atollic

White paper

Recording and analyzing execution history using ETM/ETB instruction tracing

COPYRIGHT NOTICE

TRADEMARK

Atollic, ***Atollic TrueSTUDIO*** and the Atollic logotype are trademarks or registered trademarks owned by Atollic. ECLIPSE™ is a registered trademark of the Eclipse foundation. ARM and Cortex are trademarks or registered trademarks of ARM Ltd. All other product names are trademarks or registered trademarks of their respective owners.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment of Atollic AB. The information contained in this document is assumed to be accurate, but Atollic assumes no responsibility for any errors or omissions. In no event shall Atollic AB, its employees, its contractors, or the authors of this document be liable for any type of damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

DOCUMENT IDENTIFICATION

ASW-WPETM	April 2015

REVISION

First version	April 2015
Second version	August 2016

Atollic AB
Science Park
Gjuterigatan 7
SE- 553 18 Jönköping
Sweden

+46 (0) 36 19 60 50

E-mail: sales @atollic.com
Web: www.atollic.com

Atollic Inc
241 Boston Post Road West (1st Floor),
Marlborough,
Massachusetts 01752
USA

+1 (973) 784 0047 (Voice)
+1 (877) 218 9117 (Toll Free)
+1 (973) 794 0075 (Fax)

E-mail: sales.usa@atollic.com
Web: www.atollic.com

Contents

Tables

No table of figures entries found.

ABSTRACT

ARM® Cortex® microcontrollers continue to push the price/performance ratio to unprecedented levels. This allows more complex embedded systems to be designed, comprising more software. And hence, the problem of debugging complex error situations is increased too.

This white paper outlines how the ETM and ETB instruction tracing technology can be used for advanced debugging of Cortex-M applications. Instruction tracing enables developers to record execution history for later analysis, which can be a great help in certain situations.

INTRODUCTION

Finding bugs in software is a difficult and time consuming process. Debugging is usually more difficult in many embedded applications for several reasons. Execution timing is often critical, and results can be skewed by the inclusion of instrumentation code. In cases of motor, actuator or other control applications, it may not be possible, or particularly revealing to stop the device under test for debugging operations.

The ETM and ETB debugger technologies from ARM, found in Cortex-M microcontroller product families, provides instruction tracing that enables developers to record the execution history for later analysis. This is made possible by the internal architecture of the devices, and in the case of ETM, also adding the requirement of an ETM trace-enabled debugger probe, such as SEGGER J-Trace.

This white paper gives a brief background to the Embedded Trace Macrocell (ETM) and Embedded Trace Buffer (ETB) technologies. It also outlines how advanced debuggers can exploit the ETM and ETB trace technology to deliver powerful debug capabilities to developers.

Many embedded developers continue to use “tried and true” methods of debugging such as blinking LED and printf() outputs. When the ease of use, information content and the cost efficiency of modern debugging technology is considered, it is well worth a developer’s while to devote the relatively small amount of effort to becoming familiar with these methodologies as the payback in time savings in debugging and testing will be considerable.

INSTRUCTION TRACING OVERVIEW

Instruction tracing is valuable in very tricky debug scenarios, and is considered the “heavy guns” of embedded debugging. The purpose is to record execution history, and later analyze what really happened when the system malfunctioned. Some types of applications need instruction tracing, as physical hardware can be damaged if you stop on a breakpoint during debugging. Motor control applications are a typical example here.

To use instruction tracing in compatible ARM processors, a couple of different technologies are available. Their respective roles will be explained below.

ETM TRACING

Embedded Trace Macrocell (ETM) tracing requires an ETM trace-enabled debugger probe, such as SEGGER J-Trace. Such debugger probes typically have a large on-probe trace buffer to store the recorded execution history, or even uses real-time streaming to the host PC, and are thus more expensive compared to standard debugger probes without ETM trace support. Trace buffers used with ETM tracing are typically many MB or GB in size.

To use ETM tracing, a 20-pin JTAG connector is used where 5 pins are dedicated for ETM tracing (1 clock pin and 4 data pins).

