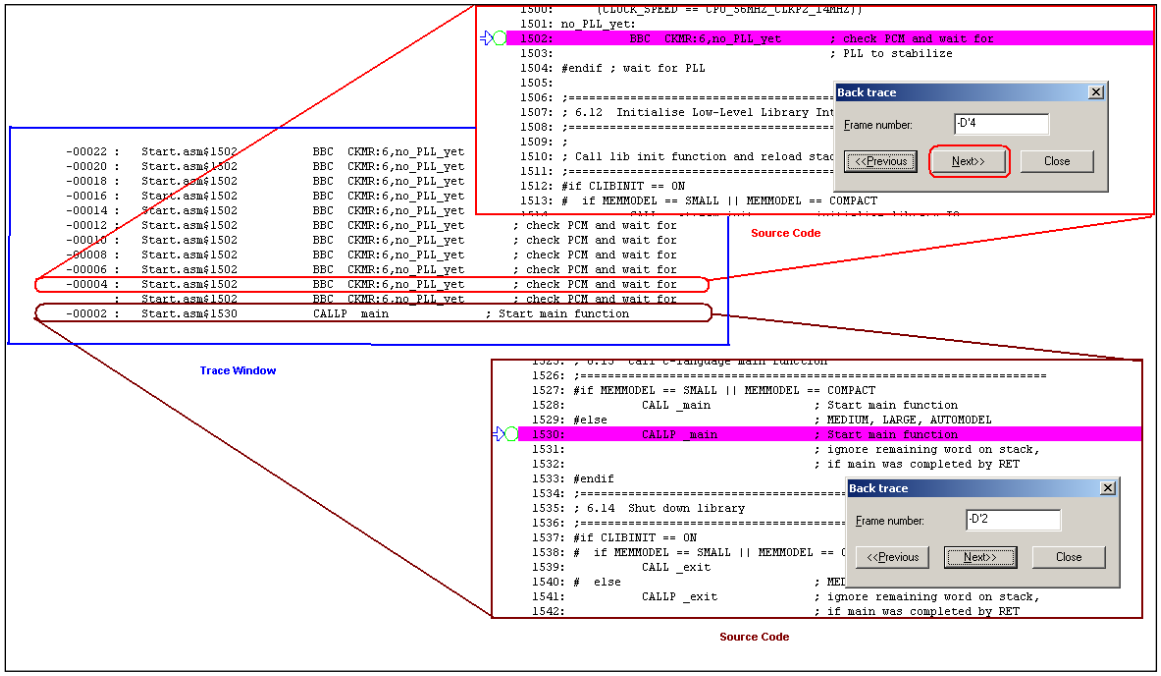
6.4 Back Trace

When selected *Source view,* one can use *Back trace…* functionality. Right click on editor window, click on *Back trace…, Back trace* dialog box will appear

With this functionality, one can traverse back and forth in source window in a same sequence, in which source code is executed. Source code at particular frame number will be highlighted by pink background while traversing.

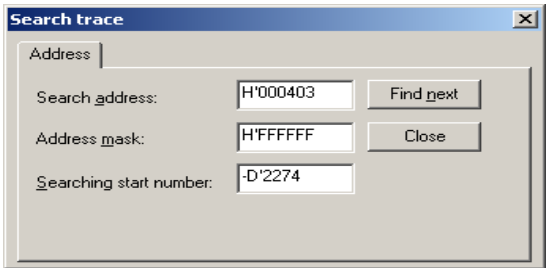
In figure below, trace window shows address and instruction at frame number -D'2 and

-D'4 executed in sequence. In a *back trace* dialog box, if we select *frame number* –D'4, corresponding line at D'1502 in source code will be highlighted, if we click next, source code line number D'1530 will be highlighted which corresponds to *frame number* –D'2.



6.5 Search Trace

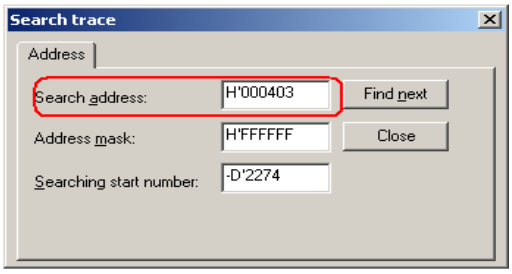
To find particular memory location in trace window, one can use *find* feature. Right click on Trace window, on a popup menu click on *Find. Search trace* dialog box will appear.



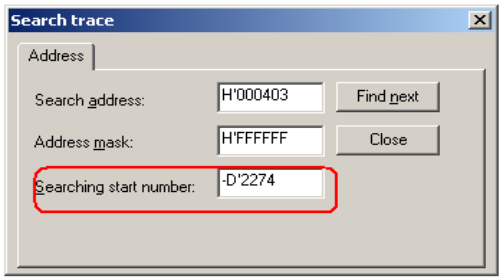
The line having information about searched memory location is highlighted with black background.

```
-01812 :  write     55AA at 00253A                    *                  1
-01811 :  write     55AA at 00253C                    *                  1
-01810 :  write     55AA at 00253E                    *                  1
-01809 :  read        B0 at 000403                    *                  9
-01808 :    F800A2  BBC     0403:6,STARTUP\no_PLL_yet*                   1
-01807 :  read        B0 at 000403                    *                  9
-01806 :    F800A2  BBC     0403:6,STARTUP\no_PLL_yet*                   1
-01805 :  read        B0 at 000403                    *                 10
-01804 :    F800A2  BBC     0403:6,STARTUP\no_PLL_yet*                   1
-01803 :  read        B0 at 000403                    *                  9
-01802 :    F800A2  BBC     0403:6,STARTUP\no_PLL_yet*                   1
-01801 :  read        B0 at 000403                    *                 10
-01800 :    F800A2  BBC     0403:6,STARTUP\no_PLL_yet*                   1
-01799 :  read        B0 at 000403                    *                  9
-01798 :    F800A2  BBC     0403:6,STARTUP\no_PLL_yet*                   1
-01797 :  read        B0 at 000403                    *                 10
```

The required memory location is entered in *Search address,*



the frame number from where to start the trace search is entered in Searching start number

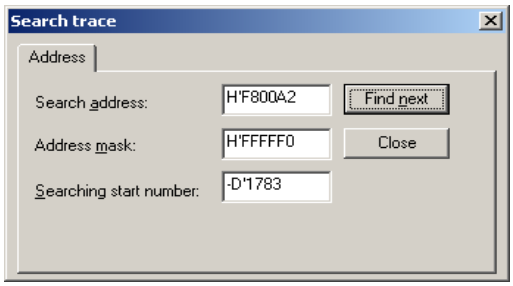


When *Address mask* is selected, for each bit set in *Address mask* field, that bit from *Search address* field is compared for the exact match with the corresponding bit of the memory location read by the debugger from trace data buffer. I.e. The line having the frame number for which following condition is met, will be highlighted in Trace window

(Memory location read by debugger) & (Address Mask) =

(Search address) & (Address Mask)

For the trace search, configured as shown in the figure below,

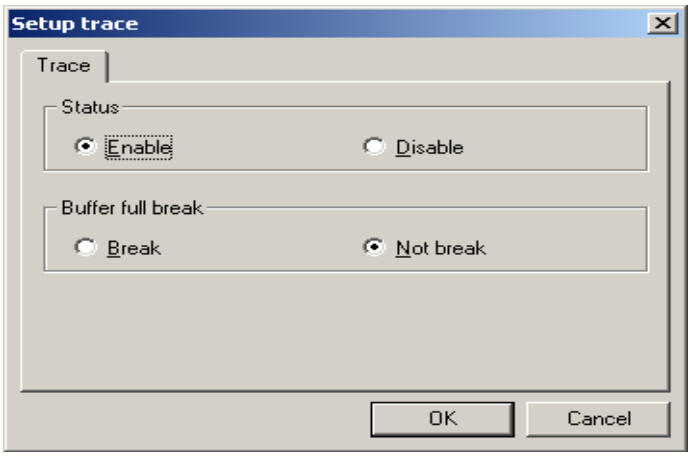


when every time *Find next* button is clicked, trace frame, containing memory address in the range of H'F800A0 to H'F800AF is highlighted. When search reaches the first frame number (-D'1) it rollbacks and search continues from last frame (-D'65535).

