WHITE PAPER

TRACE32 log method for analysing accesses to an eMMC device

Authors: Marco Ferrario, Maurizio Menegotto, Lauterbach Italy
Lauterbach S.r.l., Regus EasyPoint 1st floor, Via Caldera 21, I-20153 Milano

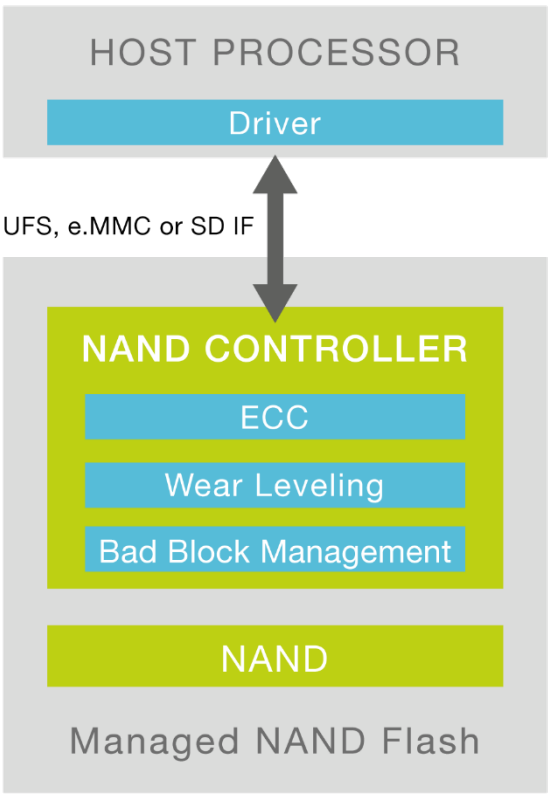
First edition: July 8th 2021, Revision: November 11th, 2021

Summary

Introduction

The widespread use of eMMC storage in many of today's applications raises the issue of premature device degradation or wear-out resulting from intensive memory usage. To study this possible problem, it is necessary to record the accesses to an eMMC device in order to obtain the required information that can be subsequently analysed to improve stability and reliability over the device's expected lifespan. From this kind of analysis, it's possible to understand how your software application actually accesses a filesystem mounted on an eMMC and if this can cause premature aging of the NAND-based memory device.

SD cards, eMMC and UFS memory chips are managed-NAND block devices, consisting of a NAND controller, an internal firmware performing ECC operations, wear-levelling and bad-block management of the raw NAND memory.



The specific architecture of a managed-NAND device can be extremely sensitive to certain read and write access sequences performed by the host processor under the direction of the application software, especially if these are frequently iterated.

A classic recording method (log) of these accesses requires the implementation of additional code that captures information and saves it securely. The information can be saved on another permanent storage device, for example an external USB drive. This software method is intrusive and in addition to the overhead of monitoring the eMMC access, additional overhead is added in order to save the data.

This study proposes a different method of capturing and saving such information through the use of a hardware-based trace tool. This can be done with minimal intrusion on the software and, in some cases, almost zero. This tool captures the program and data trace transmitted by the cores of an SoC through a dedicated trace port, and records it to its own dedicated memory.

Arm CoreSight™

Many embedded microprocessors and microcontrollers are able to trace information related to the program execution flow. This allows the sequence of instructions executed by the program to be reconstructed and examined in great detail. In some configurations it is also possible to record the data related to the read and/or write cycles performed by the program.

CoreSight™ is the name of the on-chip debug and trace technology provided by Arm®. CoreSight™ is not intended as a default logic block but, like a construction kit, it provides many different components. This allows the SoC designer to define the debug and trace resources that they want to provide. Program flow (and sometimes data flow) information is output through a resource called ETM (Embedded Trace Macrocell). The ETM trace information flow can be stored internally (on-chip trace) or can be exported outside of the SoC (off-chip trace). Arm® provides several ways for exporting a trace flow: through a parallel trace port (TPIU, Trace Port Interface Unit), or serial trace port (HSSTP, High-Speed Serial Trace Port) or through a PCIe interface.

