Fast Models Tarmac Trace

Version 7.1

User Guide

ARM®

ARM DUI 0532D (ID051712)

Fast Models Tarmac Trace
User Guide

Release Information

Change history

Description	Issue	Confidentiality	Change
October 2010	A	Non-Confidential	First release for Fast Models v6.0.
May 2011	B	Non-Confidential	Update for Fast Models v6.1.
November 2011	C	Non-Confidential	Update for Fast Models v7.0.
May 2012	D	Non-Confidential	Update for Fast Models v7.1.

Proprietary Notice

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

`http://www.arm.com`

Contents
Fast Models Tarmac Trace User Guide

Preface

This preface introduces the *Fast Models Tarmac Trace User Guide*. It contains the following sections:

About this book

This document describes the use of the Fast Models tarmac trace plug-in from ARM®, and the format of the trace files it generates.

Intended audience

This book has been written for experienced hardware and software developers to aid the development of ARM-based products using Fast Models as part of a development process.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

This chapter describes the main features of Fast Models tarmac trace.

Chapter 2 *Tarmac Trace Plug-in*

This chapter describes how to set up the environment to make use of the Fast Models tarmac trace plug-in, and how to start a simulation. It also describes the parameters that control the type of events that are traced.

Chapter 3 *Tarmac Trace File Format*

This chapter provides information on the Fast Models tarmac trace file format.

Chapter 4 *Tarmac Example*

This chapter contains an example of the tarmac trace output.

Glossary

The *ARM Glossary* is a list of terms used in ARM documentation, together with definitions for those terms. The *ARM Glossary* does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See *ARM Glossary*, `http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html`.

Typographical conventions

The typographical conventions are:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`monospace` **italic**	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
`monospace bold`	Denotes language keywords when used outside example code.

< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`

Additional reading

This section lists related publications by ARM and by third parties.

See Infocenter, http://infocenter.arm.com for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. See the following documents for other relevant information:

- *Fast Models User Guide* (ARM DUI 0370)
- *Fast Models Reference Manual* (ARM DUI 0423).

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to `errata@arm.com`. Give:

- the title
- the number, ARM DUI 0532D
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter describes the main features of Fast Models tarmac trace. It contains the following sections:

1.1 Overview

Fast Models v5.1 onwards supports the generation of traces that consistently track the execution and related activities in the model, particularly those that affect the state of the modeled IP. Generated virtual platforms provide trace support by using plug-ins in the form of DLLs and shared objects on Windows and Linux, respectively. Using the plug-in, trace information is written to a file in textual form in the format described in this document.

ARM provides a plug-in to produce a textual trace output (tarmac) as described in Chapter 3 *Tarmac Trace File Format*. Other plug-ins, using the *Model Trace Interface* (MTI), can be used instead, or at the same time. Figure 1-1 shows how MTI is embedded into Fast Models. You can connect various plug-ins to this interface in the form of a shared object loaded at simulation startup.

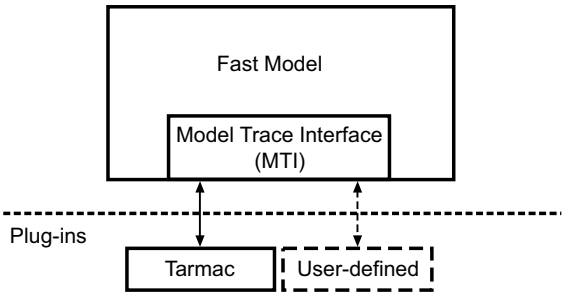

Figure 1-1 Interaction between MTI and plug-ins

This document describes:

- how to enable and disable tarmac trace
- how to control tarmac trace
- file formats and how to analyze the output.

Chapter 2
Tarmac Trace Plug-in

This chapter describes how to set up the environment to use the tarmac trace plug-in, and how to start a simulation. It also describes the parameters that control the type of events that are traced. It contains the following sections:

2.1 Getting Started

This section provides information on specifying the location of the trace plug-ins.

2.1.1 Pointing to the position of the tarmac trace plug-in

In instances of running Fast Models with RVD, or when using SystemC-based platforms, you specify the trace plug-in that is to be loaded on simulation start by setting the environment variable `FM_TRACE_PLUGINS`.

