Germany France Great Britain Italy USA China Japan Lauterbach GmbH Lauterbach S.A.R.L. Lauterbach Ltd. Lauterbach Srl Lauterbach Inc. Suzhou Lauterbach Technologies Co. Ltd. Lauterbach Japan Ltd.



DEBUGGER, REAL-TIME TRACE, LOGIC ANALYZER

LONG-TERM TRACE ETMv3

Implementation

Lauterbach, the leading manufacturer of real-time trace tools, is introducing its new long-term trace for the ARM ETMv3. The aim of this innovation is to enable greatly extended measurement times for TRACE32 profiling and code-coverage functions.

This article describes the concepts of the long-term tracing technology as well as the technical requirements it places on the trace tool and the respective host computer.

ARM ETMv3

Tracing means recording detailed information about a program as it runs on the core. This information is usually generated by on-chip trace logic. For ARM cores, this logic element is known as the *Embedded Trace Macrocell* or ETM. The latest version of this logic, the ETMv3, can be found today in most ARM11 and Cortex cores. Since the functionality of the on-chip trace logic is the basis for the trace data, let's start with a brief introduction.

The ETMv3 generates a package-oriented trace log. The following information can be generated at program runtime and collected in trace packages:

- **Program flow packages:** Contain information about the program instructions executed by the core mainly the target addresses of jumps as well as the number of instructions executed between two jumps.
- Data flow packages: Contain the memory addresses read/written by the program as well as the respective data values.
- **Context-ID packages:** Contain a process/task ID in the event that an operating system is running.

The trace packages are output by the on-chip trace logic via the trace port. The trace port for the ETMv3 typically consists of 8 or 16 pins for the trace packages plus two pins for the control signals.

To minimize the bandwidth for the package output, the ETMv3 compresses the trace packages. For example, all addresses are shortened by a special algorithm. However, if the data volume is higher than the maximum bandwidth of the trace port, the FIFO buffer can overflow and some trace packages can be lost.

Compressing the trace information alone is not enough to prevent FIFO overflows. The next stage is provided by the programmability of the ETMv3. To reduce the number of trace packages, you can simply define what trace information you want generated and output. For example, no data flow information is needed for the TRACE32 profiling functions. This is very useful because it is mainly the data packages that inflate the trace volume at the port.



Classical tracing currently consists of two steps that are carried out consecutively:



1. Recording

The trace packages are sampled at the trace port and placed in the trace memory.

2. Analysis

The trace packages are transferred from the trace memory to the host, where they are decompressed and analyzed.

The classical tracing method is limited in that only the section of the program that fits within the buffer memory can be saved for evaluation and analysis. The memory depth of the TRACE32 trace tools is currently between 1 GBytes and 4 GBytes. This allows the recording of up to 3 G trace packages.

Recording

In classical tracing, the real challenge is the recording of the trace packages. Since ARM cores today usually run at frequencies up to 1 GHz, only a fast trace port can guarantee the loss-free output of all trace packages.

The Lauterbach trace tools for the parallel ETMv3 support package recording at a frequency of up to 275 MHz DDR and can therefore handle the following data rates (see Figure 1):

- 8.8 GBit/s with 16 pins for the trace package
- 4.4 GBit/s with 8 pins for the trace packages



Fig. 1: The trace tool for the parallel ETMv3 supports a data rate of 8.8 GBit/s with 16 pins for the trace packages

With the serial trace tools for the ETMv3, data rates of up to 20 GBit/s can be recorded.

Analysis

To analyze the recorded program section, you have to transfer the trace packages from the trace memory to the host, decompress and then evaluate them.

Since the trace packages for the program flow contain no program code, this has to be added prior to analysis. The following data is used:

- The program code read by the TRACE32 software from the target system memory over the JTAG interface.
- Symbol and debug information loaded by the user for the TRACE32 software.



Long-term tracing is implemented by transferring the trace packages to the host during recording and analyzing them immediately. In this case, the trace memory of the TRACE32 trace tool is basically used as just a FIFO.