ETB TRACING

Embedded Trace Buffer (ETB) tracing do not need any particular trace-enabled debugger probe, but instead uses a (very much) smaller trace buffer in RAM of the Cortex-M device. ETB can thus be used without the more expensive trace-enabled debugger probes with compatible Cortex-M devices, but the execution time that can be recorded is a lot shorter. Trace buffers used with ETB are typically very small, only a couple of KB, as they use the RAM of the microcontroller device as a trace buffer.

To use ETB tracing, no particular extra JTAG pins are needed, as the trace buffer is just uploaded from the target RAM just like any other array of bytes.

TRACE TRIGGERS

Because recording the execution history of a processor running at full speed generates massive amounts of data (easily many gigabytes of trace data on a Cortex-M core during only a couple of seconds of execution time), great care is taken to reduce the amount of trace data that is recorded.

Trace-enabled debuggers therefore support configuration of where in the code, or at what events, trace recording shall start and stop. This is to reduce the massive volumes of trace data recorded.

RECORDING AND ANALYZING EXECUTION HISTORY WITH ATOLLIC TRUESTUDIO PRO

Atollic TrueSTUDIO Pro® supports ETM and ETB instruction tracing, provided that trace-enabled hardware is being used. Instruction tracing records the execution flow of the processor in real-time. The recorded trace buffer can then be uploaded to the debugger and analyzed to locate the cause of software errors. Instruction tracing is particularly useful when debugging problems that only occur sporadically, or when stopping on a breakpoint will damage physical hardware.

Both ETM and ETB tracing records all executed machine code instructions, until the hardware buffer limits are reached. A trace buffer is filled very quickly even though it is highly compressed. The compressed trace buffer in a JTAG probe with a 16MB of trace buffer typically expands into 200MB of uncompressed machine readable data, and to 2-3GB of human readable data. Instruction tracing thus quickly generates a huge amount of data. This can happen in a fraction of a second on a speedy processor.

ENABLE TRACE

Instruction tracing (using ETM or ETB) must be enabled in the debug configuration of the debugger. To enable instruction tracing, first open the debug configuration dialog box:

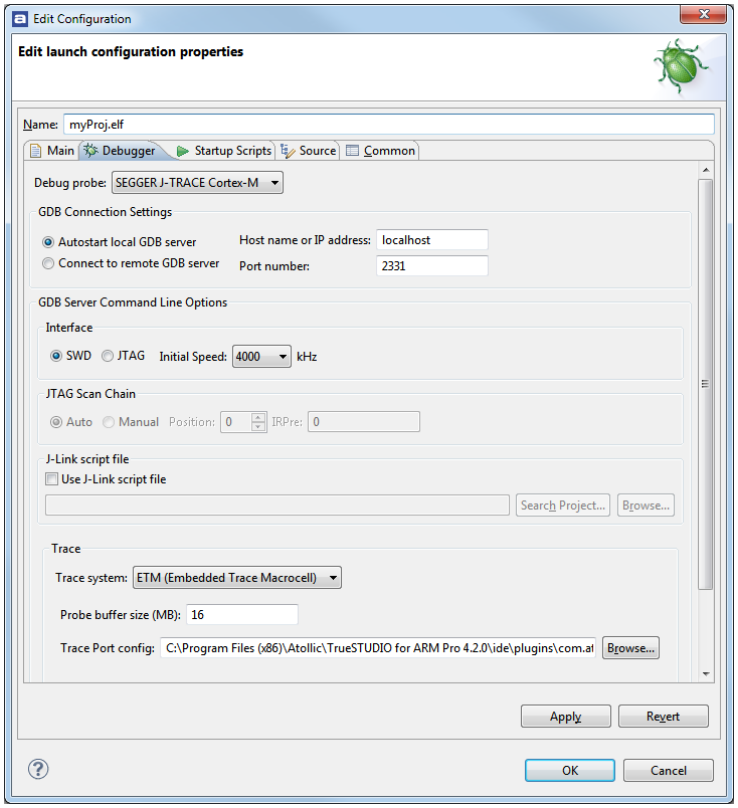

Figure 1 – Enable Tracing in the Debug Configuration

Then, select ETM (requires a trace enabled debugger probe like SEGGER J-Trace) or ETB (can be used with any debugger probe, provided the device supports ETB) trace system as appropriate.