This must point to the full path of the tarmac trace plug-in. On Linux, for `sh` users this might be, for example:

```
export FM_TRACE_PLUGINS
/home/<user>/<installation_path>/plugins/<platform>/TarmacTrace.so
```

On Windows the full path might be, for example:

```
C:\Program
Files\ARM\FastModelPortfolio_X.Y\plugins\Win32_VC2005\Release\TarmacTrace.dll
```

If multiple plug-ins are to be used at the same time, separate them by ‘;’. You can also load the same plug-in multiple times. You can give a name for the plug-in instance by prefixing `instancename=` to the plug-in path or paths.

2.2 Starting the simulation

There are a two ways to start a simulation and use the tarmac trace plug-in. You can:

- load the simulation library, for example `RTSM_EB_CortexA8.so`, by a debugger (see *Running the simulation with RVD* and *Running the simulation with Model Debugger*)
- start it in standalone mode without a debugger (see *Running the simulation without a debugger* on page 2-4).

2.2.1 Running the simulation with RVD

To run a simulation in RVD:

1. In RVD, access the **Connect to Target** dialog box.
2. Set the connection type as **RTSM**.
3. To add a new configuration, select the **Add** button and point to the location of the simulation library.
4. After connecting to the target, set the parameters, as shown in Figure 2-1.

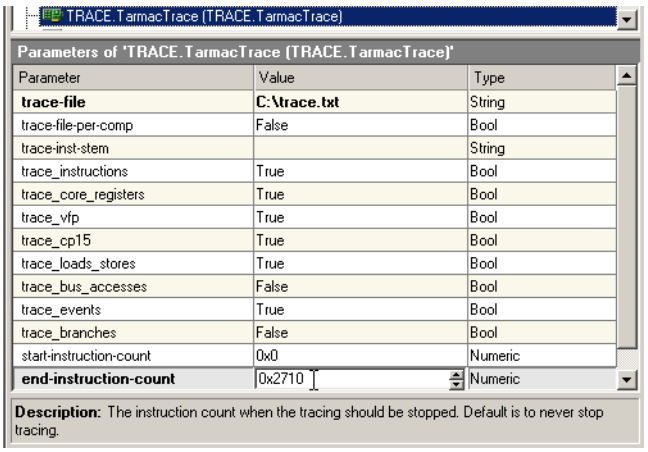

Figure 2-1 Setting parameters

   You might set, for example, the name and location of the trace output file and the trace end count value.

5. Load the application file.

For more information on using the tools, see the *RealView Debugger User Guide*.

2.2.2 Running the simulation with Model Debugger

To run a simulation in Model Debugger:

1. Ensure that you have specified the tarmac trace plug-in that is to be loaded on simulation. See *Pointing to the position of the tarmac trace plug-in* on page 2-2.
2. In Model Debugger, select **File** → **Load Model...**
3. Set the file path to the simulation library.
4. Set the model parameters in the Configure Model Parameters dialog box, as shown in Figure 2-2 on page 2-4.

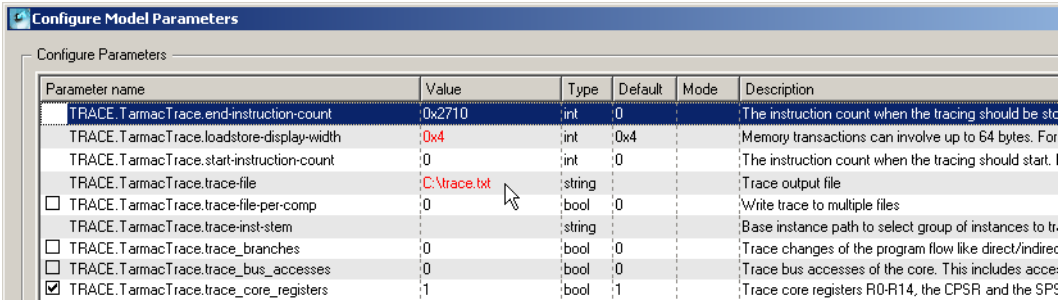

Figure 2-2 Setting model parameters

You might set, for example, the trace end count value and the name and location of the trace output file.

5. Load the application file.

For more information on using the tools, see the *Model Debugger for Fast Models User Guide*.

2.2.3 Running the simulation without a debugger

The simulation library must be started using Model Shell. This tool permits running arbitrary simulation targets like the Cortex-A8 based EB platform. The corresponding executable `model_shell` is located in the `bin` directory of the installation. It provides several options to set parameters and load application files. The most convenient way to set parameters is to use a configuration file.

