ASHLING
EMBEDDED TECHNOLOGY AND INNOVATION

Ashling Product Brief APB200

Advanced Debugging using the Ashling MPC5500 tools

1. Contents

2. Introduction

This document provides some examples of advanced debugging features available when using the Ashling MPC5500 Tools. Examples are based on PathFinder for MPC5500 v1.1.1 using an Ashling Vitra for MPC5500 Emulator and MPC5566 based target board. It is assumed that you have installed/configured PathFinder appropriately for use with the target board. Advanced debugging features described include:

- How to break (halt) when a particular variable in your program is accessed
- How to trace (capture) all write accesses to a particular variable
- How to trace code execution between two events, after an event, up to an event or a program halt

3. Break on Data Access

This example shows how to break (halt execution) when a particular variable in your program is accessed. We will use the example program

`C:\PFMPC\Examples\Controlr\MPC5534\BIN\CONTROLR_RAM.CSO`

Load the program via PathFinder's **File|Load** menu

To halt when ever the variable `iLastRandValue` is accessed then setup an e200 Data Watchpoint at this variable as follows:

1. Open the **Breakpoint Configuration** dialog via the **Run** menu
2. Set the **Address** field of **e200 Data Watchpoints Watchpoint 0** to `iLastRandValue` (use the **Browse** button to symbolically pick `iLastRandValue`)

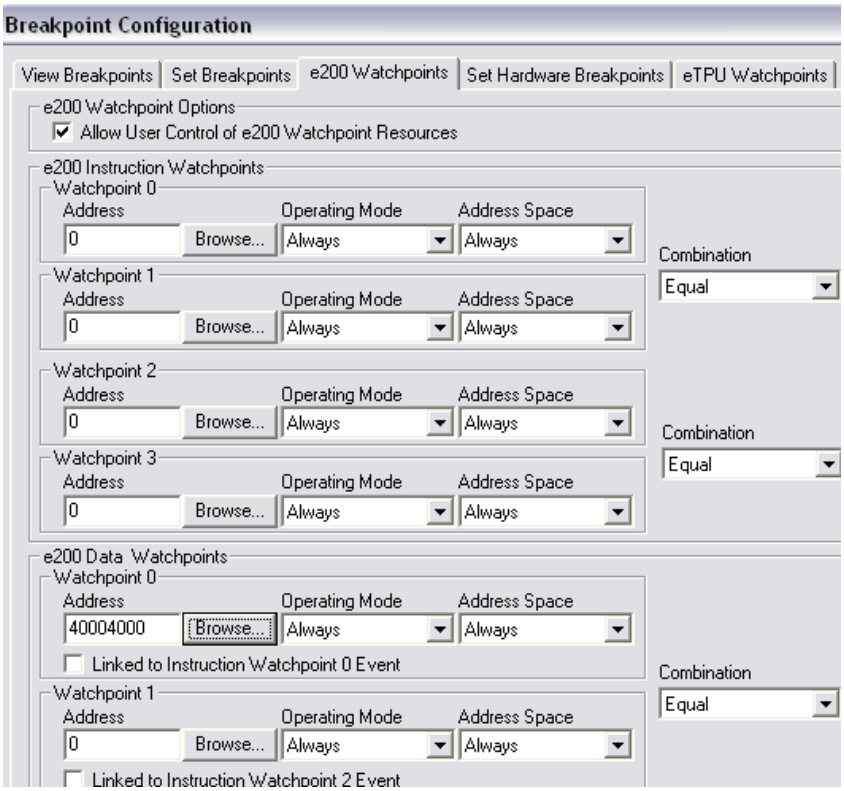


Figure 1. e200 Watchpoints Dialog

3. In the **Set Hardware Breakpoints** tab check **e200 Data Watchpoint 0 Load Debug Event Set** to halt when `iLastRandValue` is read and **e200 Data Watchpoint 0 Store Debug Event Set** when `iLastRandValue` is written to.

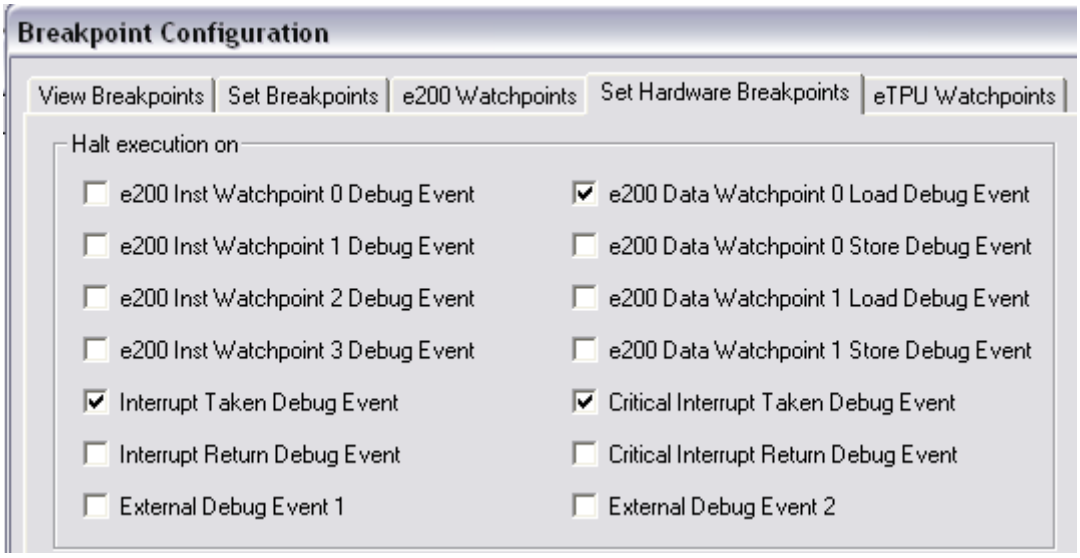


Figure 2. Set Hardware Breakpoints Dialog

4. Click OK and run the program from reset (**Run|Go from Reset**). The program will halt when `iLastRandValue` is either read or written to depending on your selection. Notice how the cause of break is shown in PathFinder's Status bar (we halted in the example below when `iLastRandValue` was read).

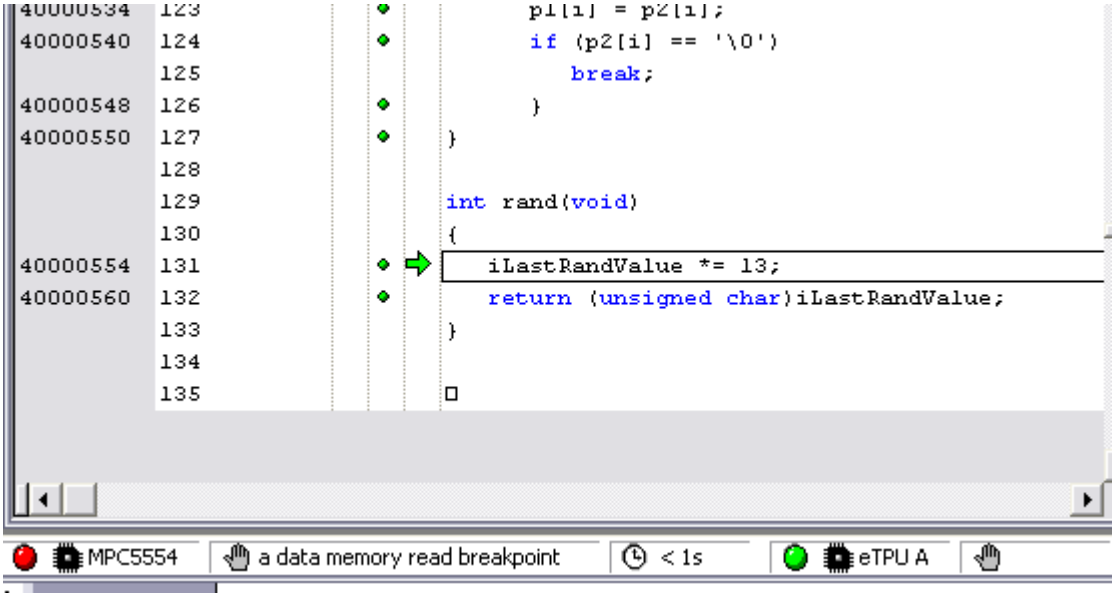


Figure 3. PathFinder Source Window

4. Tracing Accesses to a Variable

This example shows how to trace (capture) all write accesses to specified variable in your program. We will use the example program

`C:\PFMPC\Examples\Controlr\MPC5534\BIN\CONTROLR_RAM.CSO`

Load the program via PathFinder's **File|Load** menu

To trace write accesses to the variable `iLastRandValue` then setup a Trigger using **Trigger|Trigger Configuration** as follows:

1. The **Trace Options** tab can be left at default settings i.e.:

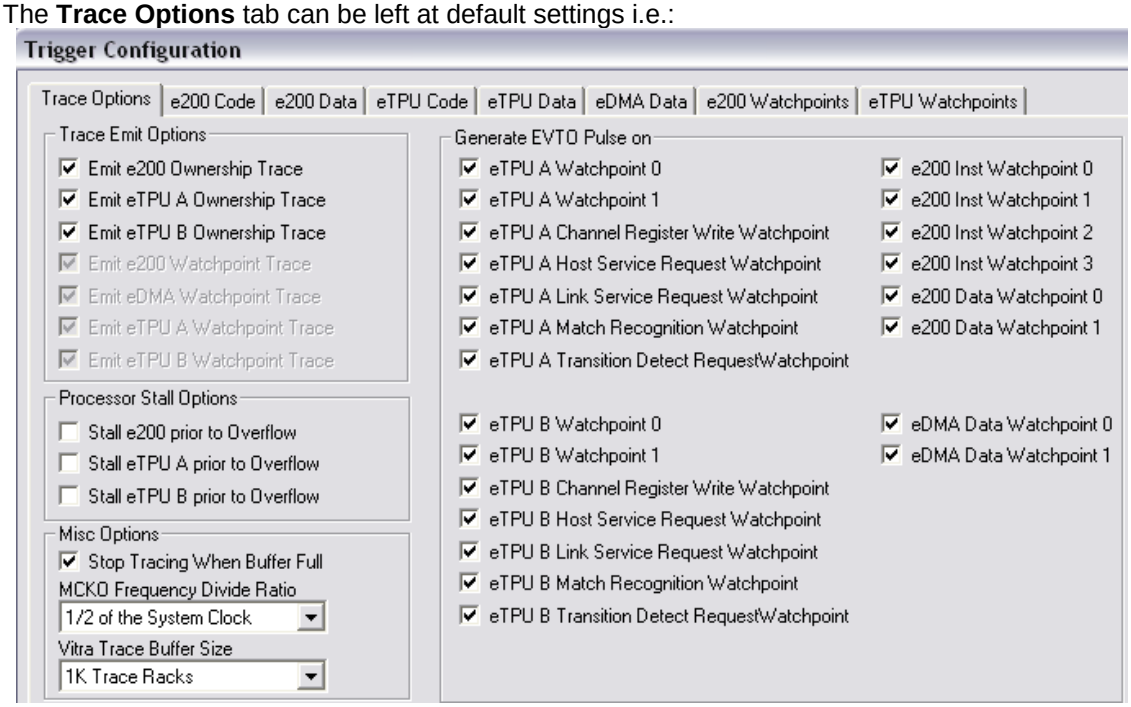


Figure 4. Trace Options Dialog

Depending on your Trace requirements you may want to adjust the **Vitra Trace Buffer Size**.