When data trace is not available, Arm® provides the Context ID register. This is often used by an Operating System (OS) to indicate that a task switch has occurred. This is done by code in the OS kernel writing the task identifier to this register. In a multicore Arm®/Cortex® SoC, each core implements this register.

Lauterbach TRACE32 development tools

Lauterbach's TRACE32 development tools enable hardware-based debug and trace of a wide range of embedded microprocessors and microcontrollers and support debug technologies such as JTAG or SWD, as well as trace technologies such as NEXUS or ETM.

The TRACE32 tools support all Arm® CoreSight™ configurations. A TRACE32 development tool for debug and trace is typically comprised of these units:

- a universal PowerDebug module connected to the host computer via USB3 or Ethernet;
- a debugger (debug cable) for the specific architecture of the microprocessor or microcontroller under debug;
- for the off-chip trace, a universal PowerTrace II or PowerTrace III module providing 4GB or 8GB memory, complemented by a parallel or serial pre-processor to access the trace data;
- or a dedicated PowerTrace Serial module for serial or PCIe trace data.

TRACE32-based eMMC access log solution

In all operating systems or device drivers that manage an eMMC memory device, some functions are provided for device access which incorporate the eMMC JEDEC standard commands. Long-term monitoring of the execution of these commands and their parameters is the best way to collect the data necessary for the access analysis. After accessing the eMMC device, a function or a code point is usually available where the eMMC command is completed. Monitoring this code point allows the detection of additional information, such as the execution time of the command.

The TRACE32 trace tool can sample the code points where eMMC accesses start and finish. By adding a tiny amount of instrumentation to your source code, you can also trace device access data. In cases where data trace is not available, the instrumentation code writes the access data to the ContextID register, allowing both types of system to be adapted to use this technique.

The following data is traced in the TRACE32-based log solution:

- at the beginning of eMMC access: eMMC device id, command executed and related flags, access address, number of accessed memory blocks and their size;
- at the end of the eMMC access: eMMC device id, command executed, result code and other return codes;
- access duration.

A possible example of access monitoring is shown below, as it appears in the trace views available in TRACE32:

```
2| ptrace  \\vmlinux\core_core\mmc_start_request  24.228827980s
2| info                                           24.228828005s     31636D6D
2| info                                           24.228828030s     00000019
2| info                                           24.228828055s     01620910
2| info                                           24.228828080s     000000B5
2| info                                           24.228828105s     00000200
2| info                                           24.228828130s     00000010

0| ptrace  \\vmlinux\core_core\mmc_request_done   24.231239610s
0| info                                           24.231241385s     31636D6D
0| info                                           24.231241410s     00000019
0| info                                           24.231241435s     00000000
0| info                                           24.231308085s     00000900
0| info                                           24.231308210s     00000000
```

This is, typically, a few trace records for each eMMC access. Stress tests have verified that logging an eMMC access (functions **`mmc_start_request()`** and **`mmc_request_done()`** with related data) requires about 416 trace records in the PowerTrace memory and these accesses occur on average every 4 mSec.

This corresponds to approximately 1GB/416 = 2.5 million eMMC logs, or approximately 10,000 seconds (2h45min) for each gigabyte of trace storage. The PowerTrace family provides either 10 million eMMC logs (11h) for a 4GB PowerTrace or 20 million (22h) for an 8GB module.
By extending the trace duration with trace streaming, the limit becomes the size of the computer hard-disk/SSD or the TRACE32 limit which is 1 Tera-frame, i.e., 2.5 billion eMMC logs (over 100 days!).

The trace data can be filtered and saved on disk, and then converted into a more suitable format for analysis using a TRACE32 script (PRACTICE script), Python script or an external conversion program.