Extremely large data volumes are created during a long-term trace, so it is advisable to analyze the trace packages in parallel with the recording. Even if the trace packages are compressed before they are stored in a file, up to 5 GBytes are typically collected per hour. At the same time, you have to allow a lot of time for further analysis after recording is completed. For example, if you collect trace packages for a twohour program run in a file, you need several hours for a subsequent conventional analysis, even on a highpowered host.





Fig. 2: Long-term tracing needs a fast peer-to-peer interface to the host

If large data volumes are to be quickly recorded, transferred, and analyzed in long-term tracing, the following conditions must be met:

- Fast host
- Fast trace tool
- Compact data formats

Fast Host

In order to be able to analyze the trace packages on the host at program runtime, you need a fast dualcore computer. Here, one core receives the trace packages while the second core evaluates the packages in parallel. For the analysis, the program code is needed in addition to the trace packages. Since many ARM cores are not able to read the code from the target system memory at runtime, the code must be copied to the TRACE32 software before the start of the long-term trace.

Fast Trace Tool

As described above for classical tracing, the trace tool has to sample the trace packages loss-free at a fast trace port. High-speed transfer of the trace packages to the host is the new requirement for long-time tracing. For this purpose, the TRACE32 trace tool provides a GBit Ethernet interface. If the trace tool is connected peer-to-peer to the host, a transmission rate of more than 500 MBit/s can be achieved (see Figure 2).

The maximum transmission rate to the host is currently the bottleneck of long-term tracing. This means that long-term tracing will only work if the average data rate at the trace port does not exceed the maximum transmission rate to the host (see Figure 3).

High peak loads are not critical since they can be buffered by the trace memory.



Fig. 3: Long-term tracing works for this example if the average load at the trace port does not exceed 500 MBit/s

>>>

Software	Mobile Terminal	Floating Point Arithmetic	HDD Controller		
Trace information per instruction	0.8 Bit	2.2 Bit	4.3 Bit		
Core	Cortex-A	ARM11	ARM9		
Core frequency 500 MHz		300 MHz	450 MHz		
Trace port frequency DDR	166 MHz	75 MHz	150 MHz		
RTOS Linux		—	—		
Average data rate at trace port	340 MBit/s	406 MBit/s	798 MBit/s		

1. Optimal programming of ETMv3

LAUTERBACH

DEVELOPMENT TOO

You can directly influence the data rate at the trace port by programming the ETMv3 so that trace packages are only generated for analysisrelevant information. The data flow packages that represent a high load for the trace port are not usually needed for profiling and code coverage.

The other factors influencing the average data rate at the trace port unfortunately have to be

Compact Data Formats

Since the maximum transmission rate to the host is limited, it is important to keep the data volume as compact as possible. The data volume can be influenced in two ways:

1. Optimal programming of ETMv3

2. Compact buffering of trace packages

considered as unchangeable:

ARM core frequency: The higher the ARM core frequency, the more trace data per second.

Software on the target system: A software program that makes a large number of jumps and finds data/ instructions in the cache generates more trace packages per second than a software program that pro- »



Fig. 4: The Lauterbach trace tool for long-term tracing with the ETMv3



cesses many sequential instructions and often has to wait for the availability of data/instructions.

The table on page 4 shows a few examples of average data rate measurements at the trace port. It is surprising that the data rate is greatly influenced by the software running on the core. The core frequency and the core architecture do not play as significant of a role.

2. Compact buffering

The firmware of the TRACE32 trace tool has been enhanced so that, the optimal packing density of the packages in the trace memory is achieved with 8 pins for the output of the trace packages.

Summary

In the TRACE32 software, the configuration and analysis of the long-term trace runs under the name of *Real-Time Streaming* – RTS for short. A Lauterbach trace tool for long-term tracing of the parallel ETMv3 consists of the following TRACE32 products (see Figure 4):

PowerDebug II: Provides the GBit Ethernet interface to the host and transmits the trace packages.