To generate this configuration file, start `model_shell` with the `--list-params` option and the simulation library. For Linux this is:

```
model_shell --list-params <path_to_simulation_library> > params.config
```

The configuration file for the parameters can have any arbitrary name and can be edited using a normal text editor to set the parameter values. For Linux, the `<path_to_simulation_library>` looks like:

```
./<installation_path>/lib/Linux-Release-GCC-4.1/lib/RTSM_EB_CortexA8.so
```

For Win32 platforms:

```
./<installation_path>/Win32-Release-VC2005/lib/RTSM_EB_CortexA8.dll
```

For Linux, the simulation library might be started using the parameter configuration file with the following command:

```
model_shell <path_to_simulation_library> -f params.config -a <application_file.axf>
```

For Windows, the command is similar:

```
model_shell.exe <path_to_simulation_library> -f params.config -a <application_file.axf>
```

The `--help` option lists all available options for `model_shell`.

Note

Use the `-C, --parameter PARNAME=VALUE` option to set individual parameters on the `model_shell` command line. This permits priority over parameters specified in a parameter file.

2.3 Parameters

Configure the tarmac trace plug-in using the parameters listed in Table 2-1. The parameters appear prefixed with the path `TRACE.instancename`, where instance-name is `TarmacTrace` unless overridden. See *Starting the simulation* on page 2-3.

Table 2-1 Parameter descriptions

Parameter name	Type	Default Value	Description
`trace-file`	String	Empty	Name of the trace output file. If empty (default) the trace output is printed on `stdout`. If `STDERR` the trace output is printed on `stderr`.
`trace-file-per-comp`	Boolean	`False`	Create a separate trace file for each component traced. At present the only components that support trace are cores, so this option is only relevant when there are multiple cores. The component name is added to the trace file name to disambiguate it.
`trace-inst-stem`	String	Empty	If set to a component path only a sub tree of components is traced. In the simplest case this can be set to the component path of a single CPU then only this CPU is traced.
`trace_instructions`	Boolean	`True`	Determines whether instructions should be traced.
`trace_core_registers`	Boolean	`True`	Determines whether core registers (R0-R14, CPSR and SPSR) should be traced. This produces a lot of data and can considerably slow down simulation.
`trace_vfp`	Boolean	`True`	Determines whether VFP and NEON registers (including FPSCR and FPEXC) should be traced.
`trace_cp15`	Boolean	`True`	Determines whether writes to CP15 registers should be traced.
`trace_branches`	Boolean	`False`	Trace all non-sequential changes of the program flow. The information traced is sufficient to completely reconstruct program flow, and the tracing is fairly efficient.
`trace_bus_accesses`	Boolean	`False`	Trace all bus accesses. This forces all direct memory accesses to turn into full transaction which considerably slows down the simulation.
`trace_loads_stores`	Boolean	`True`	Determines whether load/stores are traced. This is much cheaper performance-wise than bus tracing.

Table 2-1 Parameter descriptions (continued)

Parameter name	Type	Default Value	Description
`trace_events`	Boolean	`True`	Determines whether exceptions and mode changes (for cores implementing modes) are traced.
`start-instruction-count`	Integer	`0x0`	Set the instruction count where tracing starts. Default `0x0` is to start from the beginning.
`end-instruction-count`	Integer	`0x0`	Set the instruction count where tracing ends. Default `0x0` is to trace until the end of the simulation.
`loadstore-display-width`	Integer	`0x4`	Memory transactions can in the case of LDM/STM involve up to 64 bytes. For easier readability you can break these up into multiple memory access records with a smaller size of bytes. `0` means not to break up any transaction. The default `4` means to break up transactions into words.

Chapter 3
Tarmac Trace File Format

This chapter describes the Fast Models tarmac trace file format. It contains the following sections:

3.1 Instruction trace

If enabled, this trace source generates one record for every instruction being executed.

Records (lines) of the instruction trace provide related information in the following command syntax:

```
<time> <scale> [IT|IS] (<inst_id>) <addr> <opcode> [A|T|X] <mode>_<security> : <disasm>
```

The fields have the following meaning:

`<time>` Timestamp (decimal value).

`<scale>` Unit for the previous field `<time>`. `clk` is used to indicate the timestamp is not related to real time, but an increasing count.

`[IT|IS]` This field set to `IT` indicates that the instruction passed the condition code (taken). This field set to `IS` indicates that the instruction failed the condition code (skipped).

`<inst_id>` The tick count of this core. This is equivalent to the number of instructions executed, except for certain instructions like `WFI`/`WFE` (decimal value).

`<addr>` Address from where this instruction was fetched, in hexadecimal format (virtual address).

`<opcode>` 16-bit/32-bit hexadecimal opcode of the instruction.