For example, the trace shown above can be converted into the format shown below, which is more suitable for importing into specific eMMC analysis tools:

```
24.228827980 mmc_start_req_cmd: host=mmc1 CMD25 arg=01620910 flags=000000B5
blksz=00000200 blks=00000010
24.231239610 mmc_request_done: host=mmc1 CMD25 err=00000000 resp1=00000900
resp2=00000000
```

These tools perform a complete analysis of the eMMC device application accesses, in terms of addresses accessed, frequency and access methods.

The end-goal is calculating the Write Amplification (WA) seen by the eMMC (or by any other managed-NAND block device). Write Amplification (WA) is defined as the ratio of NAND physical writes and the host induced writes (WA = NAND writes / Host Writes).

When the host writes logical sectors of the eMMC, the internal MMC controller erases and reprograms physical pages of the NAND device. This could cause a management overhead. Large sequential writes aligned to physical page boundaries typically result in minimal overhead and optimal NAND write activity (WA=~1). Small-chunks of random writes could result in a higher overhead (WA>>1).

This becomes important when considering the life of the raw-NAND memory inside the eMMC, which has a finite number of Program/Erase cycles. See the example below:

Item	Value
Device Capacity	8GB
Write Endurance	2K Program/Erase cycles
Data Written Per Day to Device	2GB
Expected Life w/ WA=1 =(8×2000)/(2*1)	8,000 days
Expected Life w/ WA=5 =(8×2000)/(2*5)	1,600 days

To estimate the WA for any particular eMMC device, and hence its expected lifetime on your application, you can capture the log file of the activity.

Once a log is obtained, it's recommended to contact your eMMC vendor to get more information about the log analysis tools required for analysing the specific eMMC product.

Implementation example for Linux OS

Below is an example of how the TRACE32-based log method can be applied to a Linux system. The solution is based on light instrumentation of the **mmc_start_request()** and **mmc_request_done()** functions defined in the Linux "**drivers/mmc/core/core.c**" source code file. Relevant eMMC device accesses are captured through the instrumentation code and they are written to a static data structure making them immediately traceable if data trace is available in the SoC. If data trace is not possible, the instrumentation code writes the data to the Arm®/Cortex® Context ID register.

The solution was successfully tested on the DAVE Embedded Systems "MITO 8M Evaluation Kit" (see https://www.dave.eu/en/solutions/system-on-modules/mito-8m). The kit consists of three boards: SoM, SBCX carrier board, adapter board. This setup provides off-chip trace via a parallel trace port or a PCIe interface. The SoM is equipped with the NXP i.MX8M processor based on the Quad Core Arm® Cortex-A53 CPU. The Linux kernel version used is 4.14.98.

The instrumentation code is provided in Appendix 1. The zero initialization of the **T32_mmc** structure is guaranteed by Linux, since this variable is allocated in the bss section. The instrumentation is normally disabled but can be enabled by writing the value "1" in the **enable** field of the **T32_mmc** structure. The identifier of the eMMC device to be traced must be written in the **dev** field. Both of these operations can be performed from a TRACE32 script with the following commands:

```
Var.set T32_mmc.enable = 1
Var.set T32_mmc.dev = 0x30636D6D   // e.g.: "mmc0" in reverse ASCII order
```

The infoBit field can be written as follows:

```
Var.set T32_mmc.infoBit = 0x80000000
```

This allows the user and the tools to distinguish between data written in the Context ID register by the instrumentation code from those written by Linux for task switches. In this case, the range of values must also be reserved so that they are not interpreted as task switch identifiers. The command to do this is shown below:

```
ETM.ReserveContextID 0x80000000--0xffffffff
```

It's important to note that the Linux kernel must be compiled for debug (see the Training Linux Debugging manual at https://www.lauterbach.com/manual.html). The TRACE32 debugger also offers extensions for many different operating systems, known as an "OS awareness". These add OS specific features to the TRACE32 debugger such as the display of OS resources (tasks, queues, semaphores, ...) or support for MMU management in the OS. In TRACE32, the ability to trace tasks and executed code is based on task switch information in the trace flow. The command **ETM.ReserveContextID** allows simultaneous use of the Linux OS awareness support and the instrumentation for eMMC access analysis.

