

AI2 option for the PromICE

or

the Trace Board

This option adds several features to the PromICE memory emulation system. It is single add-on board that adds 1) CodeTrace, a 32-bit wide trace that is 128k or 512k deep depending on the model. 2) CodeCoverage memory to monitor and mark every location accessed in the covered space which is 512k or 2M deep depending on the model. 3) It also adds the AI virtual serial channel which has been enhanced. 4) It adds JTAG/BDM port to the PromICE for such target that have JTAG or BDM interfaces, only available on select models.

The CodeTrace memory is used for collecting trace data which can be accesses made to the ROM space and also elsewhere with the external connector. The trace memory is 128k deep or 512k deep depending on the model. There are stop and stop comparators and two 16-bit counters to skip events and record events. There is clock management for fine tuning the trace clock derived from chip_select, output_enable and the write_enable signals.

The CodeCoverage memory is up to four maps of 128k each or 512k each depending on the model. Each map can be mapped to cover and address range of 128k or 512k arbitrarily. A17-A23 or A19-A23 can be specified for the 128k or the 512k map. This memory is then dumped to get a map of location accesses. It is useful in final testing of your product for quality assurance testing.

The virtual serial channel is used for setting up debug link between the host and the target. This channel operates through the ROM socket and operates in a manner similar to the current AI option. It has been substantially enhanced to allow a more flexible operation.

All these functions are performed without affecting the target system or memory emulation process. The interface is configured by the PromICE under host control (via LoadICE application). You can use the status command to monitor the target status and trace interface status. There are commands to configure and control the trace and code coverage systems. The data is read out by the PromICE and passed to the host. The system is useful in diagnosing startup problem, reset vector fetch, state of ROM signals and location of ROM code. Then you can proceed to track real-time bugs and finish it off with code coverage testing for quality assurance.

Models with external connector allow you to 'see' more of your target space. You can supply eleven signals directly from the target system. Presumably A21-A23 and D0-D7, but they can be anything. An external clock allows you to generate trace and code coverage data from arbitrary address space

PromICE with Trace offers a complete base level diagnostic tool for firmware development. From emulation of the ROM space to virtual serial channel for debugging to real-time trace and code coverage information gives you a comprehensive set of features.

NOTE: THE COMMANDS ON THE FOLLOWING PAGES ARE AVAILABLE FULLY ONLY ON LOADICE VERSION 2.0 OR LATER FOR WINDOWS 9x AND WINDOWS NT.

4.1. tri

Initializes the trace board.

4.1.1. Command Forms

tri Dialog mode

4.1.2. Syntax

{tri} [code]

4.1.3. Use

- code - (hex) Initialized the interface operation per bits in 'code'
 The bits are encoded as follows and mean when set to a 1.
- Bit 0 - Use word size (per word command) to normalize the trace data display and
 trace data input, such as start and stop addresses.
- Bit 1 - Show full 32-bit trace data, otherwise data displayed is masked by the current
 ROM size, as well as the start and stop addresses after input are OR'd with ROM
 size mask.
- Bit 2 - Show code coverage data as one byte, instead of 1-bit. Handy if you are using
 the xm command with specific data to be stored when doing coverage.
- Bit 3 - Shows debug output, such as actual start and stop comparator values etc.

4.1.4. Default

By default the trace data is shown masked off to show relative to ROM addresses.

4.1.5. Description

This command may be used to initialize or reinitialize the trace board. It can also be used to change the display of trace data.

4.1.6. Notes

This command is automatically executed if user uses any other trace command before issuing this command.

4.1.7. Examples

tri 2 Initialize the trace board and dump full 32-bit trace data.

4.2. tr

Collect trace data.

4.2.1. Command Forms

tr Dialog mode

4.2.2. Syntax

{tr} [*fr addr to addr fl fu sk count tr count dl # rc xc*]