Debug cable for the ARM core: Programs the ETMv3 over the JTAG interface.

PowerTrace II: Stores the trace packages, max. trace depth currently 4 GBytes.

Preprocessor AutoFocus II: Samples the trace packages at the parallel trace port and transfers them to the trace memory.

With long-term tracing, Lauterbach has taken an important step toward a trace technology that permits an almost unlimited analysis of the program run. It is fair to assume that both the processing power of the hosts and hard-disk capacity will increase constantly during the next few years, so we expect even more comprehensive analysis options will become available.

LONG-TERM TRACE ETMv3

Code Coverage-Analysis and Long-Term Tracing

B::RTS.COVerag	ge.ListModule										- • •
🌽 Setup 🔒	Goto	List 🕇 🕇	Add 🛛 🔀 Load	Save	🛛 Init						
addres	55	tree			coverage	execut	ted 0%	50%	100	taken	nottaken
P:0000A1F	F80000C1	E7 🗄 jdr	narker		parti	al 36.30	01%	_		78.	31. ^
P:0000C1F	C0000CD		ineg make d deri	ved thl	parti	al 05.54	75%		_	4	29.
P:0000C4E	E00000C6	27 🗉 3	tart_pass_huff_	decoder	parti	al 74.39	90%		-	8.	4.
P:0000C62	280000C7	5B 🗉 :	peg_fill_bit_bu	ffer	parti	al 54.54	45%			2.	3.
P:0000C75	5C0000C8	₩Ę 🗄	peg_hutt_decode		parti	al 62.29	95%			8.	1.
P:0000C8	500000CC	13 13	ipit buff decod	or	parti	al 58.50	00%			0.	15.
P.0000CCL	0000000	1	Thre_nurr_decou	ci		06 100.00				0.	
,											
B::Data.List P:	0xC850 /ISTAT	COVerage]									
N Sten	Over J	Nevt	Return 📌 Un	b Go	II Break	12 Mode	Find:	_	idbuff.c		
exec	notexec	COVERSIE	addr/line	code	label	mnemoni	c		comment		
37662.e6	0.	100.000%	580	couc	r = GET	_BITS(s)	:		connerre		
7532.e6	0.	100.000%	SR:0000CB44	E3A01001		mov	r1,#0x1		; MCU_da	ata,#1	
7532.e6	0.	100.000%	SR:0000CB48	E1A01411		mov	r1,r1,	sl r4	; MCU_da	ata,r1,ls	1 r4
7532.00	0.	100.000%	SR:0000CB4C	E04EE004		sub	r14,r14	,r4 10v1	; DITS_	ert, Dits	_lett,r4
7532.e6	0.	100.000%	SR:0000CB54	E0010E5B		and	r0.r1.r	11.asr r14	: cinfo.	MCU data	.r11.asr
33585.e6	11609.e6	100.000%	581		s = HUF	F_EXTEND	(r, s);				
7532.e6	0.	100.000%	SR:0000CB58	E59F11E8		ldr	r1,0xCD	48	; MCU_da	ata,0xCD4	8
/532.e6	0.	100.000%	SR:0000CB5C	E/911104		Idr	r1,[r1,	+r4,1s1 #(JX2]; MCL	J_data,[r	1,+r4, Is I
3662.06	3869 66	100.000%	SR:0000CB60	C59E11E0		Ideat	r1 0xC	40	, MCU_da	ata OvcD4	
3662.e6	3869.e6	100.000%	SR:0000CB68	c7911104		Idrat	r1.[r1.	+r4.lsl #()x21: MCL	J data.[r	1.+r4.lsl
3662.e6	3869.e6	100.000%	SR:0000CB6C	C0810000		addgt	r0,r1,r	0	; cinfo	MCU_data	,cinfo
					/* Outp	ut_coeff	icient i	ņ natural	(dezigza	agged) or	der.
					* Note	: the ext	tra entr	ies in jpe	eg_natura	i_order[j will sa
					*/	>- 0C15.	1202, W	nen coura	nappen	in che da	ica is cor
39832.e6	3610941.	100.000%	586		(*block)[jpeq_n	atural_c	rder[k]] =	(JCOEF)	s;	-
				•							▶

Fig. 5: Lists showing code coverage (overview and detailed)

One application for long-term tracing is checking whether all of the program code is processed during a system test.