`[A|T|X]` Current instruction set:

- `A` represents an ARM instruction
- `T` represents a Thumb instruction
- `X` represents a Thumb-2EE instruction.

`<mode>` Processor execution mode (`svc`, `irq`, `fiq`, `usr`, `mon`, `sys`, `abt`, `und`).

`<security>` Processor security state (`s` or `ns`).

`<disasm>` Disassembly of the instruction executed.

3.2 Program flow trace

If enabled, every executed branch instruction triggers this trace source. This is a more efficient way to reconstruct the program flow than by tracing every instruction.

Branch trace records have the following command syntax:

```
<time> <scale> [FD|FI|FR] (<inst_id>) <addr> <targ_addr> [A|T|X]
```

The fields have the following meaning:

`<time>` Timestamp (decimal value).

`<scale>` Unit for the previous field `<time>`. This is used for consistency with device-specific tarmac trace formats.

`[FD|FI|FR]` This is a program flow change by:
- a direct branch `FD`
- an indirect branch `FI`
- a return from exception `FR`.

`<inst_id>` The tick count of this core. This is equivalent to the number of instructions executed, except for certain instructions like `WFI`/`WFE` (decimal value).

`<addr>` Address from where this instruction was fetched, in hexadecimal format (virtual address).

`<targ_addr>` The (virtual) address at which the execution continues.

`[A|T|X]` The instruction set after the branch:
- `A` represents an ARM instruction
- `T` represents a Thumb instruction
- `X` represents a Thumb-2EE instruction.

Note

This event is not shown in the trace example file

3.3 Register trace

If enabled, all writes to the CPU registers are traced. This includes writes to core registers R0 to R14, CPSR and SPSR, VFP registers such as S0 to S31, D0 to D31, FPSCR, FPEXC, and writes to CP14 and CP15 registers. Banked registers are traced separately using the mode as a suffix to the register name, for example r13 (current register R13) and r13_mon (banked register R13).

Register traces have the following command syntax:

```
<time> <scale> R <register> <value>
```

The fields have the following meaning:

`<time>` Timestamp (decimal value).

`<scale>` Unit for the previous field `<time>`. This is used for consistency with device-specific tarmac trace formats.

`<register>` Register name in lowercase letters. Banked core registers can have a mode appended with a single underscore. Banked CP14/CP15 registers have `_s` or `_ns` appended to indicate access of either the secure or non-secure banked register.

`<value>` Hexadecimal value written to the register (64 bits maximum).

3.4 Event trace

If enabled, this source traces exceptions and interrupts occurring.

Event traces have the following command syntax:

```
<time> <scale> E <value> <number> <desc>
```

The fields have the following meaning:

`<time>` Timestamp (decimal value).

`<scale>` Unit for the previous field `<time>`. This is used for consistency with device-specific tarmac trace formats.

`<value>` Hexadecimal representation of a value associated with the event.

`<number>` Event number.

`<desc>` Event name.

Supported values for the `value`, `number` and `desc` fields are detailed in Table 3-1:

Table 3-1 Supported values

Number	Event description	Value
00000001	CoreEvent_Reset	-
00000002	CoreEvent_UndefinedInstr	-
00000003	CoreEvent_SWI	SWI number
00000004	CoreEvent_PrefetchAbort	-
00000005	CoreEvent_DataAbort	-
00000007	CoreEvent_IRQ	-
00000008	CoreEvent_FIQ	-
0000000E	CoreEvent_ImpDataAbort	-
00000019	CoreEvent_ModeChange	New mode

3.5 Core memory access trace

If enabled, core data accesses are traced.

Memory traces are provided in the following command syntax:

```
<time> <scale> M<rw><sz><attrib> <addr> <data>
```

The fields have the following meaning:

`<time>` Timestamp (decimal value).

`<scale>` Unit for the previous field `<time>`. This is used for consistency with device-specific tarmac trace formats.

`<rw>` `R` indicates a read access, and `W` indicates a write access.

`<sz>` Size of the data transfer in bytes (1, 2, 4, 8).

`<attrib>` Optional access attribute:

- `X` indicates an exclusive access
- `T` indicates a translated (unprivileged) access
- `L` indicates a locked access (SWP, SWPB instructions).

`<addr>` Virtual address used to access memory in hexadecimal format.

`<data>` Hexadecimal value of data transferred. The data is padded according to the size of the transfer.

3.6 Memory bus trace

If enabled, transactions initiated through the memory bus master port of the core are traced.