4.2.3. Use

- fr addr** - (hex) sets up the start comparator with addr value. Trace event happens when this address is encountered. Trace events are counted traced or skipped according the other paramters.
- to addr** - (hex) sets up the stop comparator with addr value. Trace event happens when this address is encountered. Trace events are counted traced or skipped according the other paramters.
- fl** - sets up the start and stop comparator as flags instead of range (default). The trace event then is described as having encountered the specific start or stop address, without this flag the start and stop are used as ranges, trace event then are when the address is greater or equal to start or less or equal to stop..
- fu** - sets up the trace to stop when the trace buffer is full, otherwise the trace will continue to wrap around.
- sk count** - (decimal) sets up the counter-a (16-bit) to skip this many trace events and then start tracing.
- tr count** - (decimal) sets up the counter-b (16-bit) to trace this many trace events and stop tracing.
- dl #** - manage clock de-skew. By default clock is de-skewed by 20ns and then a 60ns write pulse writes the trace data. If no arguments are given then the de-skewing is turned off all together. Otherwise values of 0, 1,2, or 3 may be used. You should use this parameter only if you suspect bogus trace data.
- rc** - By default ROM clock is made from chip_select, output_enable and write_enable signals and is used for collecting trace data. This specification allows you to specify that the ROM clock should not used. Handy when you only want to trace the external cycles. See next.
- xc** - sets up the trace to use external clock also for tracing. This is applicable only when external clock is supplied with an external header. Not all models of the trace board have external header

4.2.4. Default

By default the trace collects all accesses to ROM space by using ROM clock.

4.2.5. Description

This command is used to start the trace collection process. The various arguments provide for fine tuning the trace parameters, allowing you to trace through specific scenarios.

4.2.6. Notes

Any other trace command will stop the trace process if it is still going on.

4.2.7. Examples

```
tr fr 7fe40 to 7fe80 sk 12 tr 50
```

trace accesses between addresses 0x7fe40 and 0x7fe80, skipping the first 12 and tracing the next 50. Then stop the trace.

4.3. **trr**

Read the trace data from trace memory.

4.3.1. **Command Forms**

`trr` Dialog mode

4.3.2. **Syntax**

`{trr} [start stop]`

4.3.3. **Use**

`start, stop` - read the trace memory and display the trace data. Dump the data from `start` to `stop` locations. If only one arg is given then dump next 64 location..

4.3.4. **Default**

If no args are given then the entire trace memory is dumped.

4.3.5. **Description**

This command is used to read the trace data. It also reports the status of the trace interface at the time of the read command. The status will show whether there is any trace collected and if any of the counters overflowed etc..

4.3.6. **Notes**

This command stops the trace collection. If you just want to check on the status of the trace interface, use the `status` command (`st` on command line). This command will show you the target power and execution status as well as the AI2 interface status and the trace status. This way you will not stop the trace process and still know if trace has stopped. If `stop` is less than `start` then `stop` is considered a count of location to read.

4.3.7. **Examples**

`trr 1f00` dump trace data from 0x1f00. Dumps next 64 locations.

4.4. **trf**

Find pattern in trace memory.

4.4.1. **Command Forms**

`trf` Dialog mode

4.4.2. **Syntax**

`{trf} data [start stop]`

4.4.3. **Use**

`data` - hex value to search for in trace memory
`start, stop` - range of memory to search..

4.4.4. **Default**

If no start and stop are given then the entire trace memory is searched.
If `stop` is less than `start` then `stop` is interpreted as count.
If no stop value is given then trace memory is searched from start to end.

4.4.5. **Description**

This command is used to search the trace memory. The memory is searched to match with the given pattern. When a match is found, its location is indicated and then the block containing the location is dumped.

4.4.6. **Notes**

This command work like the `trr`. It stops tracing.

4.4.7. **Examples**

`trf f003 0 200` search pattern 'f003' in trace memory from 0 to 200.

4.5. trw

Write trace memory.

4.5.1. Command Forms

trw Dialog mode

4.5.2. Syntax

{trw} [*start, stop, data*]

4.5.3. Use

start - start filling the trace memory from start.
stop - fill trace memory till stop
data - write 32-bit data in trace memory.

4.5.4. Default

If no args are given then this command will zero out the entire trace memory. If a single argument is given then it is used as a start address and zeros are stored in the next 64 location. And if no data is supplied then zero is used.

4.5.5. Description

There is no particular reason to zero out the trace memory or write any data patterns in it. But if you wanted to do that use this command.

4.5.6. Notes

This command will destroy any trace data in memory referred. If stop is less than start then stop is considered a count of locations to write.

4.5.7. Examples

trw 1f0 write zeros to trace from 0x1f0 to next 64 locations

4.6. trs

Save the trace data to a file.

4.6.1. Command Forms

trs Dialog mode

4.6.2. Syntax

{trs} [*filename* [*start*, *stop*]

4.6.3. Use

filename - specifies the filename to use for saving the trace data

start, *stop* - specifies the range to data to be saved

4.6.4. Default

save to file called *trace.log* and save the entire trace memory,

4.6.5. Description

This command is used to save the trace memory data to a file. The data is stored in text format same as the dump on the screen.