6.6 Trace Setup

Trace functionality can be configured by using *Setup Trace* dialog box.

Right click on *trace* window, on pop up menu, click on *Setup...*

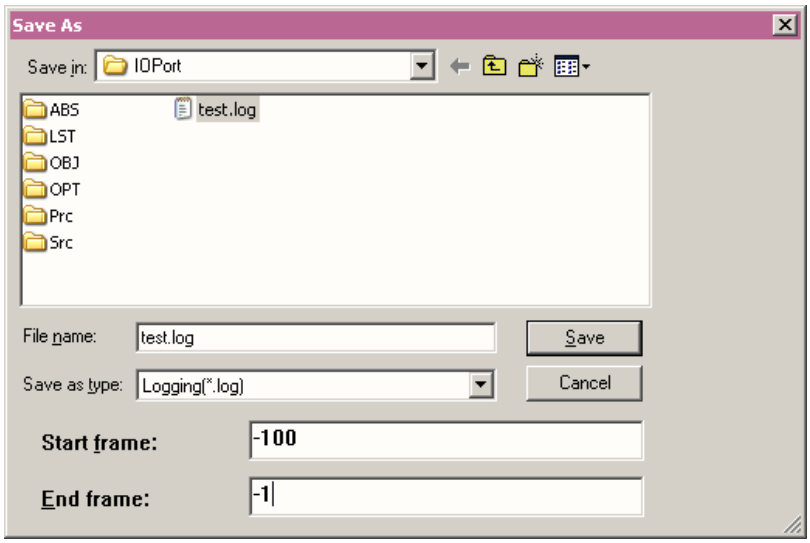


Trace data buffering can be enabled or disabled by selecting *Enable* or *Disable* radio button, under *Status* option.

Trace data is buffered in the trace buffer. The trace buffer becomes full some time during debugging because its size is finite. When the trace buffer becomes full, the program being executed can be stopped. Trace buffer full break is set by selecting *Break* radio button under *Buffer full break* option.

6.7 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 … .*

7 Memory Map

How to use Memory Map Function

A unit to allocate memory is allocated is called an area. There are three different area types as follows:

- Internal ROM Area

The area where the emulator internal memory is substituted for internal ROM is called the internal ROM area, and this memory is called the internal ROM memory.

- Undefined Area

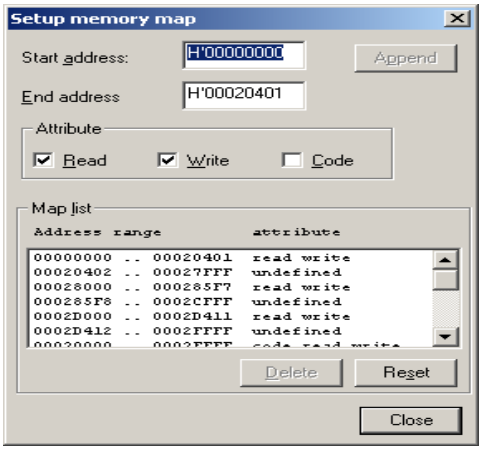
A memory area that does not belong to any of the areas described above is part of the user memory area. This area is specifically called the undefined area.

Access attributes can be set for each area; for example, CODE, READ, etc., can be set for ROM area, and READ, WRITE, etc. can be set for RAM area. If the MCU attempts access in violation of these attributes, the MCU operation is suspended and an error is displayed (guarded access break).

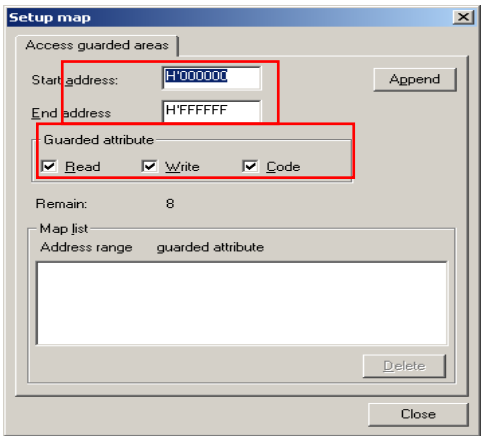
"Memory Map..." sets the debugger's memory map. When set, particular memory area can be guarded for read, writes and/or code access.

7.1 Memory Map Setup

Memory map setup dialog box can be opened by clicking on *Setup* and then *Memory Map…* on pull down menu



An address range, an access attribute for the range can be set from the memory map setup dialog box.

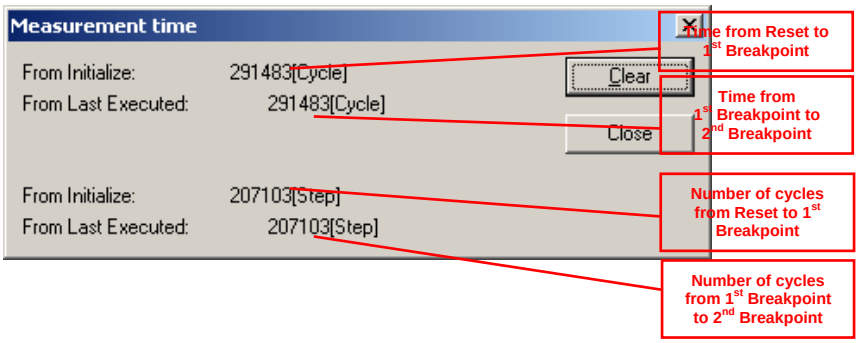


8 Time Measurement

How to use Time Measurement function

The Time Measurement is the simplest method of measurement. A 64-bit counter is used for this measurement resulting maximum cycle count of 18446744073709551615.

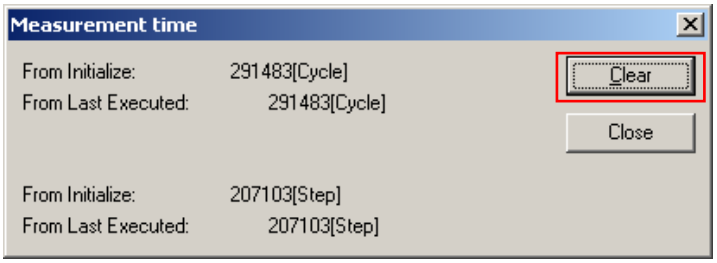
In this method, two breakpoints have to be set. After the program has run from breakpoint 1 to 2, the Time Measurement will show the difference between the breakpoints.



Now, take a look at example as shown below, in a *main()* function, two breakpoints are set. In *a wait()* function *PORT00* is set to high when entered in function and then set to low when left the function.