For this code-coverage analysis of the trace data, the TRACE32 software provides a list of all modules/ functions and their code coverage. Additionally, a statistical summary of the execution of conditional instructions is displayed. Each function can be analyzed in detail: For linear code, you can see how often a command was run during the test (exec). For conditional instructions, you can also observe how often a command was skipped because its condition was not met (notexec).



LONG-TERM TRACE ETMv3

Profiling and Long-Term Tracing

For time-critical functions, maximum times are often defined and have to be checked during the system test. The long-term tracing feature makes this check easy and causes of timeouts to be quickly found.

Verification

First is the check whether time-critical functions

E B::RTS.ISTAT.ListModule											
🥔 Setup 🖪 Goto 💿 Init											
address	tree	coverage	count	time	clocks	ratio	cpi				
P:00009B580000A1F3	🗉 jdinput	64.302%	-	27.745s	4883.e6	0.058%	6.66	~			
P:0000A1F80000C1E7	🗉 jdmarker	36.301%	-	190.735s	33569.e6	0.402%	3.86				
P:0000C1FC0000CD43	⊟ jdhuff	65.927%	-	5.671ks	998.e9	11.977%	1.91	-			
P:0000C1FC0000C4DF	■ jpeg_make_d_derived_tb1	75.675%	7221882.	492.328s	86650.e6	1.039%	2.64				
P:0000C4E00000C627	start_pass_huff_decoder	74.390%	1203647.	9.763s	1718.e6	0.020%	5.37				
P:0000C6280000C75B	⊞ jpeg_fill_bit_buffer	54.545%	2024.e6	882.308s	155286.e6	1.863%	1.58	=			
P:0000C75C0000C84F	Image: Barbard Bar	62.295%	144.e6	87.325s	15369.e6	0.184%	2.96				
P:0000C8500000CCE7		58.503%	180.e6	4.195ks	738270.e6	8.859%	1.91				
P:0000CCE80000CD43	jinit_huff_decoder	100.000%	1203647.	4.308s	758.e6	0.009%	15.4				
P:0000CD580000DA9B	🗉 jdphuff	0.000%	-	0.000us	0.	0.000%	0.00				
P:0000DAB00000E13F	🗉 jdmainct	80.000%	-	131.902s	23214.e6	0.278%	3.17				
P:0000E1400000F09F	🗉 jdcoefct	19.410%	-	680.222s	119719.e6	1.436%	2.03				
P:0000F0B00000F4DF	🗉 jdpostct	11.567%	-	4.497s	791.e6	0.009%	18.3				
P:0000F4E00000F693	🗉 jddctmgr	68.807%	-	51.031s	8981.e6	0.107%	5.45				
P:0000F6A40000FB93	∃ jidctint	100.000%	-	15.325ks	2697.e9	32.368%	1.69	Ŧ			
	4										