4.6.6. Notes

Overwrites any previous file of the same name.

4.6.7. Examples

trs mytrace.log Save the trace memory data to file *mytrace.log*.

4.7. **trt**

Test the trace memory.

4.7.1. **Command Forms**

trt Dialog mode

4.7.2. **Syntax**

{trt}

4.7.3. **Use**

no args are needed.

4.7.4. **Default**

none.

4.7.5. **Description**

This command is used to test the trace memory. It down-loads a non-repeating pattern and then reads the trace memory and compares to see that all is okay..

4.7.6. **Notes**

This command destroys trace data filling memory with random data.

4.7.7. **Examples**

trt Test the trace memory reporting any errors.

4.8. xmi

Initializes the trace board.

4.8.1. Command Forms

xmi Dialog mode

4.8.2. Syntax

{xmi} [*code*]

4.8.3. Use

- code - (hex) Initialized the interface operation per bits in 'code'
 The bits are encoded as follows and mean when set to a 1.
- Bit 0 - Use word size (per word command) to normalize the trace data display and
 trace data input, such as start and stop addresses.
- Bit 1 - Show full 32-bit trace data, otherwise data displayed is masked by the current
 ROM size, as well as the start and stop addresses after input are OR'd with ROM
 size mask.
- Bit 2 - Show code coverage data as one byte, instead of 1-bit. Handy if you are using
 the xm command with specific data to be stored when doing coverage.
- Bit 3 - Shows debug output, such as actual start and stop comparator values etc.

4.8.4. Default

By default the code coverage data is shown as if it were single bit wide.

4.8.5. Description

This command may be used to initialize or reinitialize the trace board. It can also be used to change the display of trace and code coverage data.

4.8.6. Notes

This command is automatically executed if user uses any other trace command before issuing this command.

4.8.7. Examples

xmi 4 Initialize the trace board and show code coverage data as byte wide

4.9. xm

Start code coverage.

4.9.1. Command Forms

xm Dialog mode

4.9.2. Syntax

{xm} [[m0| m1|m2|m3 addr] data]

4.9.3. Use

m*n* addr - (hex) sets up to use one or more maps for code coverage as specified by the address byte. The address byte specifies A16-A23 address mask for each map.
Data - (hex) byte of data to be used as pattern for code coverage, by default the data stored is 0xFF. Also use 'tri 4' to display this data

4.9.4. Default

Nothing happen by default, you must specify what map to use where.

4.9.5. Description

This command is used to collect code coverage data. There are four maps of 128k bytes each. They can be mapped anywhere to respond to addresses A16-through A23. However since each map is 128k A16 is ignored. Value of other addresses depends on whether they are supplied by the ROM socket or by external header.

4.9.6. Notes

You will need to zero out the map before using this command. Use xmw command for that purpose. Any other xm command will stop the process if it is still going on.

4.9.7. Examples

```
xm m0 fe
```

This will set up the map 0 to be mapped to respond to A17-A23 as high (A16 is ignored). This will cause the map to perform code coverage for the top (128k) page of a 27040 ROM or the entire 27010 ROM. Going back to the 27040 ROM, the following will set up the second map to cover page 0 of the ROM (0-0x1fff)

```
xm m1 f8 22 - note that A17 and A18 are zero.to select page 0, pattern is 0x22
```

4.10. **xmr**

Read the code coverage data from code coverage memory.

4.10.1. **Command Forms**

`xmr` Dialog mode

4.10.2. **Syntax**

`{xmr} [m0|m1|m2|m3 [start stop]]`

4.10.3. **Use**

`mn` - read the code coverage memory from given map. Displaying only those blocks where non-zero bits are present.

`start,stop` read only between the stop and start addresses, if no stop is given read a block of 256 locations. When either the stop or start is used, the code coverage memory is treated as one contiguous chunk, in other words the second map starts at 0x20000 and so on.

4.10.4. **Default**

If no args are given then the entire code coverage memory is dumped.

4.10.5. **Description**

This command is used to read the code coverage data. It will start by dumping the first block of 256 and then it will dump only those blocks that have non-zero values in them.

4.10.6. **Notes**

This command stops the code coverage process. If stop is less than start then stop is considered a count of locations to dump.

4.10.7. **Examples**

`xmr m0` dumps the code coverage map 0.

4.11. xmw

Clear code coverage memory.