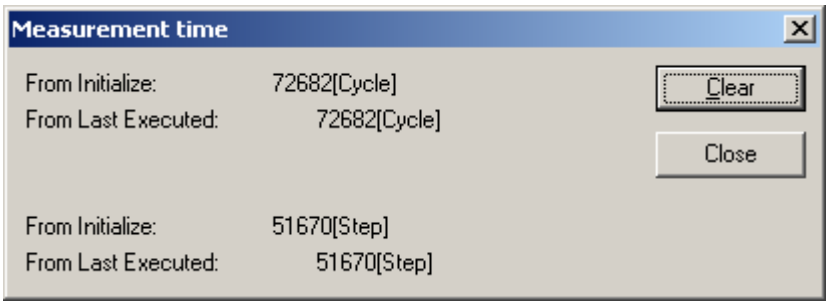
```
23:  void main(void)
24:  {
25:
26:          DDR00 = 0xFF;
27:          __asm(" NOP");
28:          wait (5000);
29:          __asm(" NOP");
30:
31:  }
32:
33:
34:
35:  void wait (int cnt)
36:  {
37:          int i    ;
38:          PDR00 = 0xFF;
39:          for (i = 0; i< cnt ; i++);
40:          PDR00 = 0x00;
41:
42:
43:  }
```

On Debug pull down menu click on *Time Measurement…*, Click on Clear if there are any time entries not equal to zero, Click on Close then.



Run the program. The MCU will stop at the beginning of the main function (1st breakpoint). This first stop initializes the time measurement counter. Next, run the program till the second breakpoint.

Now on Debug pull down menu click on *Time Measurement…*, one can find the following information

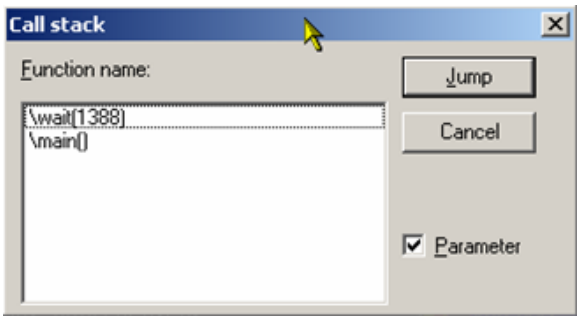


9 Call Stack

How to use call stack feature

Usually, a program is a set of several subroutines. For this reason, as debugging advances, function calls of several stages occur. For example, one routine calls another and the called routine further calls another.

Call stack window can be opened by clicking on *Call…* under *Debug* menu



The call stack retains the relationship between function calls. Clicking a function name from the function name list immediately displays information for the function in the Source Window.

The function written in the lowermost line of the function name list is the main function. This main function calls the function above it. The called function further calls a function above it. In this way, the function written in the uppermost line is the function in which the current PC exists.

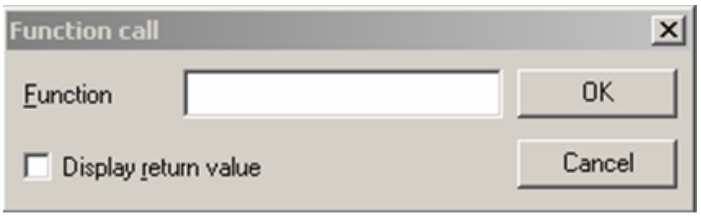
When return is executed, functions are deleted in turn from the function name list, starting from the uppermost line.

When a check mark is set to Parameter, an argument value is displayed after each function name, as shown in above figure. When no check mark is set, only parentheses "( )" are displayed after each function name.

10 Function Call

How to use Function Call feature

The specified function can be started during debugging without reference to the flow of the program. This function is known as function call. This feature can be used when execution is stopped. After MCU reset one should at least execute one instruction before using this feature. The *Function call* dialog box is opened by right clicking on *Call…* on Debug pull down menu



When the function call dialog box opens, specify the function you want to call with a correct argument. If a breakpoint is set in the called function, the program stops at this breakpoint. When processing of the called function is terminated and control is returned, the function call result dialog box opens, if Display return value check box is checked. The PC then returns to the value before the function was called.

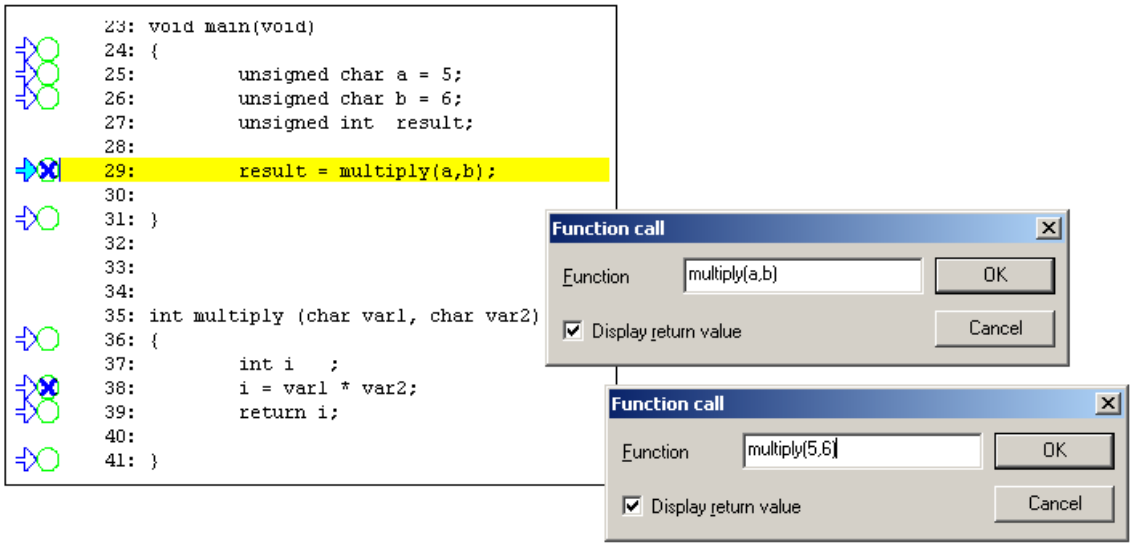
To understand the function, let us consider following example code

```
23:  void main(void)
24:  {
25:            unsigned char a = 5;
26:            unsigned char b = 6;
27:            unsigned int  result;
28:
29:            result = multiply(a,b);
30:
31:  }
32:
33:
34:
35:  int multiply (char var1, char var2)
36:  {
37:            int i    ;
38:            i = var1 * var2;
39:            return i;
40:
41:  }
```