2. The **e200 Code** tab can be left at default settings i.e.:

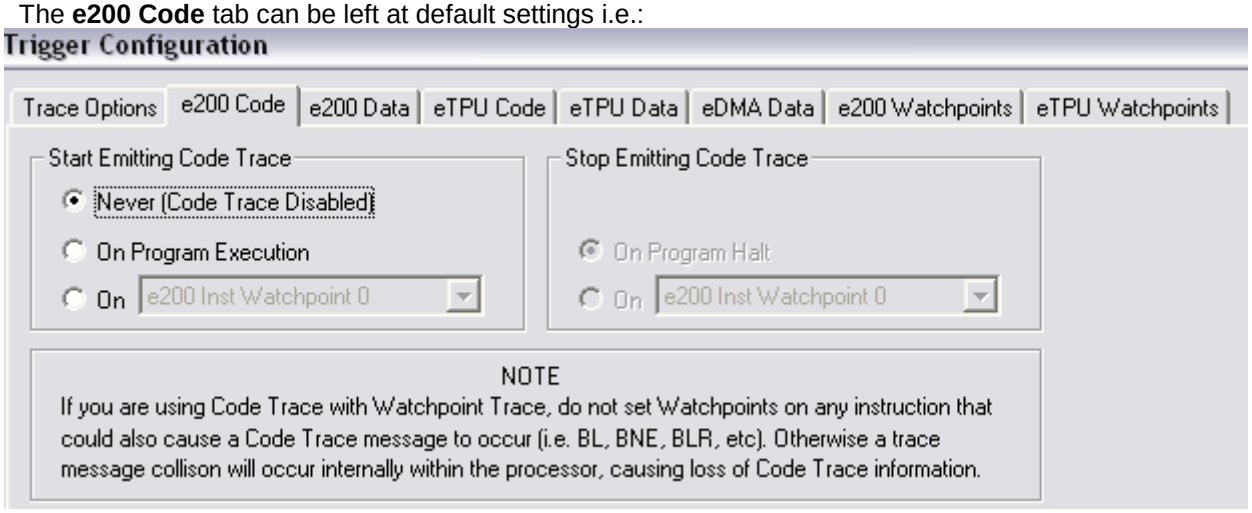


Figure 5. e200 Code Dialog

3. The **e200 Data** tab should be set as shown below.
   - Set **Start Emitting Data Trace** to **On Program Execution** and **Stop Emitting Data Trace** to **On Program Halt**.
   - Set **Data Trace Region 0|Trace Type** to **Write Trace** and the **Start Address** and **End Address** to `iLastRandValue` (use the **Browse** button to symbolically pick `iLastRandValue`)

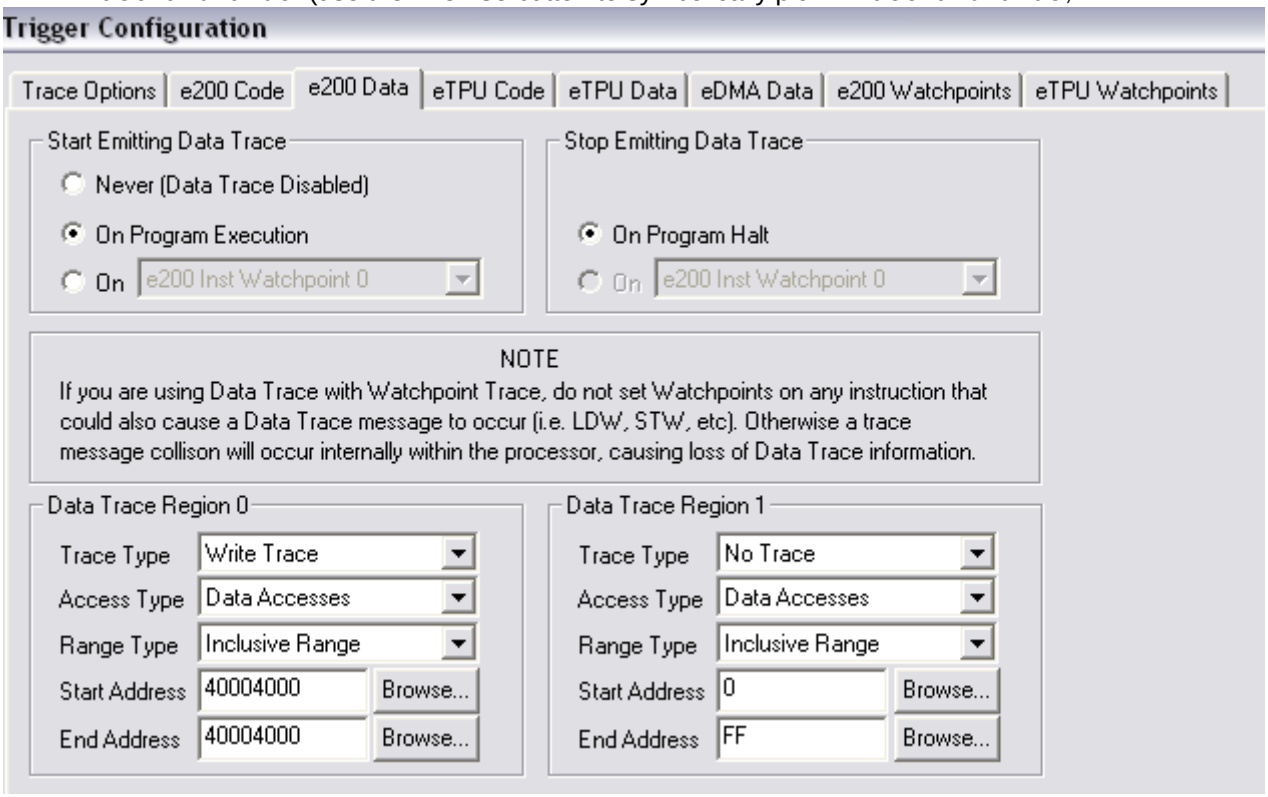
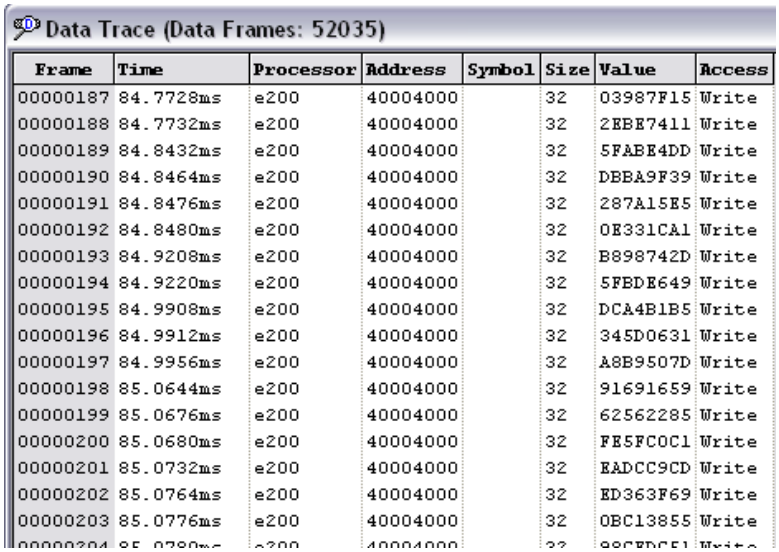


Figure 6. e200 Data Dialog

4. Click **Activate**, enable Trace (**Trace|Enable Trace**…) and run the program from reset (**Run|Go from Reset**). Halt the program after a few seconds and open the Data Trace window. All write accesses to `iLastRandValue` will be shown as below:



Frame	Time	Processor	Address	Symbol	Size	Value	Access
00000187	84.7728ms	e200	40004000		32	03987F15	Write
00000188	84.7732ms	e200	40004000		32	2EBE7411	Write
00000189	84.8432ms	e200	40004000		32	5FABE4DD	Write
00000190	84.8464ms	e200	40004000		32	DBBA9F39	Write
00000191	84.8476ms	e200	40004000		32	287A15E5	Write
00000192	84.8480ms	e200	40004000		32	0E331CA1	Write
00000193	84.9208ms	e200	40004000		32	B898742D	Write
00000194	84.9220ms	e200	40004000		32	5FBDE649	Write
00000195	84.9908ms	e200	40004000		32	DCA4B1B5	Write
00000196	84.9912ms	e200	40004000		32	345D0631	Write
00000197	84.9956ms	e200	40004000		32	A8B9507D	Write
00000198	85.0644ms	e200	40004000		32	91691659	Write
00000199	85.0676ms	e200	40004000		32	62562285	Write
00000200	85.0680ms	e200	40004000		32	FE5FC0C1	Write
00000201	85.0732ms	e200	40004000		32	EADCC9CD	Write
00000202	85.0764ms	e200	40004000		32	ED363F69	Write
00000203	85.0776ms	e200	40004000		32	0BC13855	Write

Figure 7. Data Trace Window

4.1 Cycle accurate mode Data Tracing

PathFinder v1.1.1 supports **Cycle accurate mode** data tracing. In this mode, every Nexus data-access trace packet emitted from the chip is given a unique time-stamp. When **Cycle accurate mode** is off, then multiple Nexus packets will be assigned the same time-stamp. **Cycle accurate mode** therefore gives you more accurate time-stamps at the expense of less trace capacity.