CONFIGURING THE TRACING SESSION

Once the debugger probe and trace system have been configured, and a debug session has been started, the tracing can be configured. To configure trace, suspend the debug session and open the **Trace Log** view (Select **View** in the top menu and then **ETM/ETB**, **Trace Log)**.

In the **Trace Log** view toolbar, click on the **Configuration** toolbar button.

Figure 2 – Configuration Toolbar Button

The **Trace Configuration** dialog box will be displayed:

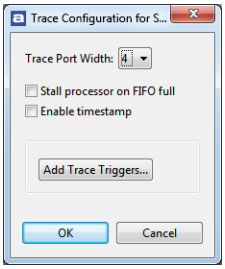

Figure 3 - Trace Configuration

Configure the **Trace Port Width** dropdown list to match the number of pins used for ETM tracing on the hardware board.

Using the **Stall processor on FIFO full** checkbox, select one of these two options:

- Stall the processor when the ETM trace FIFO buffer becomes full. With this setting, no trace data is lost but the timing behavior of the application can be changed.

- Do not stall the processor when the ETM trace FIFO buffer becomes full. With this setting, the processor will always continue to run at full speed but trace data may be lost.

Some devices support timestamps. Enabling the timestamps can be useful if timing information is needed. It will however reduce the amount of other information available.

TRACE TRIGGERS

The trick with Instruction tracing is to trace only where tracing is needed. Otherwise the important information can be impossible to locate in the huge amount of data that will be collected or lost since it occurred to long time before debugging was suspended and the trace information uploaded.

There are four hardware triggers that can be set to starting and stopping the tracing on different conditions. To access them, open the **Trace Configuration** as above and select **Add Trace Triggers…**

The triggers can also be added from the **Breakpoints** view.

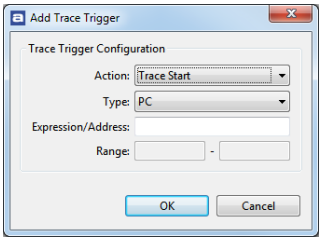

Figure 4 - Trace Configuration

For each of the triggers 0-3, it is possible to define that the trigger shall start or stop tracing, if its configured conditions are met. Each trigger has the following options:

- The **Action** to perform when the condition is triggered:
  - **Trace Start**: Starts collecting trace data
  - **Trace Stop**: Stops collecting trace data
- The **Type** of memory access that triggers the action:
  - **PC**: Triggered when execution reaches an address
  - **Data Read**: Triggered when data is read from an address
  - **Data Write**: Triggered when data is written to an address
  - **Data Read/Write**: Triggered when data is read or written to an address
- Enter the address to trigger on in the **Expression/Address** field. This field accepts:
  - Numeric address constants such as 0xffff0010
  - Numeric address ranges such as 0xffff0010 to 0xffff001f

- Function symbols such as “main”
- Variable symbols such as “MyGlobalCounter”
- It is also possible to define mathematical expressions like “main + 7”

Typically, at least one trigger is configured to start tracing, and another trigger is configured to stop tracing.

ADD TRACE TRIGGER IN THE EDITOR

Start and Stop Trace Triggers can also be added directly in the **C/C++ Editor** ruler and the **Disassembly** view. These triggers work in line with the Breakpoints, except that they will not suspend the execution. Instead they will start or stop collecting of trace data when execution reaches that line. In short, they are “trace start/stop points” in the code.

Right click on the ruler to the left in the editor window and select **Add Trace Trigger**.

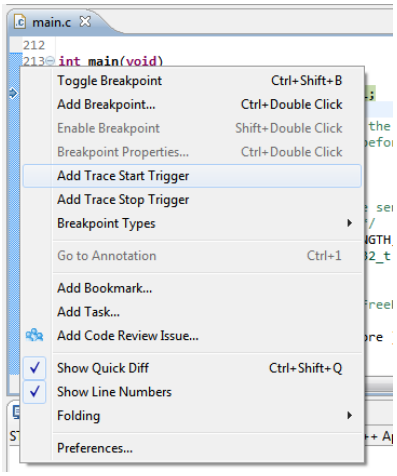

Figure 5 – Add Trace Trigger in the Editor

A new trigger will be created and tracing starts to be collected when execution reaches that line of code.