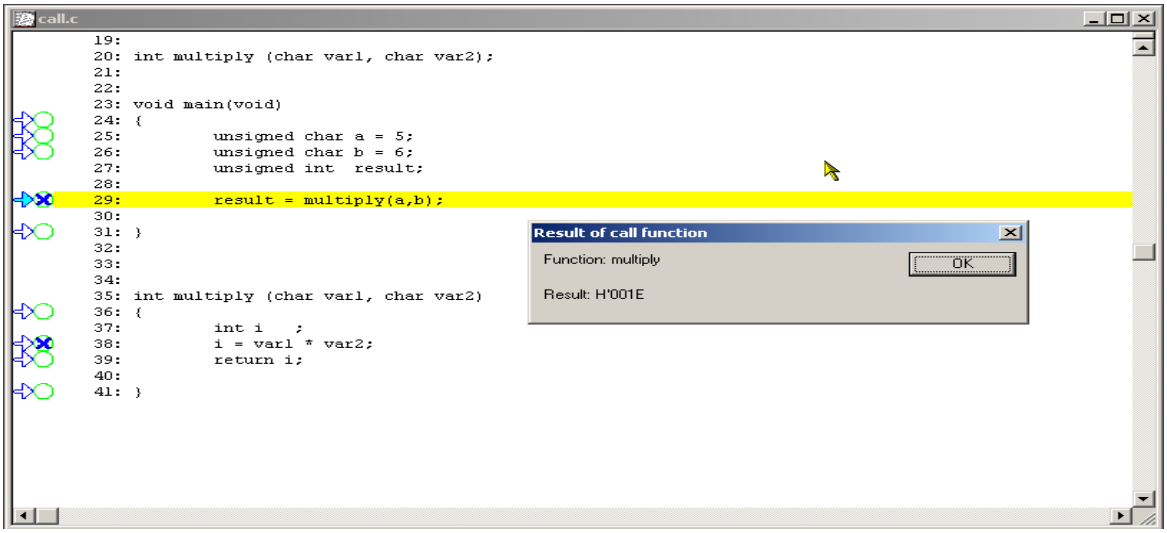
Let us set break point at line no. 29 and line no. 38. Start executing program, at first it halts at first break point. Now open *Function Call* dialog box (Debug -> Call…)

When the function definition is, *int multiply (char var1, char var2),* specify the function call as follow

- `multiply (5, 6);` Where a constant value is directly specified
- `multiply (a, b);` Where variable ‘a’ & ‘b’ is directly specified



Since break point is set in the called function, the program stops at this breakpoint. When run again at the end of processing of called function, control is returned, the *Result of call function* dialog box as shown below opens. The PC then returns to the value before the function was called.



11 Vector

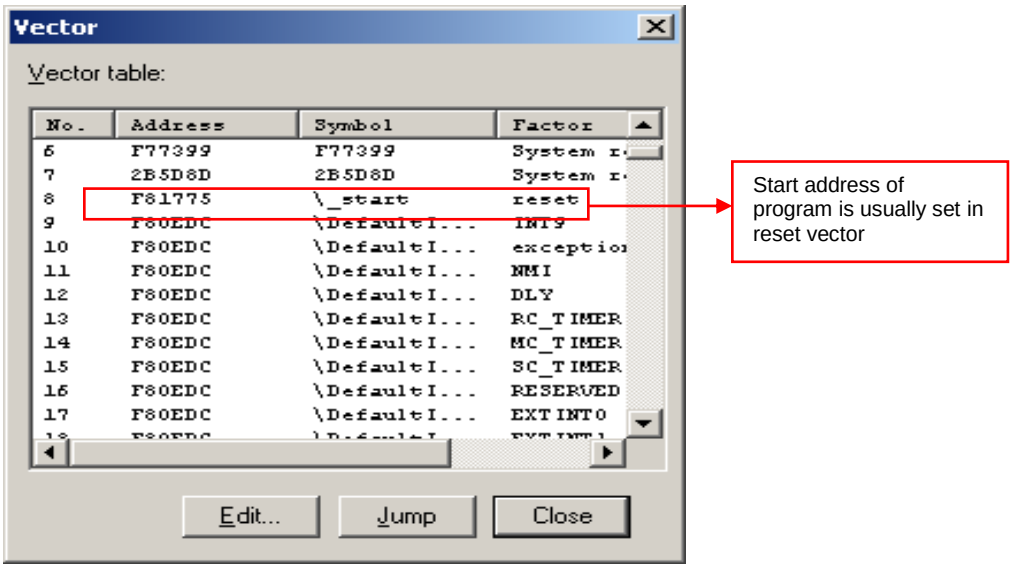
How to use Vector Feature

An interrupt vector is the memory address of an interrupt handler, or an index into an array called an interrupt vector table or dispatch table. Interrupt vector tables contain the memory addresses of interrupt handlers. When an interrupt is generated, the processor saves its execution state via a context switch, and begins execution of the interrupt handler at the interrupt vector.

11.1 Display and setting vectors

11.1.1 Display

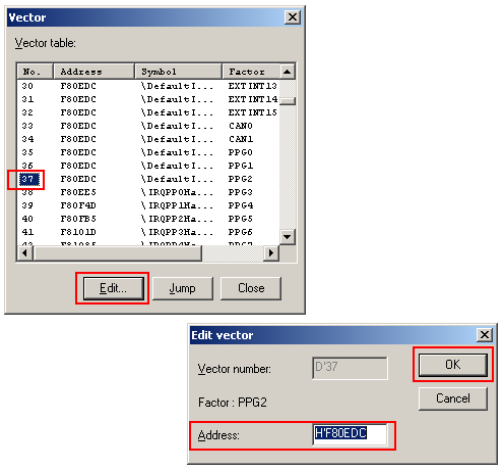
To display Interrupt vector table, click on *Vector…* at *Debug* pull down menu



11.1.2 Setting an address

Change the address set in a vector in the following procedure:

1. Select a vector table number and then click the Edit button. The vector edit dialog box shown below opens.

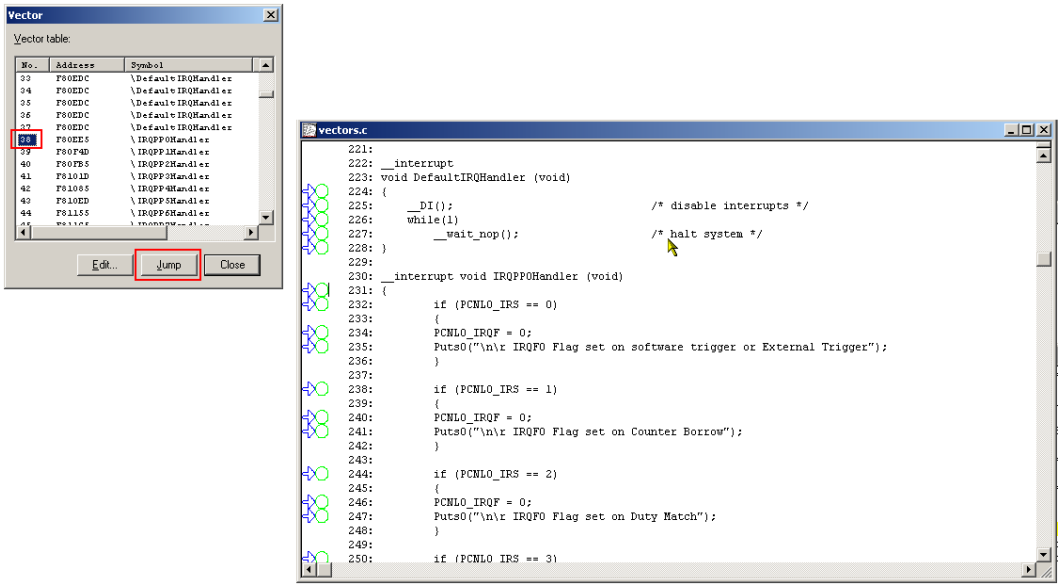


2. Set an address and then click the [OK] button

11.2 Jump

Display the source of the program stored at the address set in the vector table in the following procedure:

1. Select a vector number.
2. Click the [Jump] button.



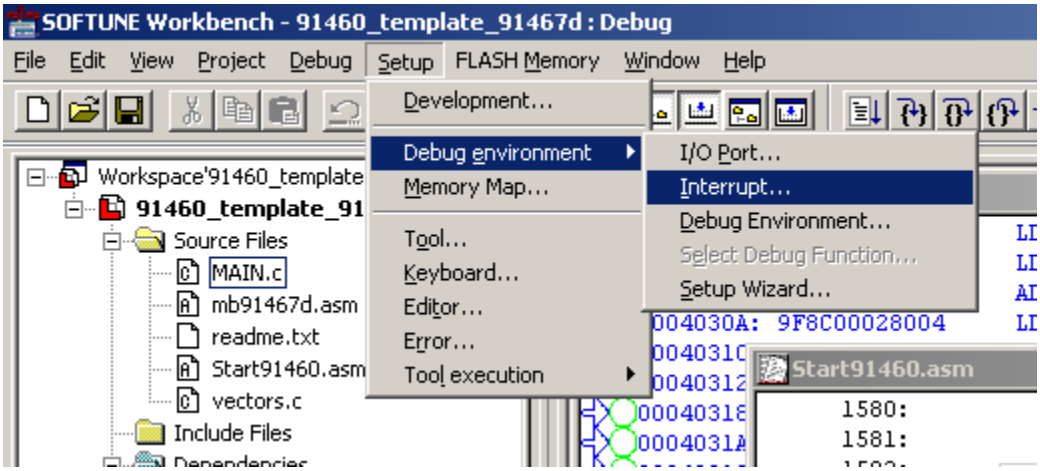
If the starting address of the program set in the vector table is incorrect, the source cannot be displayed (disassemble display).

The jump function merely displays the jump destination program; it does not update the program counter to move control to the address set in the vector table.

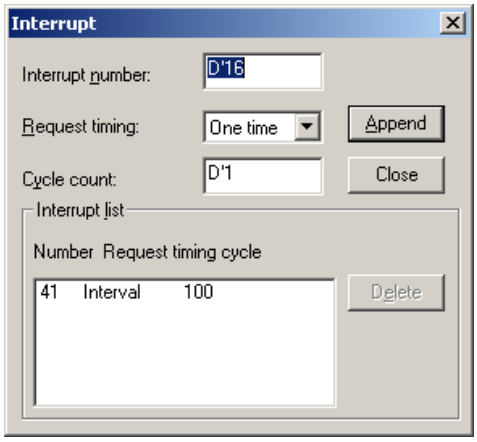
12 Interrupt Simulation

How to Simulate Interrupt

This simulates the MCU operation for an interrupt request. It is independent of corresponding peripherals interrupt enable flag and interrupt flag but depends of CCR register's Interrupt enable bit



Click on *Setup,* on a pull down menu click *Debug Environment.* On a side menu click on *Interrupt,* following dialog box will appear

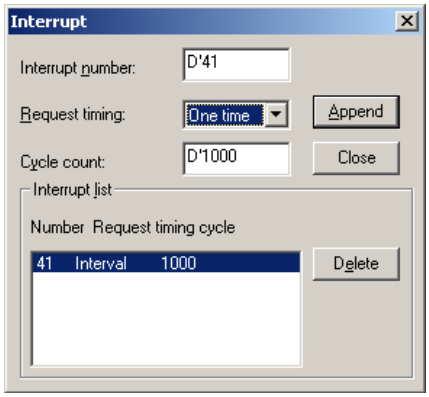


Interrupt can be selected for *one time* or *Interval* (recurring).

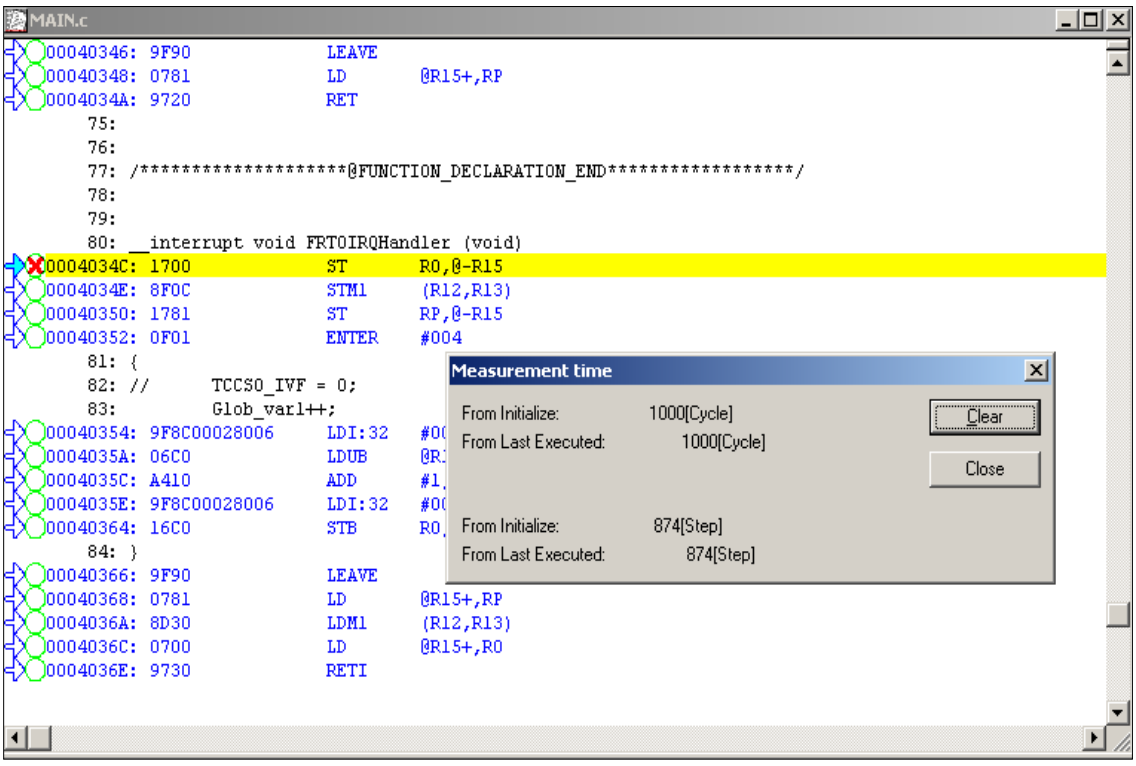
When selected *one time* and instructions are executed as many cycles as the specified *cycle count* while executing a program (executing execution commands), an interrupt is generated and the interrupt generating condition is reset

When selected *Interval* and every time instructions are executed as many cycles as the specified *cycle count* while executing a program (executing execution commands), an interrupt is generated.

Now for example if we want to simulate interrupt condition for Free running time 0 than we need to set *Interrupt number* as D'41.



Now, if we want to interrupt to occur only once we have to select *one time* under *Request timing* option. With the *cycle count* set to D'1000, FRT 0 interrupt will be generated when D'1000 cycles of instructions are executed while executing a program.