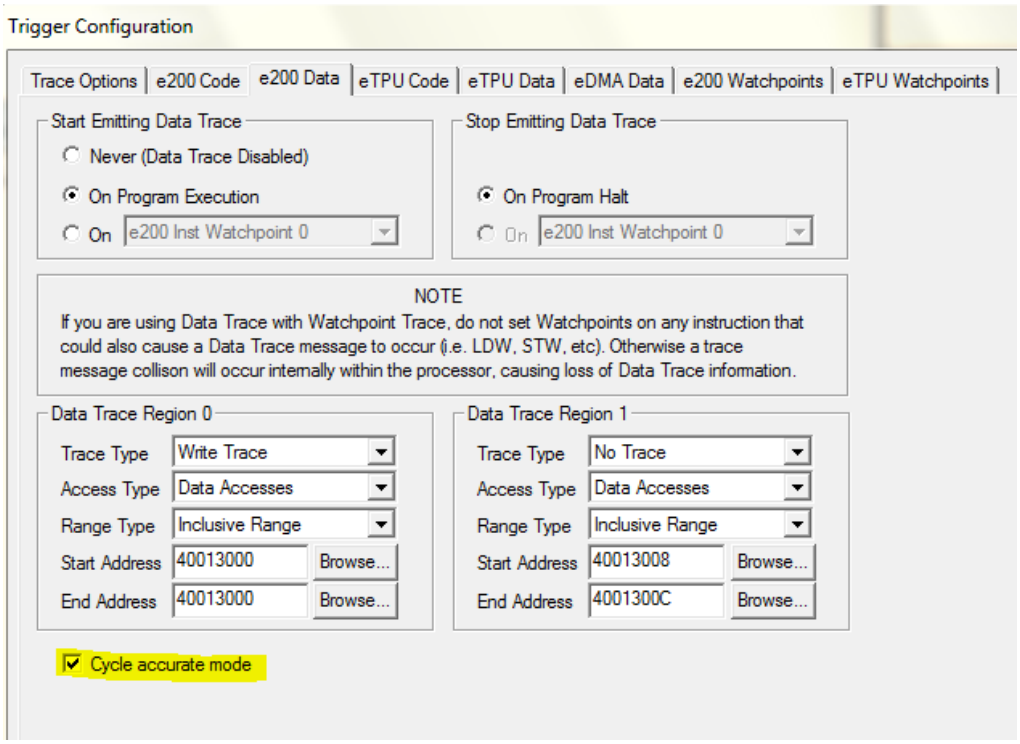


Figure 8. Enabling Cycle accurate mode

The follow screen-shots illustrate tracing with **Cycle accurate mode** on and off. These screen-shots use the example program `C:\PFMPC\Examples\1ms Timer\MPC5566\bin\RAM.CSO` which uses a 1mS interrupt handler to increments a variable. When **Cycle accurate mode** is on we can see the variable writes are accurately measured at 1mS intervals (+/-0.0005 mS)

Data Trace (Data Frames: 4865)

Frame	Time	Processor	Address	Symbol	Size	Value	Access
00000013	5.0484ms	e200	40013000	u32bit	32	00000005	Write
00000015	6.0488ms	e200	40013000	u32bit	32	00000006	Write
00000017	7.0484ms	e200	40013000	u32bit	32	00000007	Write
00000019	8.0488ms	e200	40013000	u32bit	32	00000008	Write
00000021	9.0484ms	e200	40013000	u32bit	32	00000009	Write
00000023	10.0488ms	e200	40013000	u32bit	32	0000000A	Write
00000025	11.0484ms	e200	40013000	u32bit	32	0000000B	Write
00000027	12.0488ms	e200	40013000	u32bit	32	0000000C	Write
00000029	13.0484ms	e200	40013000	u32bit	32	0000000D	Write
00000031	14.0488ms	e200	40013000	u32bit	32	0000000E	Write
00000033	15.0484ms	e200	40013000	u32bit	32	0000000F	Write
00000035	16.0488ms	e200	40013000	u32bit	32	00000010	Write
00000037	17.0484ms	e200	40013000	u32bit	32	00000011	Write
00000039	18.0488ms	e200	40013000	u32bit	32	00000012	Write
00000041	19.0484ms	e200	40013000	u32bit	32	00000013	Write
00000043	20.0488ms	e200	40013000	u32bit	32	00000014	Write
00000045	21.0484ms	e200	40013000	u32bit	32	00000015	Write
00000047	22.0488ms	e200	40013000	u32bit	32	00000016	Write
00000049	23.0484ms	e200	40013000	u32bit	32	00000017	Write
00000051	24.0488ms	e200	40013000	u32bit	32	00000018	Write
00000053	25.0484ms	e200	40013000	u32bit	32	00000019	Write
00000055	26.0488ms	e200	40013000	u32bit	32	0000001A	Write
00000057	27.0484ms	e200	40013000	u32bit	32	0000001B	Write
00000059	28.0488ms	e200	40013000	u32bit	32	0000001C	Write
00000061	29.0484ms	e200	40013000	u32bit	32	0000001D	Write
00000063	30.0488ms	e200	40013000	u32bit	32	0000001E	Write
00000065	31.0484ms	e200	40013000	u32bit	32	0000001F	Write
00000067	32.0488ms	e200	40013000	u32bit	32	00000020	Write
00000069	33.0484ms	e200	40013000	u32bit	32	00000021	Write
00000071	34.0488ms	e200	40013000	u32bit	32	00000022	Write
00000073	35.0484ms	e200	40013000	u32bit	32	00000023	Write
00000075	36.0488ms	e200	40013000	u32bit	32	00000024	Write

Figure 9. Data-tracing with Cycle accurate mode on

When running this program, ensure that Critical Interrupt Hardware Breakpoints are off as per the below screen-shot (this will prevent the programming halting at each interrupt).

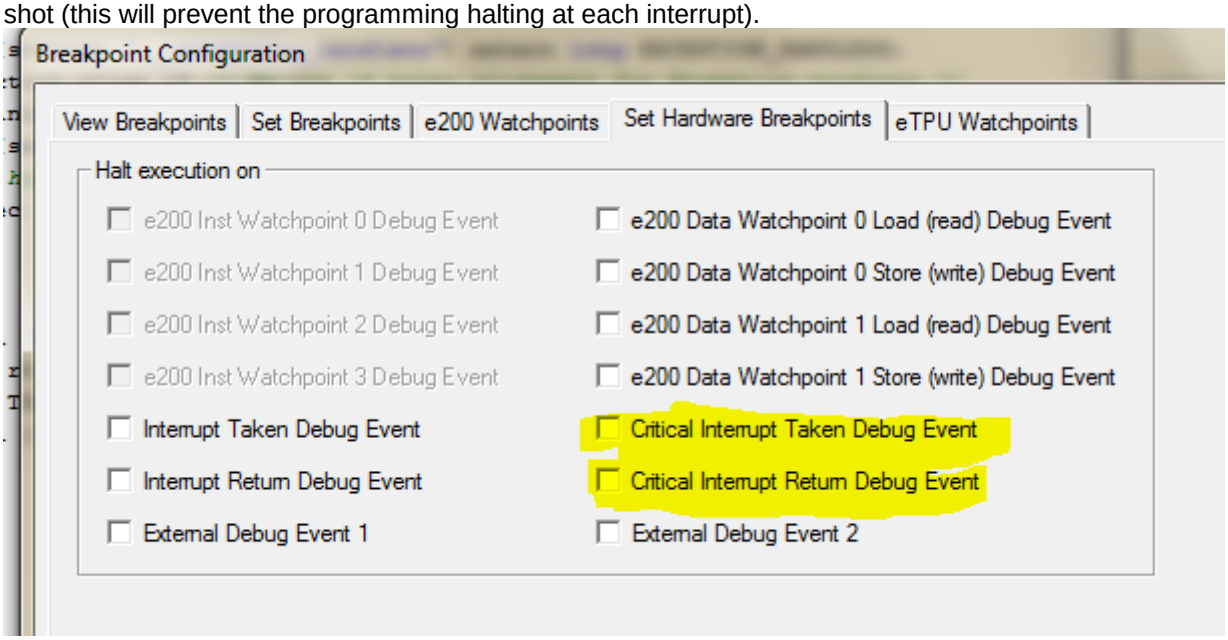


Figure 10. Setting Critical Interrupts off

5. Tracing Code Execution between two Events

In this example we demonstrate using Instruction Watchpoints to trace all e200 code execution between two specific events (the entry and exit of a particular function). We will use the example program:

```
C:\PFMPC\Examples\Controlr\MPC5534\BIN\CONTROLR_RAM.CSO
```

Load the program via PathFinder's **File|Load** menu

To capture all code execution of the function `WriteToDevice` then setup a Trigger using **Trigger|Trigger Configuration** as follows:

1. The **Trace Options** tab can be left at default settings i.e.:

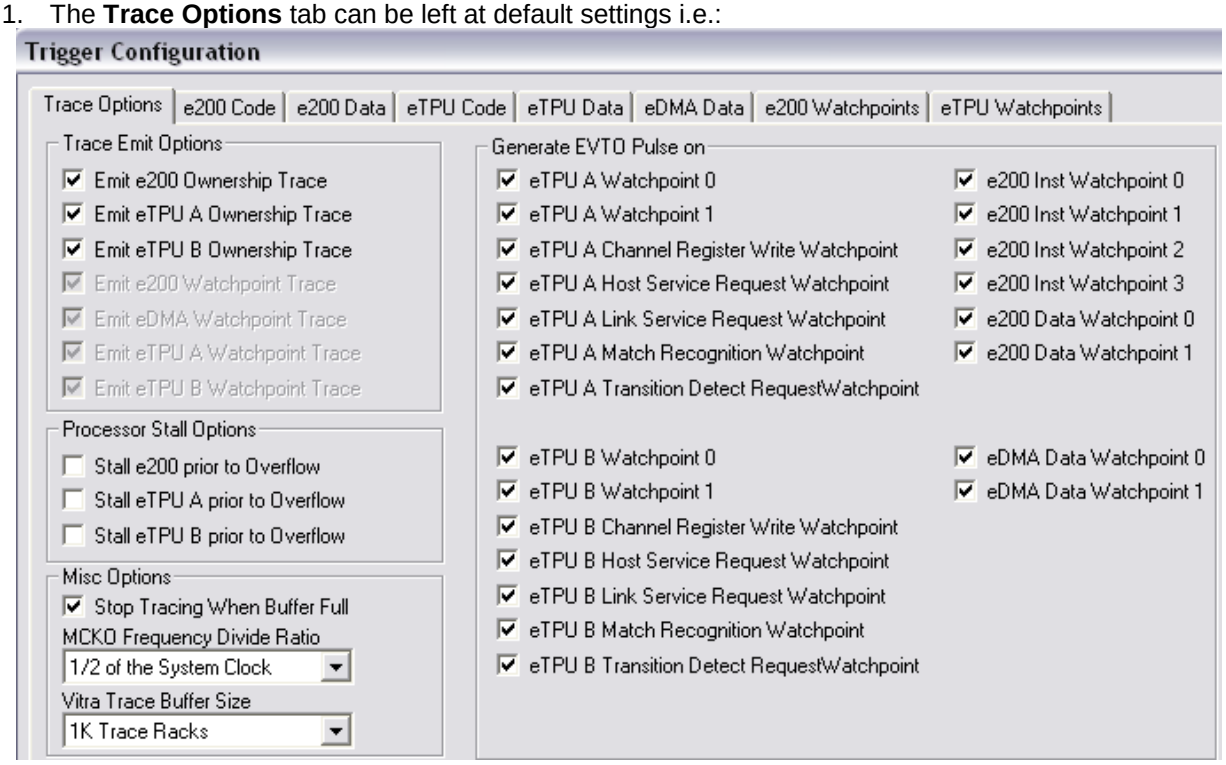


Figure 11. Trace Options Dialog

2. The **e200 Code** tab should be set to **Start Emitting Code Trace on e200 Inst Watchpoint 0** and **Stop Emitting Code Trace on e200 Inst Watchpoint 1** as shown below:

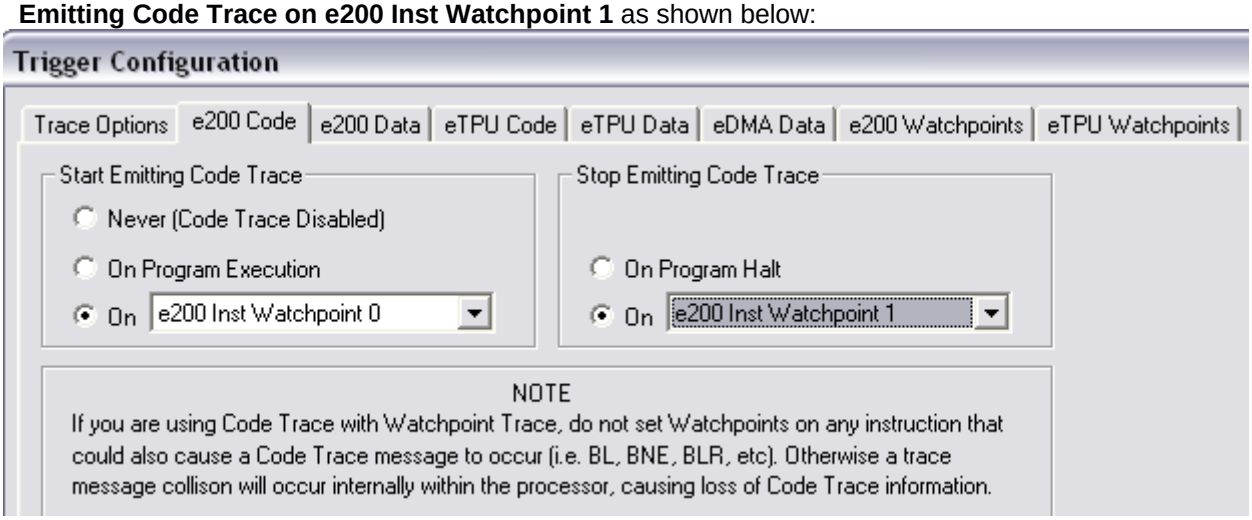


Figure 12. e200 Code Dialog

3. The **e200 Data**, **eTPU Code**, **eTPU Data**, **eDMA Data** and **eTPU Watchpoints** should be left at their default values and the **e200 Watchpoints** tab should be set as shown below.
    - Check **Allow User Control of e200 Watchpoint Resources**
    - Set **Watchpoint 0** to the entry of `WriteToDevice`
    - Set **Watchpoint 1** to the exit of `WriteToDevice`
    - This can be done symbolically using the **Browse…** dialog (invoke via the **Browse…** button and make sure that **Display Line Number Symbols** is checked)

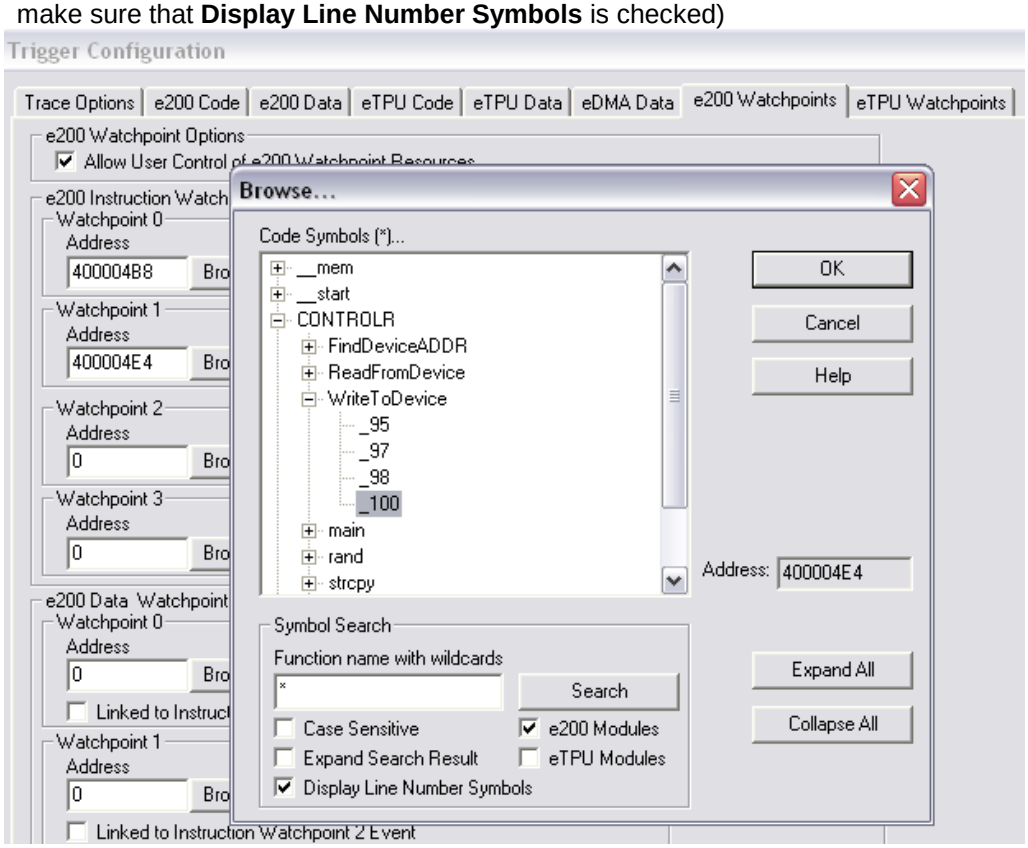
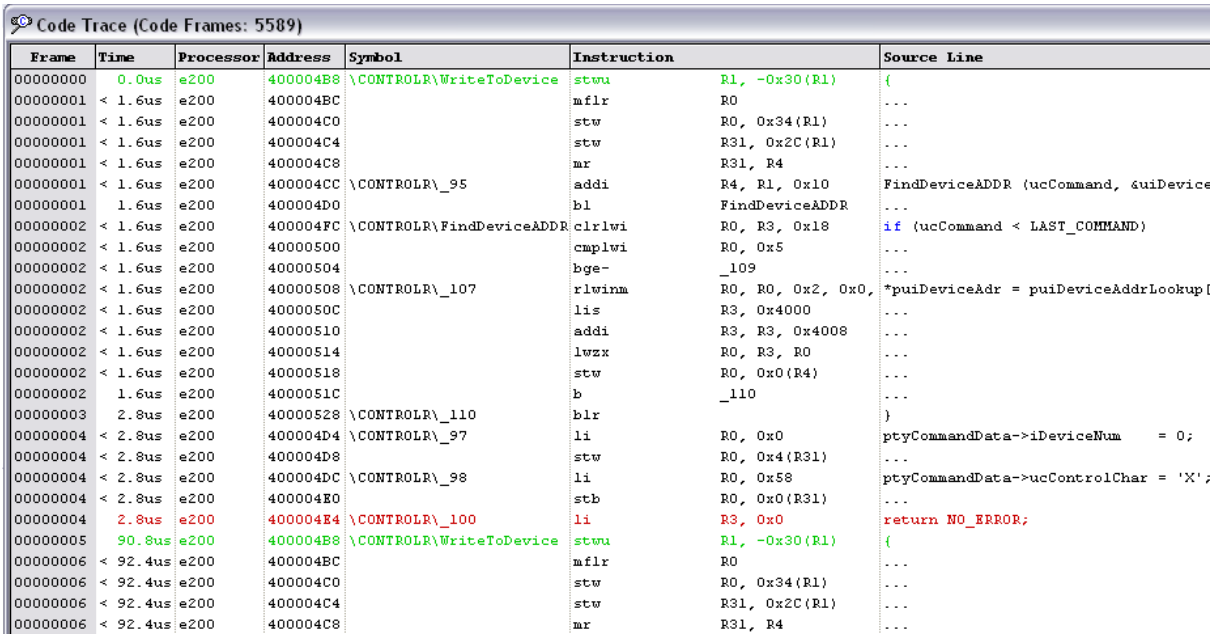


Figure 13. e200 Watchpoints Dialog (setting Watchpoint 1)

4. Click **Activate**, enable Trace (**Trace|Enable Trace**…) and run the program from reset (**Run|Go from Reset**). Halt the program after a few seconds and open the Code Trace window. All calls to `WriteToDevice` will be shown as below:

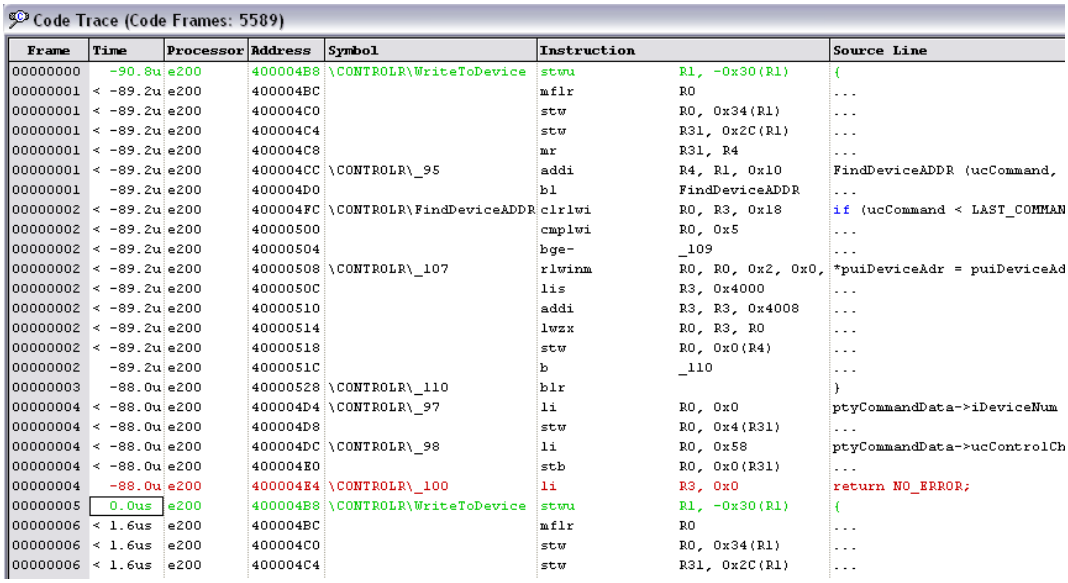


Code Trace (Code Frames: 5589)