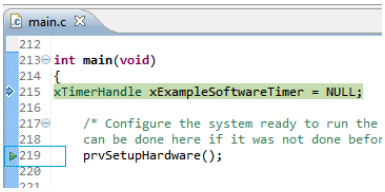

Figure 6 –Trace Trigger in the Editor

MANAGING TRACE TRIGGERS

All the Trace Triggers are visible from the **Breakpoints** view.

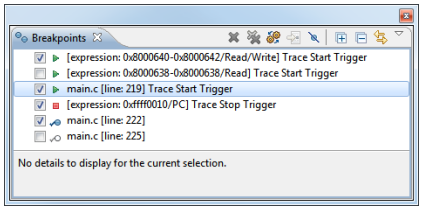

Figure 7 –Trace Trigger in the Editor

From this view the triggers can easily be inactivated, activated, removed and even added. Bear in mind that the hardware supports up to a maximum of four simultaneous Trace Triggers.

START TRACE RECORDING

Once tracing has been enabled, click the **Record** toolbar button in the **Trace Log** view to enable recording of trace data.

Figure 8 – Record Toolbar Button

With trace recording enabled, start target execution. When execution is suspended, the **Trace Log** view is filled with the recorded instruction trace (provided the trace start trigger condition was fulfilled).

ANALYZING THE TRACE

When later on suspending execution, the trace buffer is uploaded to the **Trace Log** view. It is filled with the recorded instruction stream, along with other data that is provided by analyzing the trace recording.

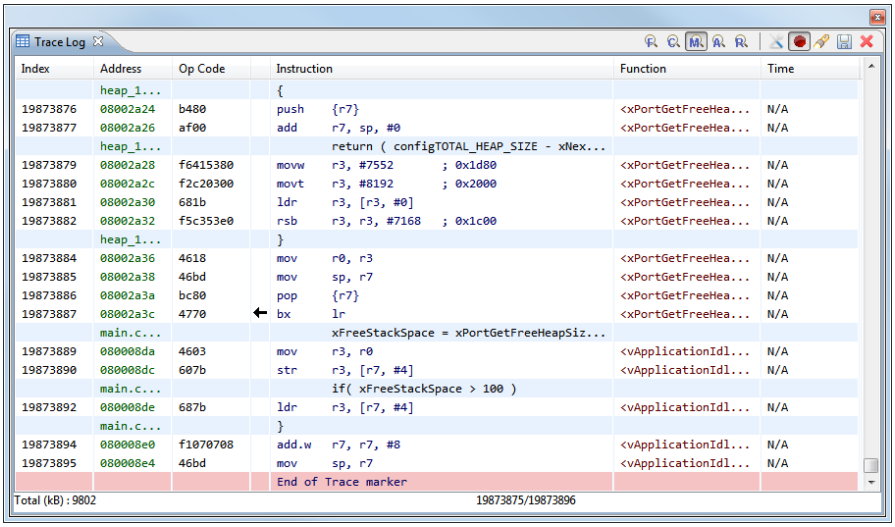

Figure 9 - The Trace Log View

The **Trace Log** view shows detailed information on what the processor was doing up to the point of suspending execution.

Please note the column with graphical icons that annotate the **Trace Log** view with information about execution flow branches:

- Call a new function
- Return from a function
- Jump up in the code
- Jump down in the code
- Iterate on the same instruction
- A conditional branch was not taken

At the end of the view is the **End of Trace marker** displayed. This is added to the Trace Log each time the buffer is overflowed and it indicates that some trace data most likely is lost.