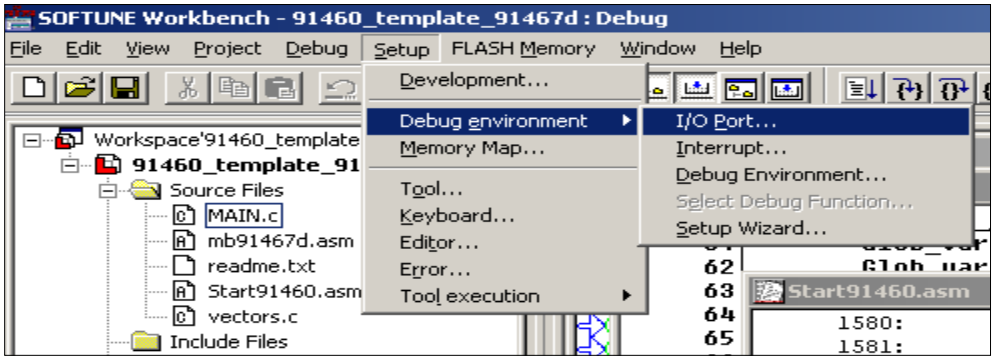


13 IO port Simulation

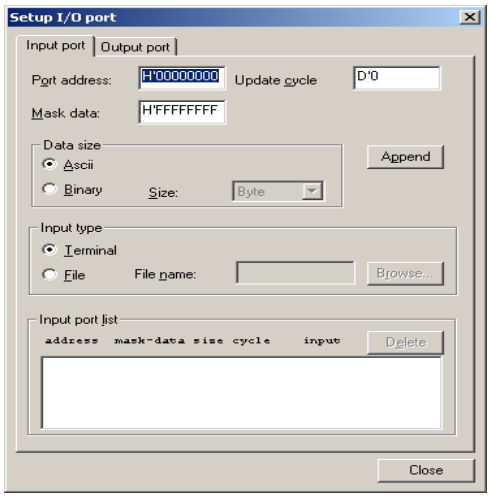
How to simulate IO Port

There are two types of simulations in I/O port simulation:

- Input port simulation
- Output port simulation

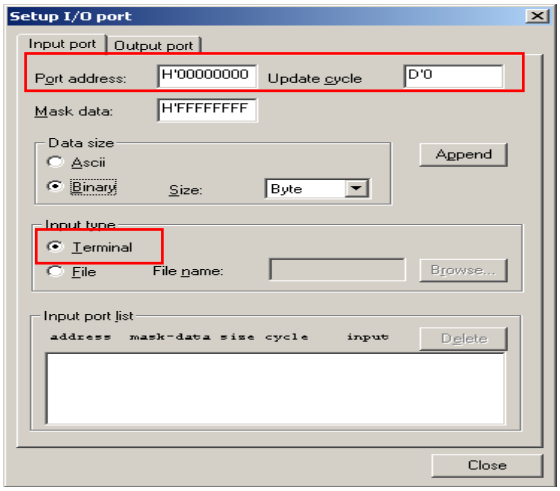


Click on *Setup,* on a pull down menu click *Debug Environment.* On a side menu click on *IO Port,* following dialog box will appear. Select *Input port* tab.

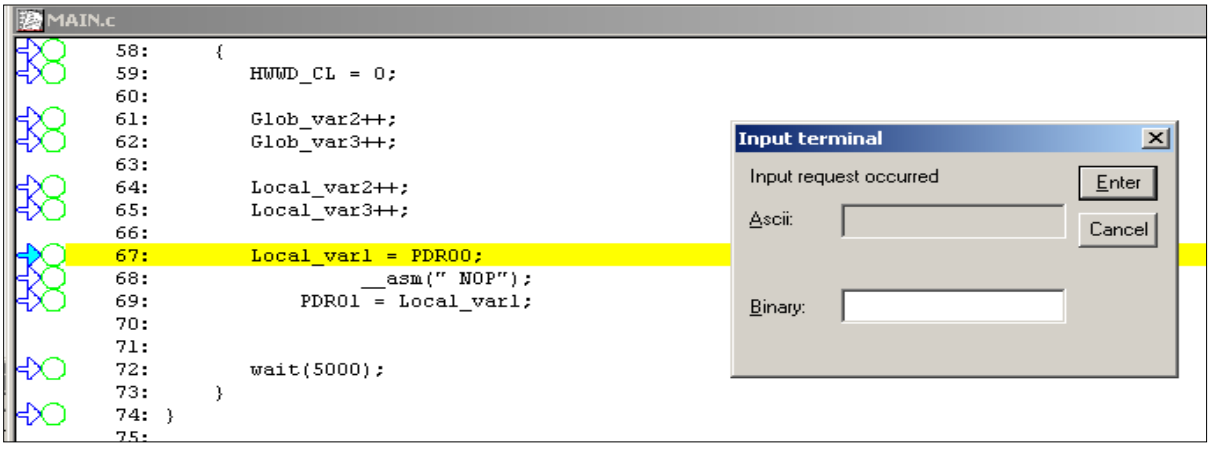


Input port simulation has two types, first, whenever a program reads the specified port, data is input from the pre-defined data input source.

For example, if we set, input port 0 for simulation

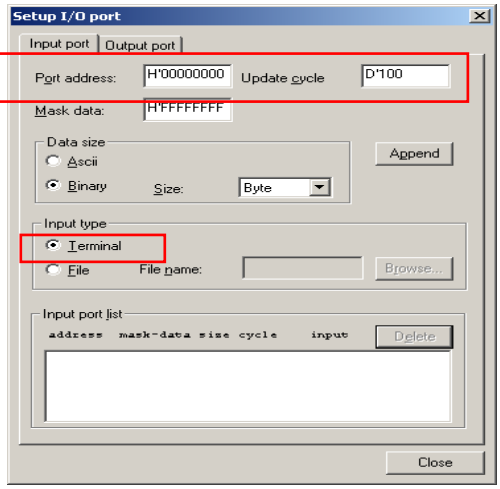


And whenever in software, data is read from the port 0, Input terminal window will popup



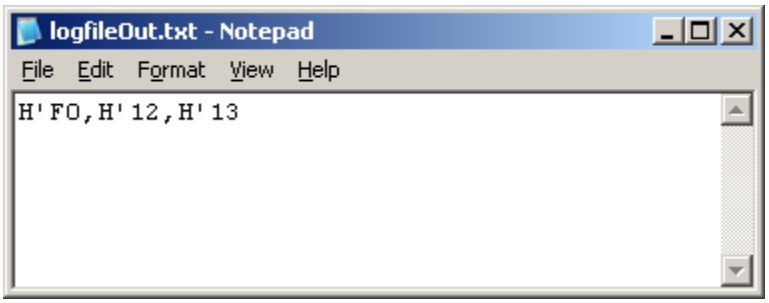
Depending on *Data size* selected in *Setup IO port* dialog box, either *Binary* or *ASCII,* input field will be enabled in *Input terminal* window

The second type of IO port simulation is that whenever the instruction execution cycle count exceeds the *Update cycle* count, *terminal* window will pop up and data is input to the port.



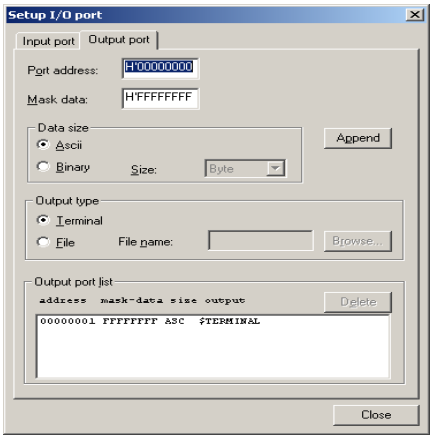
Up to 4096 port addresses can be specified for the input port. The data input source can be a file or a terminal. After reading the last data from the file, the data is read again from the beginning of the file.

A text file created by an ordinary text editor, or a binary file containing direct code can be used as the data input file. When using a text file, input the input data inside commas (,). When using a binary file, select the binary button in the input port dialog.