Frame	Time	Processor	Address	Symbol	Instruction		Source Line
00000000	0.0us	e200	400004B8	\CONTROLR\WriteToDevice	stwu	R1, -0x30(R1)	{
00000001	< 1.6us	e200	400004BC		mflr	R0	...
00000001	< 1.6us	e200	400004C0		stw	R0, 0x34(R1)	...
00000001	< 1.6us	e200	400004C4		stw	R31, 0x2C(R1)	...
00000001	< 1.6us	e200	400004C8		mr	R31, R4	...
00000001	< 1.6us	e200	400004CC	\CONTROLR_95	addi	R4, R1, 0x10	FindDeviceADDR (ucCommand, &uiDevice
00000001	1.6us	e200	400004D0		bl	FindDeviceADDR	...
00000002	< 1.6us	e200	400004FC	\CONTROLR\FindDeviceADDR	clrlwi	R0, R3, 0x18	if (ucCommand < LAST_COMMAND)
00000002	< 1.6us	e200	40000500		cmplwi	R0, 0x5	...
00000002	< 1.6us	e200	40000504		bge-	_109	...
00000002	< 1.6us	e200	40000508	\CONTROLR_107	rlwinm	R0, R0, 0x2, 0x0,	*puiDeviceAdr = puiDeviceAddrLookup[
00000002	< 1.6us	e200	4000050C		lis	R3, 0x4000	...
00000002	< 1.6us	e200	40000510		addi	R3, R3, 0x4008	...
00000002	< 1.6us	e200	40000514		lwzx	R0, R3, R0	...
00000002	< 1.6us	e200	40000518		stw	R0, 0x0(R4)	...
00000002	1.6us	e200	4000051C		b	_110	...
00000003	2.8us	e200	40000528	\CONTROLR_110	blr		}
00000004	< 2.8us	e200	400004D4	\CONTROLR_97	li	R0, 0x0	ptyCommandData->iDeviceNum = 0;
00000004	< 2.8us	e200	400004D8		stw	R0, 0x4(R31)	...
00000004	< 2.8us	e200	400004DC	\CONTROLR_98	li	R0, 0x58	ptyCommandData->ucControlChar = 'X';
00000004	< 2.8us	e200	400004E0		stb	R0, 0x0(R31)	...
00000004	2.8us	e200	400004E4	\CONTROLR_100	li	R3, 0x0	return NO_ERROR;
00000005	90.8us	e200	400004B8	\CONTROLR\WriteToDevice	stwu	R1, -0x30(R1)	{
00000006	< 92.4us	e200	400004BC		mflr	R0	...
00000006	< 92.4us	e200	400004C0		stw	R0, 0x34(R1)	...
00000006	< 92.4us	e200	400004C4		stw	R31, 0x2C(R1)	...
00000006	< 92.4us	e200	400004C8		mr	R31, R4	...

Figure 14. Code Trace Window

Instruction Watchpoint 0 (i.e. our “Start Trigger”) is shown in green (e.g. Frame 0 and 5 in the above screen-shot) and Instruction Watchpoint 1 (i.e. our “Stop Trigger”) is shown in red (e.g. Frame 4 in the above screen-shot). The **Time** column shows the time stamp of each captured frame. PathFinder only knows the absolute time for discontinuous instructions (e.g. bl, b, blr) or instructions at which a Watchpoint occurs, hence, the time for other frames is shown relative (< or >) to these frames. To quickly measure the time difference between frames, double-click on the **Time** column in the ‘reference’ frame. For example, in the below screen-shot we have set frame 5 as the reference frame by double-clicking in the **Time** column of frame 5. All other frame times are now shown relative to frame 5.



Code Trace (Code Frames: 5589)

Frame	Time	Processor	Address	Symbol	Instruction		Source Line
00000000	-90.8u	e200	400004B8	\CONTROLR\WriteToDevice	stwu	R1, -0x30(R1)	{
00000001	< -89.2u	e200	400004BC		mflr	R0	...
00000001	< -89.2u	e200	400004C0		stw	R0, 0x34(R1)	...
00000001	< -89.2u	e200	400004C4		stw	R31, 0x2C(R1)	...
00000001	< -89.2u	e200	400004C8		mr	R31, R4	...
00000001	< -89.2u	e200	400004CC	\CONTROLR_95	addi	R4, R1, 0x10	FindDeviceADDR (ucCommand,
00000001	-89.2u	e200	400004D0		bl	FindDeviceADDR	...
00000002	< -89.2u	e200	400004FC	\CONTROLR\FindDeviceADDR	clrlwi	R0, R3, 0x18	if (ucCommand < LAST_COMMAN
00000002	< -89.2u	e200	40000500		cmplwi	R0, 0x5	...
00000002	< -89.2u	e200	40000504		bge-	_109	...
00000002	< -89.2u	e200	40000508	\CONTROLR_107	rlwinm	R0, R0, 0x2, 0x0,	*puiDeviceAdr = puiDeviceAd
00000002	< -89.2u	e200	4000050C		lis	R3, 0x4000	...
00000002	< -89.2u	e200	40000510		addi	R3, R3, 0x4008	...
00000002	< -89.2u	e200	40000514		lwzx	R0, R3, R0	...
00000002	< -89.2u	e200	40000518		stw	R0, 0x0(R4)	...
00000002	-89.2u	e200	4000051C		b	_110	...
00000003	-88.0u	e200	40000528	\CONTROLR_110	blr		}
00000004	< -88.0u	e200	400004D4	\CONTROLR_97	li	R0, 0x0	ptyCommandData->iDeviceNum
00000004	< -88.0u	e200	400004D8		stw	R0, 0x4(R31)	...
00000004	< -88.0u	e200	400004DC	\CONTROLR_98	li	R0, 0x58	ptyCommandData->ucControlCh
00000004	< -88.0u	e200	400004E0		stb	R0, 0x0(R31)	...
00000004	-88.0u	e200	400004E4	\CONTROLR_100	li	R3, 0x0	return NO_ERROR;
00000005	0.0us	e200	400004B8	\CONTROLR\WriteToDevice	stwu	R1, -0x30(R1)	{
00000006	< 1.6us	e200	400004BC		mflr	R0	...
00000006	< 1.6us	e200	400004C0		stw	R0, 0x34(R1)	...
00000006	< 1.6us	e200	400004C4		stw	R31, 0x2C(R1)	...

Figure 15. Triggering between Two Events. Relative Time Display.

6. Tracing Code Execution up to an Event

In this example we demonstrate using Instruction Watchpoints to trace all e200 code execution up to a specific event (the entry to a particular function). We will use the example program:

`C:\PFMPC\Examples\Controlr\MPC5534\BIN\CONTROLR_RAM.CSO`

Load the program via PathFinder's **File|Load** menu

To capture all code execution up to the call to the function `WriteToDevice` then setup a Trigger using **Trigger|Trigger Configuration** as follows:

1. The **Trace Options** tab should be set as below i.e. uncheck **Stop Trace When Buffer Full** and set the **Vitra Trace Buffer Size** to the maximum supported size

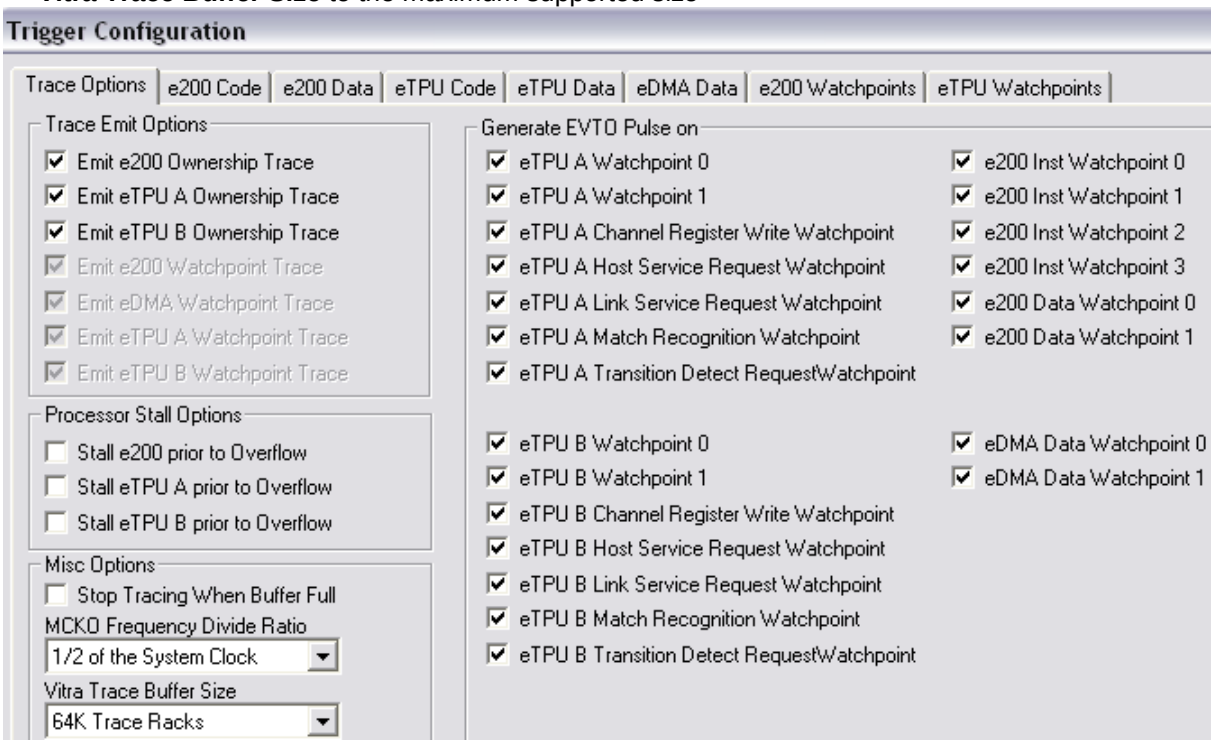


Figure 16. Trace Options Dialog

5. The **e200 Code** tab should be set to **Start Emitting Code Trace on e200 Inst Watchpoint 0** and **Stop Emitting Code Trace on e200 Inst Watchpoint 1** as shown below:

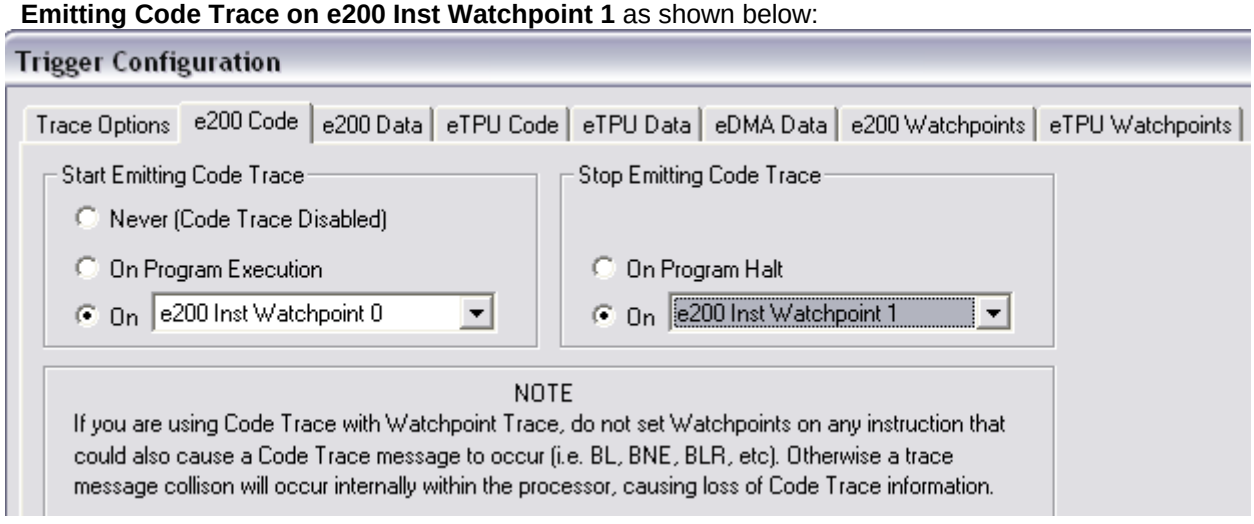


Figure 17. e200 Code Dialog

6. The **e200 Data**, **eTPU Code**, **eTPU Data**, **eDMA Data** and **eTPU Watchpoints** should be left at their default values and the **e200 Watchpoints** tab should be set as shown below.
    - Check **Allow User Control of e200 Watchpoint Resources**
    - Set **Watchpoint 0** to the entry of `main` as follows (i.e. start tracing as soon as program begins executing)

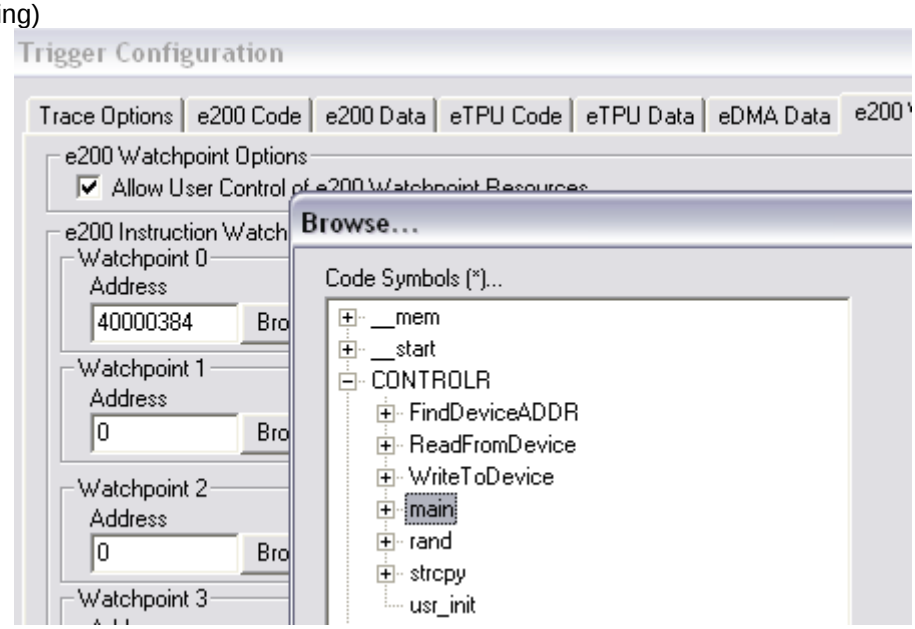


Figure 18. e200 Watchpoints Dialog (setting Watchpoint 0)

- Set **Watchpoint 1** to the entry of `WriteToDevice` (i.e. the point we want tracing to stop)

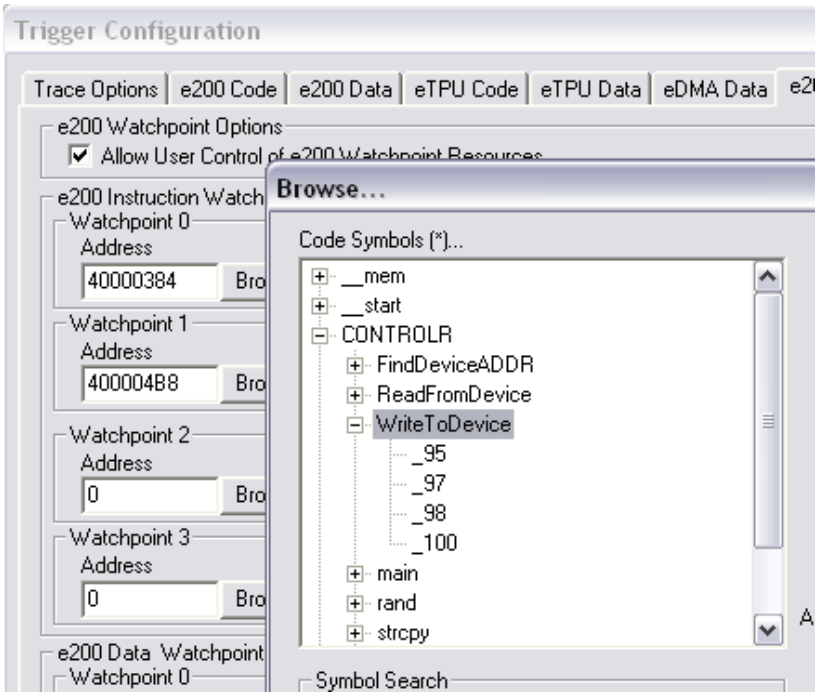


Figure 19. e200 Watchpoints Dialog (setting Watchpoint 1)

2. Click **Activate**, enable Trace (**Trace|Enable Trace**…) and run the program from reset (**Run|Go from Reset**). Halt the program after a few seconds and open the Code Trace window. All code execution up to `WriteToDevice` will be shown as below:

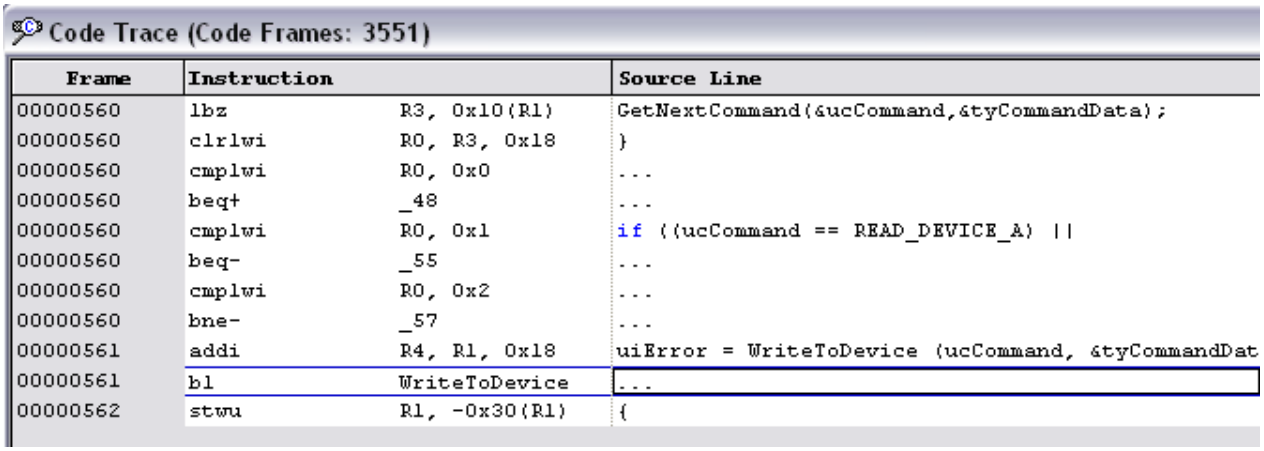


Figure 20. Code Trace Window

7. Tracing Code Execution after an Event

In this example we demonstrate using Instruction Watchpoints to trace all e200 code execution after a specific event (the entry of a particular function) until the trace buffer is full. We will use the example program:

```
C:\PFMPC\Examples\Controlr\MPC5534\BIN\CONTROLR_RAM.CSO
```

Load the program via PathFinder's **File|Load** menu

To capture all code execution after the call to the function `WriteToDevice` then setup a Trigger using **Trigger|Trigger Configuration** as follows:

3. The **Trace Options** tab can be left at default settings i.e.:

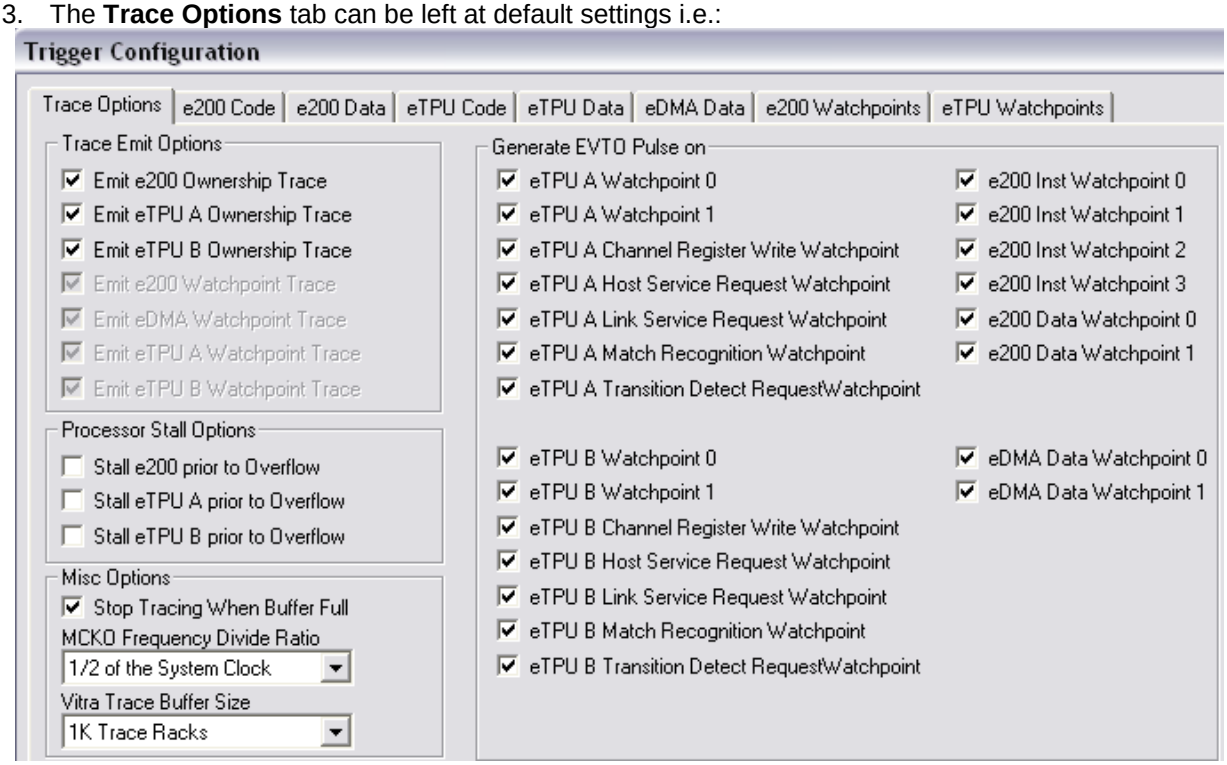


Figure 21. Trace Options Dialog

Note that the **Stop Tracing When Buffer Full** ensures we halt tracing once we have a full trace buffer (buffer size can be adjusted using **Vitra Trace Buffer Size**).

4. The **e200 Code** tab should be set to **Start Emitting Code Trace on e200 Inst Watchpoint 0** and **Stop Emitting Code Trace On Program Halt** as shown below:

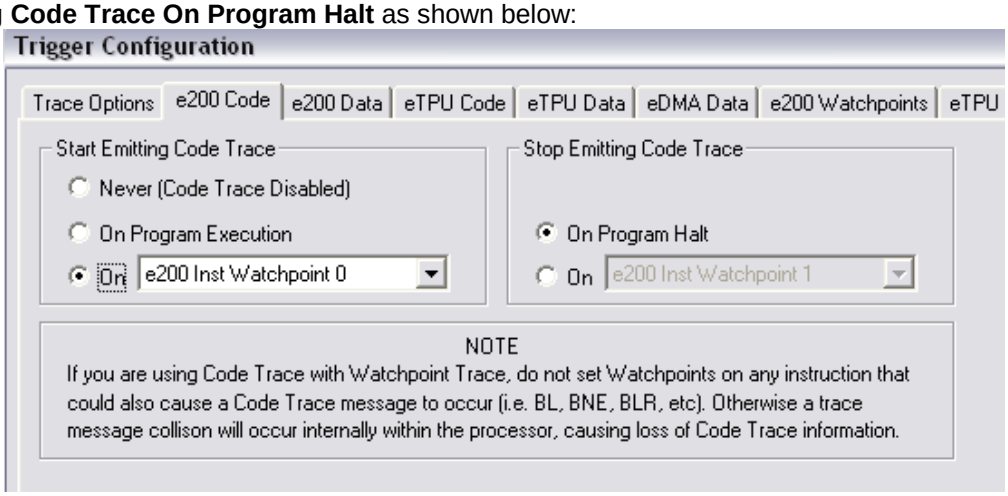


Figure 22. e200 Code Dialog

5. The **e200 Data**, **eTPU Code**, **eTPU Data**, **eDMA Data** and **eTPU Watchpoints** should be left at their default values and the **e200 Watchpoints** tab should be set as shown below.
   - Check **Allow User Control of e200 Watchpoint Resources**
   - Set **Watchpoint 0** to the entry of `WriteToDevice` i.e. the point at which we want tracing to start
   - This can be done symbolically using the **Browse…** dialog (invoke via the **Browse…** button and make sure that **Display Line Number Symbols** is checked)

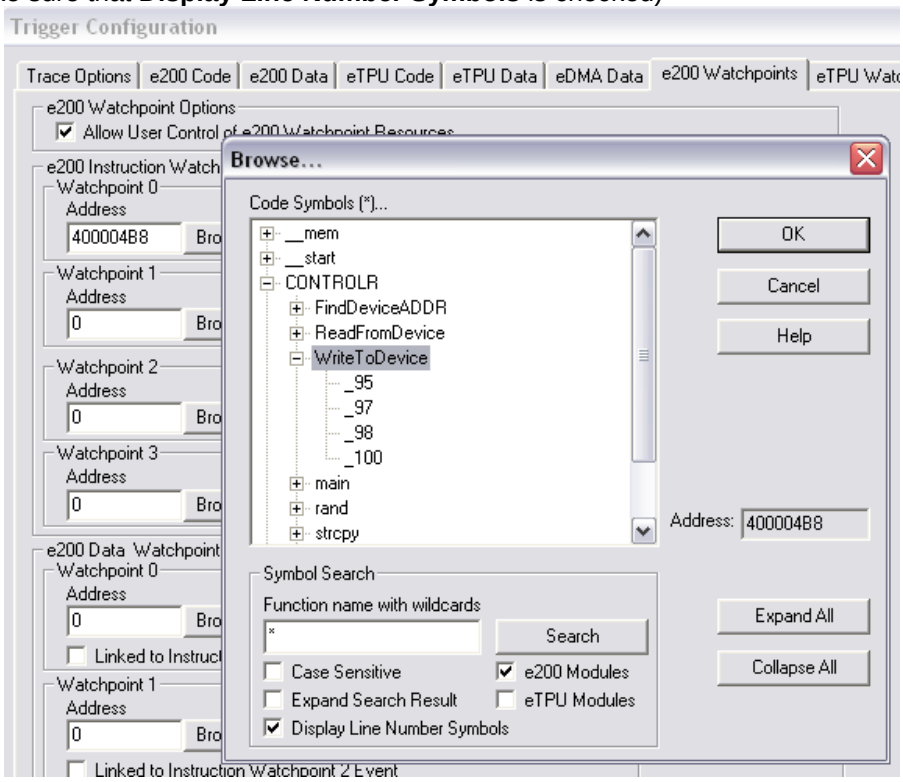


Figure 23. e200 Watchpoints Dialog (setting Watchpoint 0)

6. Click **Activate**, enable Trace (**Trace|Enable Trace**…) and run the program from reset (**Run|Go from Reset**). Halt the program after a few seconds and open the Code Trace window. All code execution from `WriteToDevice` will be shown as below:

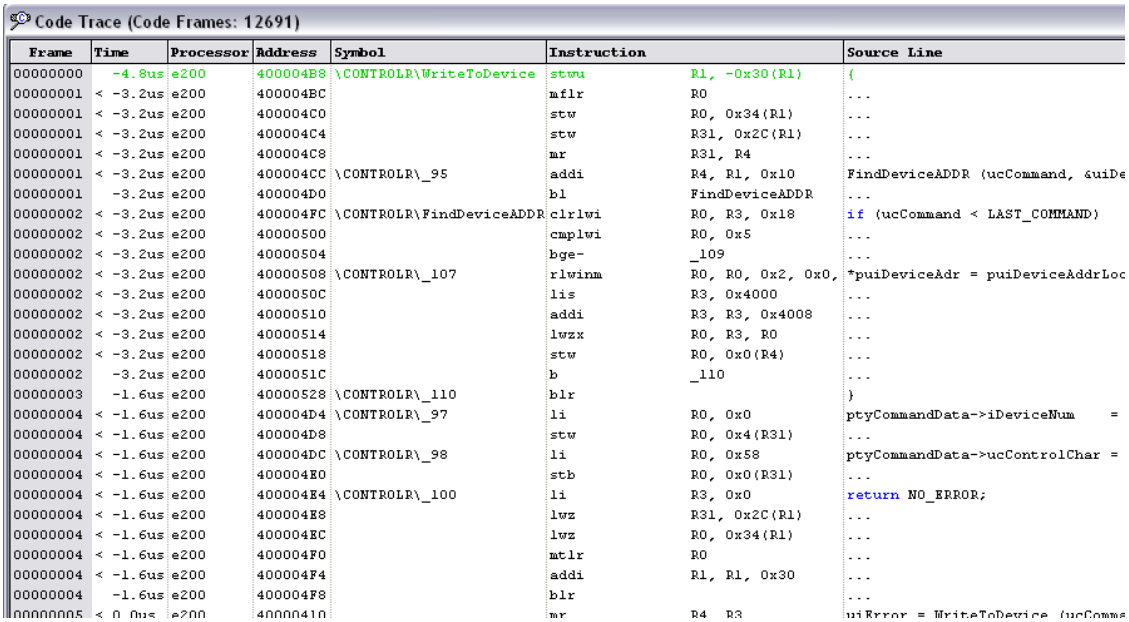
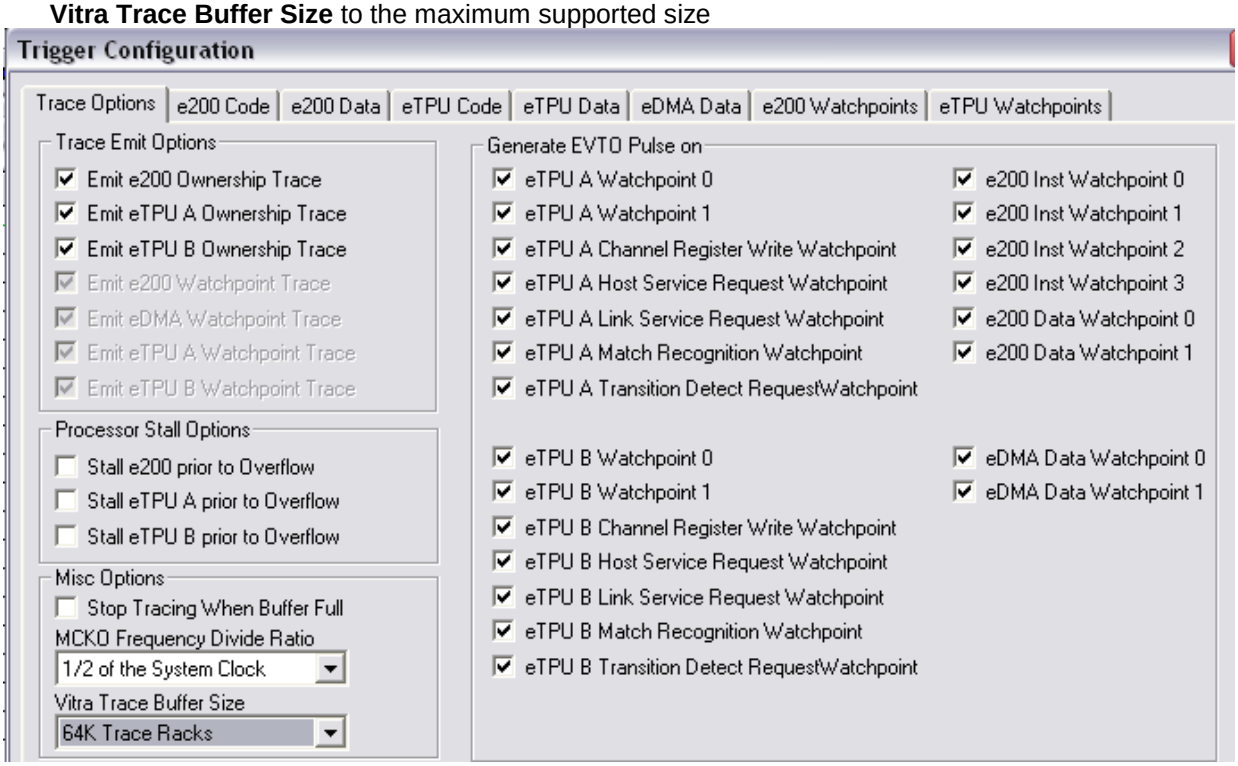