These accesses use physical addresses, and are traced in the following command syntax:

```
<time> <scale> B<rw><sz><fd><lk><p><s> l<wrcbs> O<wrcbs> <master_id> <addr> <data>
```

The fields have the following meaning:

`<time>` Timestamp (decimal value).

`<scale>` Unit for the previous field `<time>`. This is used for consistency with device-specific tarmac trace formats.

`<rw>` `R` indicates a read access, and `W` indicates a write access.

`<sz>` Size of the data transfer in bytes.

`<fd>` `I` indicates an opcode fetch, `D` indicates a data load/store or an MMU access.

`<lk>` `L` indicates a locked access, `X` indicates an exclusive access, an underscore "_" indicates a normal access.

`<p>` `P` indicates a privileged access, an underscore "_" indicates a normal access.

`<s>` `S` indicates a secure access, `N` indicates a non-secure access.

`I<wrcbs>` The inner cache attributes. See `O<wrcbs>`.

`O<wrcbs>` The outer cache attributes:

- **<w>** `W` indicates allocate on write. An underscore "_" indicates no allocate on write
- **<r>** `R` indicates allocate on read. An underscore "_" indicates no allocate on read
- **<c>** `C` indicates a cacheable access. An underscore "_" indicates a non-cacheable access
- **<b>** `B` indicates a bufferable access. An underscore "_" indicates a non-bufferable access
- **<s>** `S` indicates a shareability access. An underscore "_" indicates a non-shareability access.

`<master_id>` The master ID of the transaction.

`<addr>` Physical address used to access memory in hexadecimal format.

`<data>` Hexadecimal value of data transferred. The value is padded according to the size of the transfer. Bytes are ordered from lowest to highest byte. This means that for accesses in little endian mode, the data occurs mirrored compared to the register/memory access records.

Note

This event is not shown in the trace example file.

Chapter 4
Tarmac Example

This chapter contains an example of the Fast Models tarmac trace file format.

Example 4-1 Example trace file produced by the tarmac trace plug-in

```
10 clk IT (10) 00001088 e89d00ff A mon_ns : LDMIA sp,{r0-r7}
10 clk MR8 00103fbc 0000000000000060
10 clk MR8 00103fc4 0010400000000000
10 clk MR8 00103fcc 0000000000004000
10 clk MR8 00103fd4 0000000000000000
10 clk R r0 00000060
10 clk R r1 00000000
10 clk R r2 00000000
10 clk R r3 00104000
10 clk R r4 00004000
10 clk R r5 00000000
10 clk R r6 00000000
10 clk R r7 00000000

11 clk IT (11) 0000108c e28dd03c A mon_ns : ADD sp,sp,#0x3c
11 clk R r13_mon 00103ff8
12 clk IT (12) 00001090 f8bd0a00 A mon_ns : RFEIA sp!
12 clk MR8 00103ff8 0000001300000000
12 clk R r13_mon 00104000
12 clk R cpsr 00000013
12 clk E 00001090 00000019 CoreEvent_ModeChange

25 clk IS (25) 000010c0 13a00000 A svc_ns : MOVNE  r0,#0

26 clk IT (26) 000010c4 eee80a10 A svc_ns : FMXR FPEXC,r0
```

```
26 clk R fpexc 01c00000

27 clk IT (27) 000010c8 ed236a06 A svc_ns : FSTMDBS r3!,{s12-s17}
27 clk MW8 00104000 4455667700112233
27 clk MW8 00104008 ccddeeff8899aabb
27 clk MW8 00104010 89abcdef01234567
27 clk R r3 00104000

33 clk IT (33) 00001200 ed334b08 A abt_s : FLDMDBD  r3!,{d4-d7}
33 clk MR8 00105000 2222333300001111
33 clk MR8 00105008 6666777744445555
33 clk MR8 00105010 aaaabbbb88889999
33 clk MR8 00105018 eeeeffffccccdddd
33 clk R d4 2222333300001111
33 clk R d5 6666777744445555
33 clk R d6 aaaabbbb88889999
33 clk R d7 eeeeffffccccdddd
34 clk IT (34) 00001204 f3ba01c2 A abt_s : VZIP.32 q0,q1
34 clk R d0 487201bf46b94bfb
34 clk R d1 37cf1ce11c667e81
34 clk R d2 37200f47ff6abddf
34 clk R d3 2313de569e2cfb54
47 clk IT (47) 00001240 5a0a T abt_s : LDRH r2, [r0,r1]
47 clk MR2 00105000 1111
47 clk R r2 00001111
```