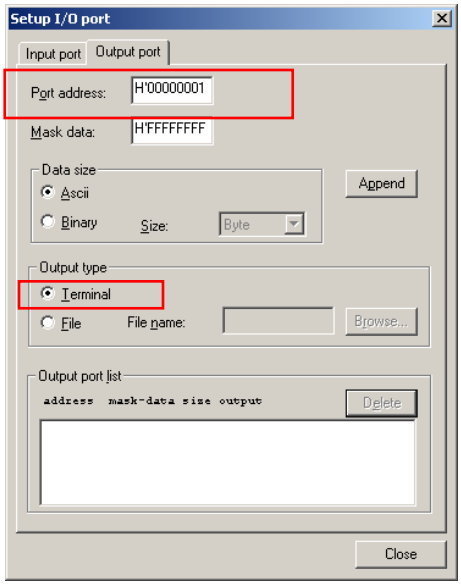


This method is not supported by high-speed simulator debugger.

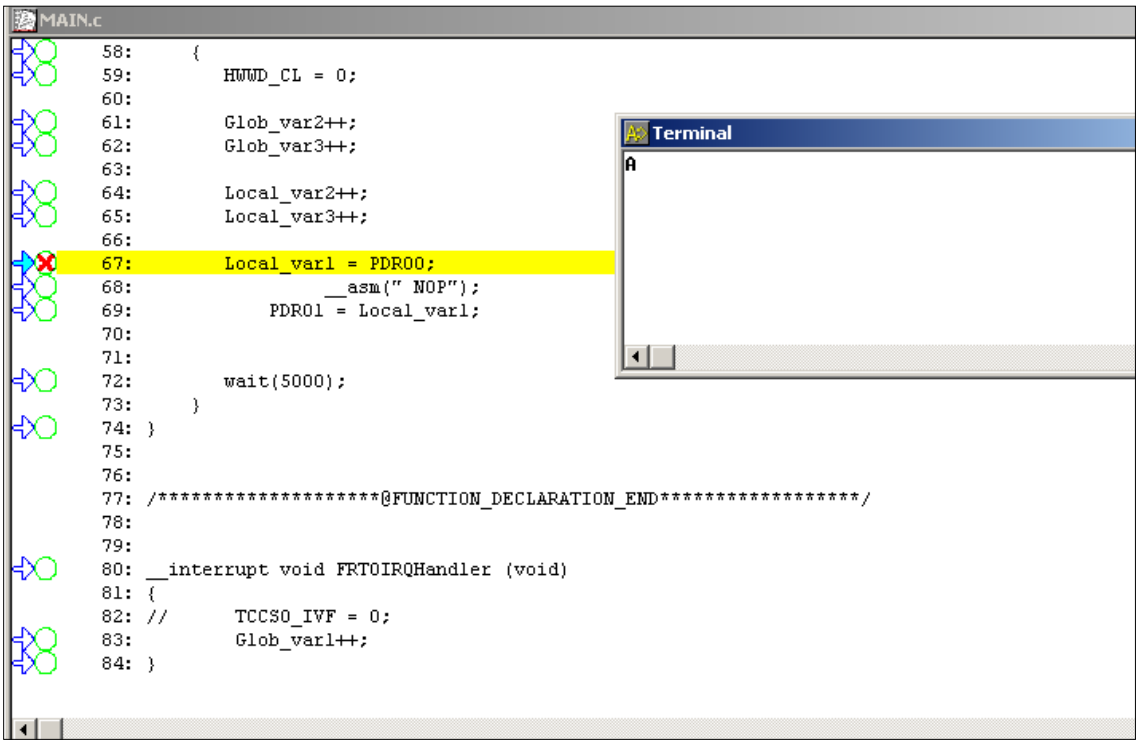
On a *Setup IO port* dialog box, select *Output port* tab



For example, if we set, input port 1 for simulation



And whenever in software, data is written to the port 1, *Input terminal* window will popup



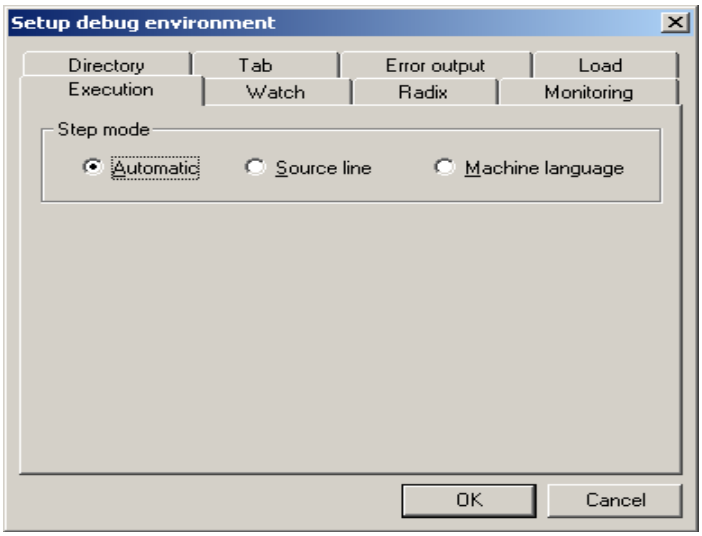
Up to 4096 port addresses can be set as output ports. Select either a file or terminal (Output Terminal window) as the data output destination.

A destination file must be either a text file that can be referred to by regular editors, or a binary file. To output a binary file, select the Binary radio button in the Output Port dialog.

Furthermore, the setting of memory map is necessary to set I/O port. When deleting memory map, I/O port is also deleted.

14 Debug environment setup procedure

14.1 Step Execution



Step Scale

- Automatic

Automatically sets the step unit according to the window display state. If it is a source code view, step will pass through each source line no. If it is a Mix mode code view, step will pass through disassemble source line no.

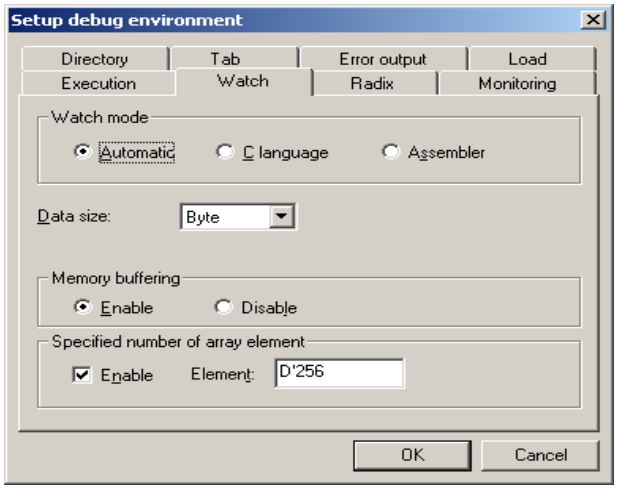
- Source Line

Executes the step in units of source lines.

- Instruct

Executes the step in units of machine languages.

14.2 Watch



Watch Mode

- Automatic

Sets the watch mode automatically according to the analysis result

- C Language

Sets the C language mode (interpretation as C language expressions).

- Assembler

Sets the assembler mode (interpretation as assembler expressions).

Data Size

Sets the display size in the assembler mode to either byte, word, long, single or double

Memory Buffering

- Enable

In case of variables as arrays or structures, memory of whole variables is read.

They are accessed by size of the top variable.

- Disable

In case of variables as arrays or structures, the memory of each variable is read.