4.11.1. Command Forms

xmw Dialog mode

4.11.2. Syntax

{xmw} [[m0|m1|m2|m3 [start, stop]]

4.11.3. Use

m*n* - clear the specified map, writing zeros to it all.
start - start clearing the code coverage memory from start.
stop - fill memory till stop

4.11.4. Default

If no args are given then this command will zero out the entire code coverage memory. If a single argument is given then it is used as a start address and zeros are stored in the next 256 location. If a map is specified then just that map is cleared

4.11.5. Description

You must use this command to zero out the code coverage memory and thus initializing the memory for use.

4.11.6. Notes

This command will destroy any code coverage data in memory referred. If stop is less than start then stop is considered a count.

4.11.7. Examples

xmw m0 initialize the map 0 for code coverage

4.12. xms

Save the map data to a file.

4.12.1. Command Forms

xms Dialog mode

4.12.2. Syntax

{xms} [*filename* [*m0*/*m1*/*m2*/*m3* [*start*, *stop*]]

4.12.3. Use

filename - specifies the filename to use for saving the map data

mn - save only the map specified to the file

start, *stop* - specifies the range to data to be saved

4.12.4. Default

save to file called *coverage.log* and save the entire map memory.

4.12.5. Description

This command is used to save the map memory data to a file. The data is stored in text format same as the dump on the screen.

4.12.6. Notes

Overwrites any previous file of the same name.

4.12.7. Examples

xms mymap.log Save the trace memory data to file *mymap.log*.

4.13. xmt

Test the code coverage memory.

4.13.1. Command Forms

xmt Dialog mode

4.13.2. Syntax

{xmt}

4.13.3. Use

no args are needed.

4.13.4. Default

none.

4.13.5. Description

This command is used to test the code coverage memory. It down-loads a non-repeating pattern and then reads the memory and compares to see that all is okay..

4.13.6. Notes

This command destroys code coverage data filling memory with random data.

4.13.7. Examples

xmt Test the trace memory reporting any errors.

4.14. st

Modified status command reports status of trace also.

4.14.1. Command Forms

st Dialog mode

4.14.2. Syntax

{st} [1]

4.14.3. Use

1 when 1 is supplied as argument then LoadICE monitors status continuously displaying the results as they change. Use <CR> to break it from the loop.

4.14.4. Default

Show status once.

4.14.5. Description

This command is used to observe the status of the target system and the trace interface. For target system the display is one of the following:

```
TARGET STATUS: Unit(s) in LOAD mode
TARGET STATUS: Power is OFF
TARGET STATUS: Power is ON - Target is accessing ROM
TARGET STATUS: Power is ON - Target is NOT accessing ROM
```

The status of target accessing ROM is determined by the PromICE by polling the data bus and seeing it change. It is not an absolute test of target operations.

For the AI2 inter face the status is reported and the following information is displayed:

AI2 INTERFACE STATUS: (followed by one or more of the following)

XINT	Interrupt to target asserted (break or AI channel). This will happen only when enabled. This interrupt is asserted on the back connector and also by the PromICE (via micro-code) on the 'int+&-' pins on the PromICE back panel.
TRACESEEN	Some trace has been collected in the buffer
TRACESTOPPED	Trace collection has stopped per some condition
XTRACESTOPEd	Trace generated by 'fine-grain' control has stopped. This is an alternate use of the code coverage memory. It is not yet implemented in LoadICE.
BREAKSEEN	Either 'start' or 'stop' comparator generated a break.
XBREAKSEEN	Fine-grain trace control generated a break. Another feature to be available with fine grain trace.
AIHDA	AI virtual serial channel, Host Data Available flag
AITDA	AI Target Data Available flag

TRACE STATUS: (followed by one or more of the following)(

TraceSeen	Some trace has been collected
TraceStopped	Some even has stopped trace collection
TraceBufferFull	The trace memory has reached its end, it is full of trace. The 'fu' option was specified in the 'tr' command. Other

	wise the trace buffer rolls over to top and starts overwriting itself.
<code>SkipCounterOver</code>	The event skip counter has overflowed, means it did skip the desired number of events.
<code>TraceCounterOver</code>	The trace counter has counted out the events as programmed and the <code>TraceStopped</code> will also be set.
<code>BreakSeen</code>	When the 'start' and 'stop' comparator are programmed to cause a break by asserting interrupt (XINT if enabled). Any case this bit indicates that break condition has occurred.

`CurrentTracePointer`: Displays the current value of the trace pointer.