Fig. 6: Analysis of time behavior of modules and functions

📕 (B::Data.List	0xf6a4 /ISTAT S	SAMPLES]		
Step	Nover	🕹 Next 🛛 🖋 F	teturn 🛛 🗶 Up 🗍	▶ Go 🔢 Break 🕅 Mode Find: jidctint.c
samples	time	ratio	addr/line s	source
				· · · · · · · · · · · · · · · · · · ·
23405684.	265.973s	0.561%	181	if (inptr[DCTSIZE*1] == 0 && inptr[DCTSIZE*2] == 0 &&
19724200.	224.138s	0.473%	182	inptr[DCTSIZE*3] == 0 && inptr[DCTSIZE*4] == 0 &&
20978599.	238.392s	0.503%	183	inptr[DCTSIZE*5] == 0 && inptr[DCTSIZE*6] == 0 &&
16204910.	184.146s	0.388%	184	inptr[DCTSIZE*7] == 0)
				/* AC terms all zero */
17911830.	203, 5435	0.429%	186	int dcval = DEQUANTIZE(inptr[DCTSIZE*0], guantptr[DCTSIZE*0]) <<
				wsptr[DCTSIZE*0] = dcva]:
4471747	50 815c	0 107%	189	wentr DCTSTZE*1 = dcval
4471747	50.8155	0.107%	100	worth DCTSTZE2) = doval
4471747	50 815c	0.107%	101	worth DCTSIZE*2] = dcval;
4471747	50.0155	0.107%	102	waptr DCISIZE'S - dcval,
44/1/4/.	50.8155	0.107%	192	wsptr[bcTSIZE^4] = dcval;
44/1/4/.	50.8155	0.10/%	193	wsptr[DCISizero] = dcval;
44/1/4/.	50.8155	0.10/%	194	wsptr[DCISIZE*6] = dcval;
				• • • • • • • • • • • • • • • • • • •

Fig. 7: Details of time behavior of program lines of a function

B::RTS.STAT.TREE									x	
🎾 Setup] 🛅 Groups] 🚼 Config] 🗭 Goto] 🛒 List all 📜 Nesting 🗮 Func (hat) 💿 Init										
funcs: 144. total: 47.346ks										
tree	total	avr	max	internal	external	intern%	1%	2%		
ecompress_onepass	21.516ks	1.788ms	2.125ms	671.197s	20.845ks	1.417%	—		~	
— jzero_tar	237.176s	1.312us	22.727us	237.176s		0.500%	+			
ecode_mcu	5.281ks	29.250us	147.727us	4.195ks	1.086ks	8.859%			-	
□ jpeg_till_bit_butter	996.030s	0.491us	125.000us	878.971s	117.059s	1.856%		-		
□ □ □ fill_input_buffer	117.059s	97.251us	113.636us	3.242s	113.817s	0.006%	+			
JpegMemorySourceCall.	113.81/s	94.560us	113.636us	113.81/s		0.240%	+			
□ □]peg_huft_decode	90.662s	0.62/us	11.364us	87.3255	3.33/s	0.184%	+			
	3.33/5	0.552us	11.364us	3.33/s	-	0.00/%	+		_	
	15.325ks	14.959us	56.818us	15.325ks	-	32.368%				
- start_1MCU_row	1.529s	Statistic		1.529s	-	0.003%	+			
	184.658ms	I ist first		184.658ms	-	<0.001%	+		-	
	m	and the second						,	•	
	List last							_		
			nalysis 😽							
			Analysis							
	here									

Fig. 8: If necessary, details on the longest function run can quickly be displayed

- exceed their maximum time. For this purpose, the ETMv3 should be programmed so that it generates only trace packages for the program flow and the context ID. There are two reasons for this:
- 1. In this way, the data rate at the trace port is kept as low as possible.
- 2. FIFO overflows preventing an exact analysis of function nesting are prohibited.

After the long-term trace is started, the TRACE32 software analyzes the time behavior of the functions. The following are analyzed: the runtime and number of clock cycles during the complete test run, the function's share of the total runtime, and the average "clocks per instruction" (see Figure 6).

A detailed analysis of the individual function also shows the time behavior of the program lines (see Figure 7).

If the analysis intermittently exceeds the defined maximum time, the cause must of course be found.

Troubleshooting

The long-term trace can be configured so that the trace packages are saved in a file at program runtime. With a data volume of 5 GBytes/hour, about four days of the program runtime can be recorded on an average hard disk. Using fast, sophisticated search functions, the TRACE32 software can then search the trace recording for the excessive function run and show it in a detailed analysis (see Figure 8).