Specified number of array element

- Enable

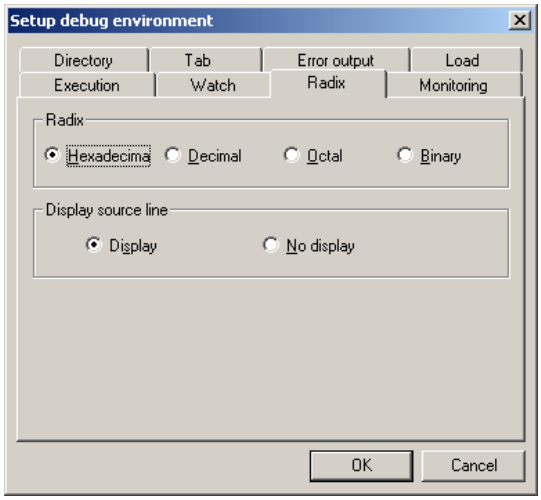
Debugger displays a warning dialogue in case of bigger array element than the number of array-element that you limited, when you register or expand an array with a watch variable.

- element

One can specify number (a default is D’256) of array element.

The default of this control is "Enable".

14.3 Radix



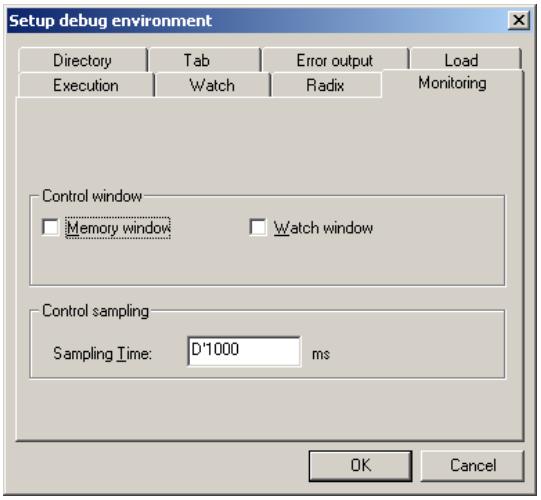
Radix

Sets the default base number for numerical value.

Display Source Line

Switches source line display to source line non display or vice versa for Trace- Instruction view.

14.4 Monitoring



Control Window

- Memory Window

Specifies whether to monitor the Memory Window.

- Watch Window

Specifies whether to monitor the Watch Window.

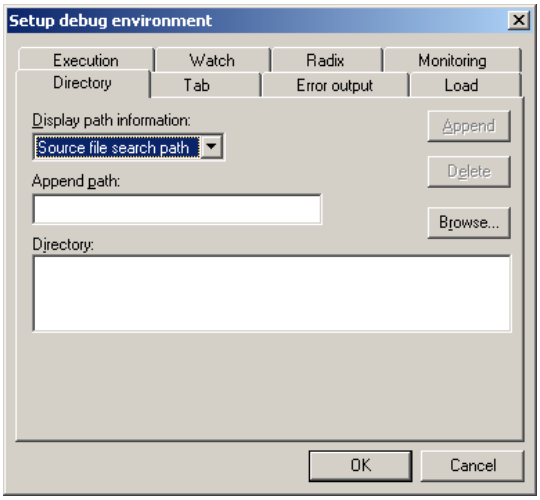
- Object Window

Specifies whether to monitor the Object Window.

Control Sampling

Specifies sampling time for Watch window, Memory window and Object window

14.5 Directory



Display path information

Specifies the path information to be displayed.

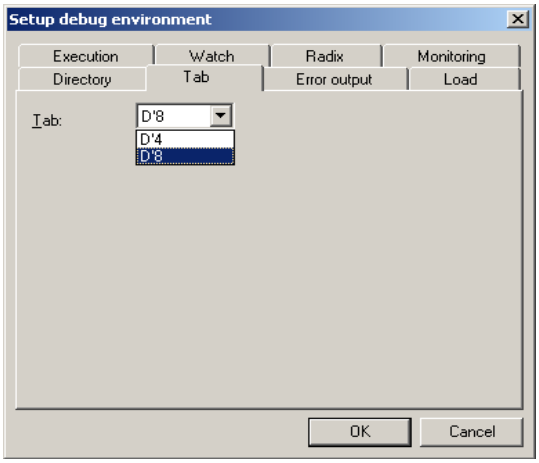
Append path

Sets the path to be added.

Directory

Displays the currently set directory.

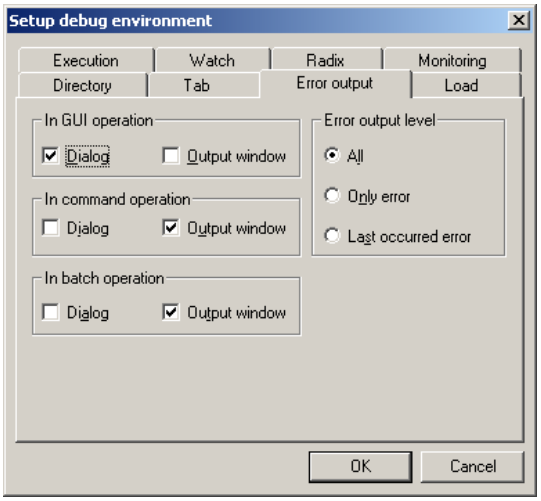
14.6 Tab



Tab

Specifies the Tab. (D’4/D’8)

14.7 Error output



In GUI Operation

Specifies where to output an error at GUI operation.

In Command Operation

Specifies where to output an error at command operation.

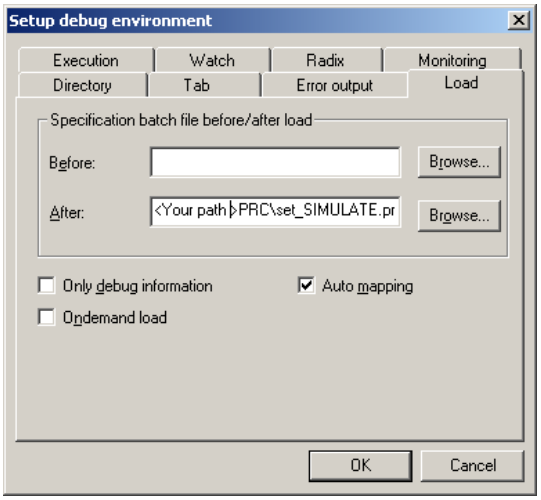
In Batch Operation

Specifies where to output an error at batch operation.

Error Output Level

Sets the output type when several errors occur.

14.8 Load



This sets the environment when loading a target file registered in the project.

Specification Batch File before/after load

- Before

This specifies the batch file to execute prior to the loading of the target file. This can also be changed using the Debugger's setup wizard.

- After

This specifies the batch file to execute after the loading of the target file. This can also be changed using the debugger's setup wizard.

- Debug Information Only

This specifies whether or not to load debug information. When checked, only the debug information is loaded.

Consider a system where MCU application program is stored in external flash and MCU is to be started in external vector fetch mode.

For the emulation of such a system one can connect the target with this check box marked. By doing this only debug information is loaded and with the help of Softune one can debug the application program. The restriction in this scenario is one can use only hardware breakpoint for debugging.

15 Additional Information

Information about Cypress Microcontrollers can be found on the following Internet page:

http://www.cypress.com/cypress-microcontrollers

Document History

Document Title: AN205430 - FR, All Series, Simulation with SOFTUNE Workbench V6

Document Number:002-05430

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	MKEA	07/25/2008	V1.0 HPi First draft
*A	5068203	MKEA	04/12/2016	Converted Spansion Application Note “MCU-AN-300103-E-V10” to Cypress format
*B	5862789	AESATP12	08/24/2017	Updated logo and copyright.
*C	6054547	NOFL	02/12/2018	Updated links Updated template

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Arm® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP | PSoC 6 MCU

Cypress Developer Community

Community | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/support

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709