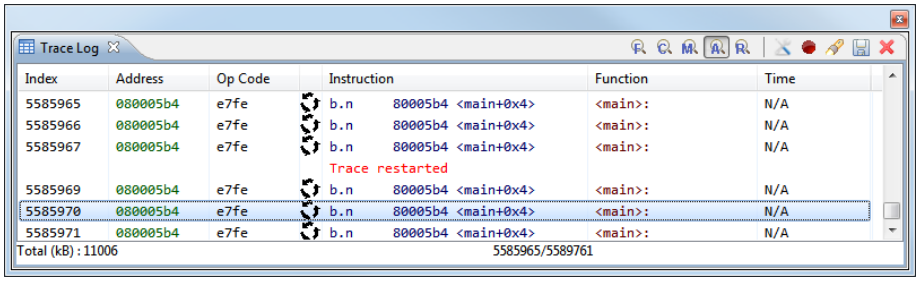

Figure 10 - Trace Restarted

The other important marker is the Trace restarted marker. It indicates that the target wasn't able to generate all the trace information without affecting the performance of the running application. Some data is lost.

To overcome this issue, enable **Stall processor on FIFO full** in the **Trace Configuration**.

DISPLAY OPTIONS

The **Log Trace** view supports several different display options ("zoom" levels):

- Function call tracing
- C tracing
- C/Assembler mixed mode tracing
- Assembler tracing
- Raw trace packet log

Use the different Display Options Toolbar Buttons to switch between the different view-modes.

The Function call tracing displays what function the execution is in and from where it is called or returned from.