4.14.6. Notes

Use this command to help you monitor target status and to find out if the trace command you issued to collect the trace (*tr*) has completed. The *trr* command for reading the trace will stop whatever tracing was going on. Use the current pointer value to figure out where the latest trace data is.

4.14.7. Examples

```
st          Report status once.
St 1       Monitor status till key-board key is hit.
```

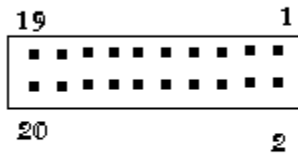
Using External Connector for the Trace Board

The AI2 trace board optionally has an external connector for connecting auxiliary signals to provide a wider range of trace options. Specifically this connector allows you to connect up to 11 external data signals and one external clock signal for use in trace or code coverage functions. In addition there are two output signals which are interrupts asserted by the trace system to indicate break conditions etc. to the target. Both the high asserted and the low asserted signals are provided. In addition there are a couple of ground pins and some unused pins.

The pins of the 20 pin IDC header are defines as follows:

<u>Pin</u>	<u>Signal</u>	<u>Use</u>
1	CA21	external address A21 - input
2	CA22	external address A22 - input
3	CA23	external address A23 - input
4	CA24	external data D0 - input
5	CA25	external data D1 - input
6	CA26	external data D2 - input
7	CA27	external data D3 - input
8	CA28	external data D4 - input
9	CA29	external data D5 - input
10	CA30	external data D6 - input
11	CA31	external data D7 - input
12	XCLK	external clock (low asserted) - input
13	xintout+	interrupt from AI2 to target (high asserted) - output
14	xintout-	interrupt from AI2 to target (low asserted) - output
15	(unused)	
16	(unused)	
17	(unused)	
18	(unused)	
19	GND	extra ground connection to target
20	GND	extra ground connection to target

The connector is oriented as the rest of the connector on the PromICE back panel, with pin 1 on the right top. See the diagram below:



Use the full 32 bit display mode in the trace command. Use the command option 'xc' to specify use of external clock for tracing, and 'rc' option to disable the internal (ROM access) clock.

Some notes on using the trace to find startup problems with your target system

When you plug your PromICE in the ROM socket and down-load some code and then boot your target and nothing happens, what do you do? Do you look through the ini file and check your specs? Do you change the ROM size or specify the socket statement or otherwise dump the ROM locations to see why the target is not booting up properly? You could just pick up the phone and call tech support. Well, if you have the AI2 option with your PromICE then you could just take the trace and figure out for your self what could be going on, better than anyone else can help you, the trace can!

To start with you can find out where your target is booting from. Simply take the trace and look. To start with use the following command:

```
tri 1
```

This command will initialize the trace board and set the option to dump the entire 32-bit of trace data. You want to see what state the unused address lines are in. Even though you may be emulating a particular ROM size, the socket could be wired for larger ROMs, or for some other reason the address line is not making through the cable or socket of buffers within the PromICE. This will help you identify any and all such problems.

Then issue this command to simply collect some trace

```
tr fu
```

This command will cause unconditional collection of trace till the trace memory is full. This way you will avoid writing over the good trace data if your target falls into infinite loop or some other piece of code. You can boot the target after this command has been issued to make sure you can capture the reset vector. You may hold the reset button on your target down, type the command, and release the reset. The trace buffer will show you the trace starting from target boot sequence.

If you have a problem with miss-wiring of address lines or the socket is wired for bigger ROM etc. then the trace will show that. Look at the trace data to determine if the reset vector fetch sequence is correct and if the unused lines are all high. Then you can follow the trace, if the target continue to run in some strange state, you will see the offending addresses as you will see the trace veer off to no where land. If the target is not able to fetch a good reset vector, then you may only see a few trace cycles. In any case by comparing the trace signals to how you think your target is wired or where you think your code is located, it is a short path to get your target to boot properly and get past the startup issues.

What constitutes an event for trace counters?

```
tr fr 74311 tr 20
tr fr 74311 fl tr 20
```

Consider the above two commands. They both say start tracing when address 74311 is seen, actually the first one says trace any address greater than or equal to 74311 and will trace 20 such events.

The second command says use the address 74311 as a flag, in other words start tracing when that exact address is seen, and keep tracing until that address (74311) is seen 20 times.

So the amount of trace collected in first case is 20, since it will record 20 addresses greater than or equal to 74311. The second command will trace some indefinite amount since it will only stop when it has seen the address 74311 itself 20 times.