Figure 24. Code Trace Window

8. Tracing Code Execution up to Program Halt

This example shows how to trace all code execution up to program halt (e.g. your program hits a breakpoint or you halt it via PathFinder)

1. The **Trace Options** tab should be set as below i.e. uncheck **Stop Trace When Buffer Full** and set the **Vitra Trace Buffer Size** to the maximum supported size



2. The **e200 Code** tab should be set to **Start Emitting Code Trace On Program Execution** and **Stop Emitting Code Trace on Program Halt** as shown below:

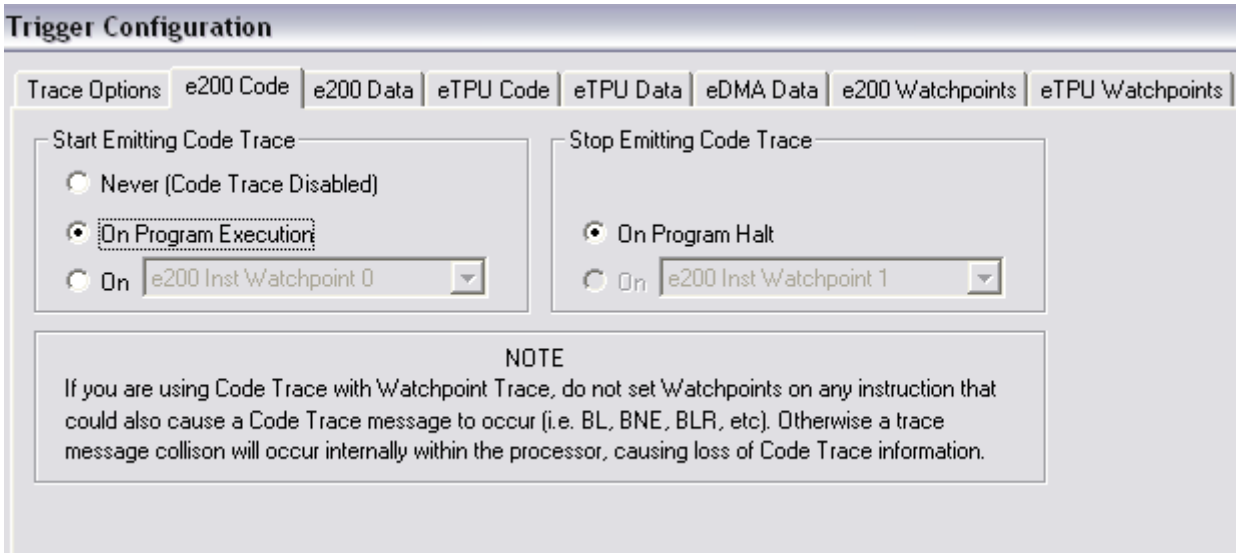


Figure 25. Tracing up to a program halt

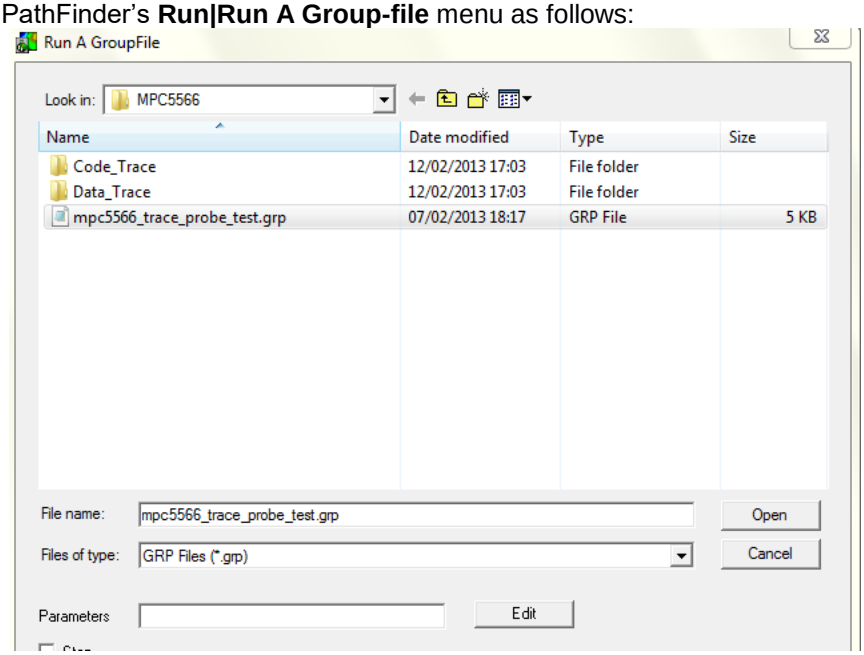
This configuration will capture trace continuously until your program halts (note that no Watchpoints are needed in this configuration).

9. Vitra Trace Diagnostics

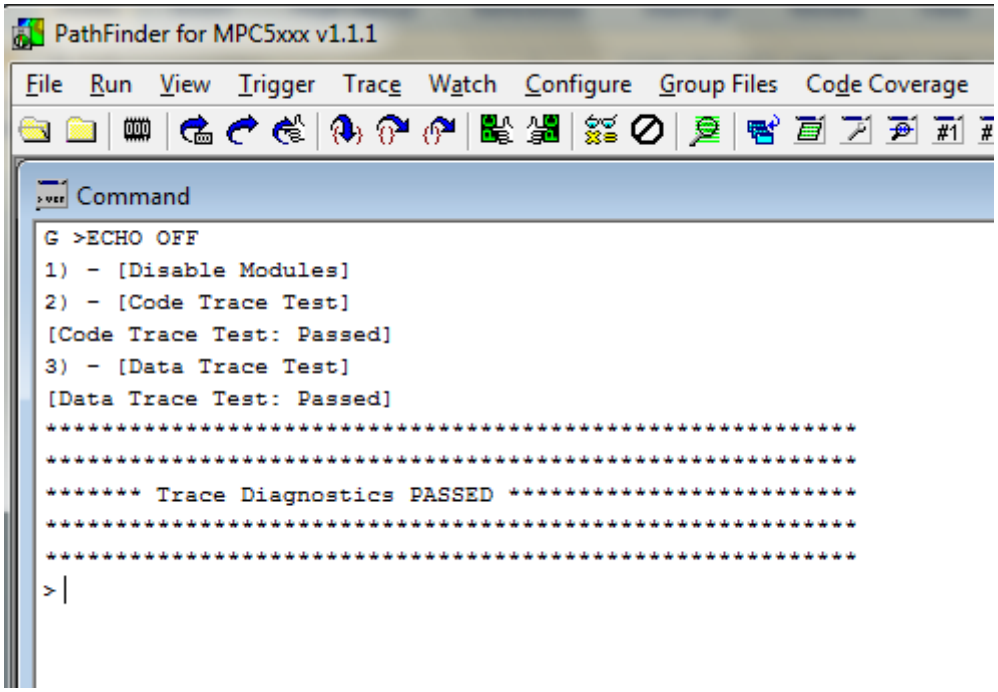
This section shows how to verify that your Vitra is correctly capturing trace data. The test will ensure that your Vitra and Target Probe Assembly (cable between your Vitra and your target system) are functional. This test requires PathFinder v1.1.1 or later. The test is included with your PathFinder software and involves running a group-file (script-file). By default, the group-file is stored in:

```
PFMPC\Ashling Trace Probe Test\MPC5566\mpc5566_trace_probe_test.grp
```

This can be run via PathFinder's **Run|Run A Group-file** menu as follows:



Browse to the group-file and select **Open**. The test will run and PathFinder will display the results in the **Command** window as follows:



If the tests fail, then:

1. Ensure Vitra is properly connected to your MPC5566 based target system and that the target system is powered up.

2. Ensure you are using with an MPC5566 based target system (other devices will not work).

3. If the problems still persist then you may have a faulty Vitra or cable; contact Ashling support on ashling.support@nestgroup.net

10. For more information…

You'll find full details on all PathFinder operations and commands in the appropriate Ashling User manuals. To keep your Ashling software up-to-date, check regularly for the latest software downloads at www.ashling.com/support/mpc5500 by following the link to **Download PathFinder-MPC5500.**

www.ashling.com/support/mpc5500

Ashling Microsystems Ltd
National Technology Park
Limerick
Ireland

Phone: +353 61 334466
Fax: +353 61 334477
Email: support@ashling.com

Doc: APB200-MPC5500Trace.doc 13th Feb 2013