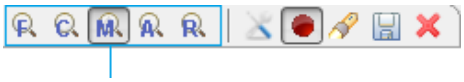

Figure 11 – Display Options Toolbar Button

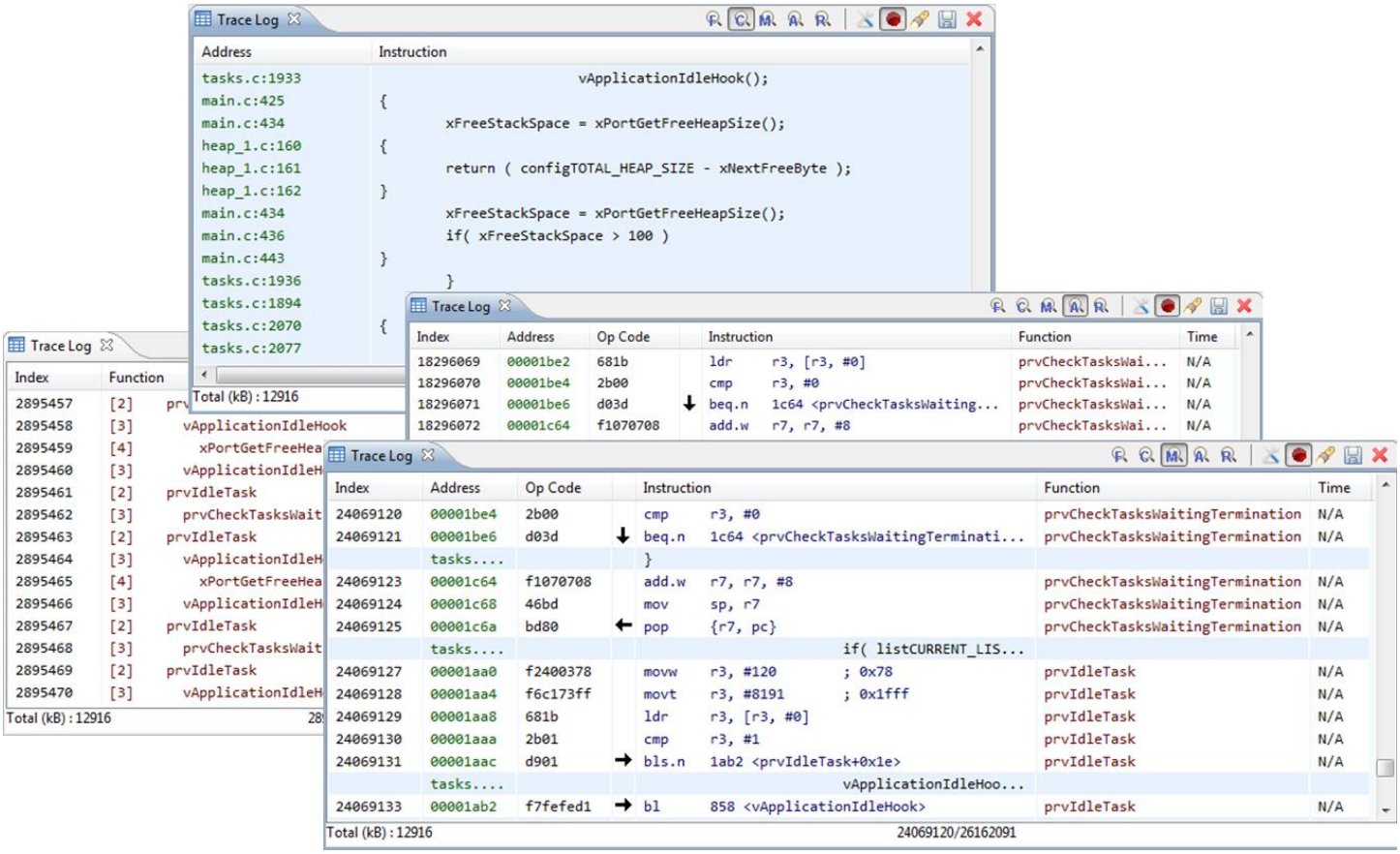

Figure 12 - Different “zoom” levels in the instruction trace log

SEARCH THE TRACE LOG

The recorded trace buffer can become very large. ***Atollic TrueSTUDIO Pro*** supports appended trace buffers of a total of 100 million lines. For this reason, a search function is available, to enable users to find important information in the potentially huge dataset.

Figure 13 – Search Toolbar Button

Using the search feature it is possible to search for certain data of particular interest. For example, assume a system crash sometimes happens because a variable has an illegal value. By searching the instruction trace for the address of the variable, it is possible to understand what code modifies the value and gives it the illegal value causing a system crash.

EXPORTING A TRACE LOG

It is possible to save the trace log by clicking on the **Export Trace** toolbar button in the **Trace Log** view.

Figure 14 – Export Toolbar Button

The trace log can be saved to either comma separated value files (*.csv) that can be imported into Microsoft® Excel®, or to human readable ASCII text files (*.txt).

Configure the trace record range to export using the **From Index** and **To Index** fields.

As the saved trace log becomes approximately 200 times larger than its compressed size in the JTAG probe trace buffer, the saved trace log can optionally be split to many files in order to avoid exported trace logs which are several gigabytes in size (for example, the 16MB compressed trace buffer in Segger J-Trace expands to 2-3GB when saved to a human readable trace log file in *.CSV or *.TXT formats).

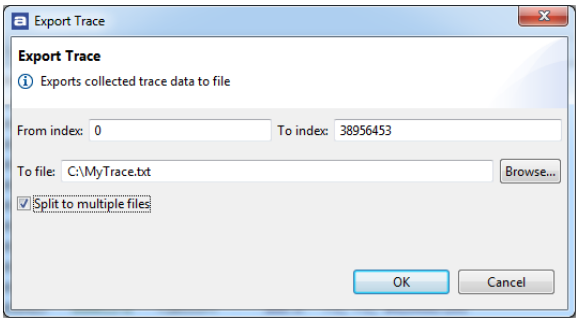

Figure 15 - Exporting the Trace Log

After a trace log has been exported into readable format, very complex problems can be analyzed offline, for example by writing a script program that programmatically analyzes the execution history, thus automatically scanning millions of executed machine code instructions, trying to find that odd reason for the “million dollar bug”.

SUMMARY

Many of the hardware components of electronics products are becoming less expensive as time goes on. This trend is actually driving up software development and testing costs, as the adoption of cheaper hardware enables more product features that must be supported by software. This means more bugs to find, and less time to find them. The information provided by ETM or ETB instruction tracing can be valuable tools in a developer's repertoire when used properly.

More information about Atollic and the ***Atollic TrueSTUDIO*** product is available here:

www.atollic.com

www.atollic.com/truestudio