To reduce the amount of trace information generated by the target and to allow long-term trace via streaming, filters can be applied to isolate just the instrumentation code and its writes to the Context ID register. For example:

```
Break.REset
Break.Set  mmc_request_done     /Program /TraceON
Break.Set  mmc_request_done\94  /Program /TraceOFF
Break.Set  mmc_start_request    /Program /TraceON
Break.Set  mmc_start_request\38 /Program /TraceOFF
```

where the filters marked as `/TraceOFF` are mapped to program addresses immediately after the instrumentation.

To ensure the task switch data generated by the OS is included in the filtered trace flow, add an additional filter to the **`__switch_to()`** function (**`arch/arm64/kernel/process.c`**) where it calls the **`static inline contextidr_thread_switch()`** function:

```
Break.Set     __switch_to+0x74 /Program /TraceON
Break.Set     __switch_to+0x80 /Program /TraceOFF
```

The trace flow recorded by TRACE32 can be arranged into a view suitable for exporting by post-processing with the command:

```
Trace.FindAll , Address address.offset(mmc_start_request) OR Address
address.offset(mmc_request_done) OR Cycle info OR Cycle task /List run cycle
symbol %TimeFixed TIme.Zero data
```

NOTE: '`OR Cycle task`' is optional.

Comparison with the software method ftrace

In Linux, eMMC access log solutions based on purely software methods are already available. The ftrace framework provides this capability, as well as being able to log many other events. The term "ftrace" stands for "function tracer" and basically allows you to examine and record the execution flow of kernel functions. The dynamic tracing mode of ftrace is implemented through dynamic probes injected into the code, which allow runtime definition of the code to be traced. When tracing is enabled, all the collected data is stored by ftrace in a circular memory buffer. In the framework there is a virtual filesystem called **`tracefs`** (usually mounted in **`/sys/kernel/tracing`**) which is used to configure ftrace and collect the trace data. All management is done with simple operations on the files in this directory.

Comparative tests performed on the DAVE Embedded Systems "MITO 8M Evaluation Kit" target showed that the ftrace impact compared to the TRACE32-based log solution is considerably higher in several respects. This is understandable, considering that ftrace is a general-purpose trace framework designed to trace many possible events, while the instrumentation required for the TRACE32 log method is specific and limited to the pertinent functions. Moreover, ftrace

requires some buffering (ring buffer) and saving data to a permanent memory, while the solution based on TRACE32 uses off-chip trace to save the data externally in real time. The following tables show a comparison between ftrace and the TRACE32 solution.

Table 1: instrumentation size

	vmlinux code size	vmlinux data	vmlinux source files	instrumentation code size (*)	instrumentation data size (*)
Clean	12,79MB	10,78MB	4640		
TRACE32	12,79MB **(+0%)**	10,78MB **(+0%)**	**+0** (41 source code lines in mmc driver)	+372 byte	+64 byte
ftrace	14,78MB **(+15,6%)**	11,77MB **(+9%)**	+836 **(+18%)**	+1,99MB	+0,99MB + ??MB ring buffer (**)

(*) ftrace instrumentation applies to the whole Linux kernel. TRACE32 instrumentation applies to the functions **`mmc_start_request()`** and **`mmc_request_done()`** only.

(**) the actual size of the ftrace ring buffer can be configured during runtime but is typically between 10—100MB.

In the ftrace-based solution, an increase in kernel size of approximately 15% (code) and 9% (data) is observed compared to the kernel without ftrace. During the execution of ftrace it's also necessary to reserve additional memory for the ring buffer. The number of source files used in building the kernel increases by 18% when the ftrace framework is included. The weight of the instrumentation required by TRACE32, on the other hand, is practically negligible both in terms of code and data.

Table 2: instrumentation time intrusion

Average duration at measuring points (*)	No ftrace No TRACE32 instr.	No ftrace With TRACE32 instr.	With ftrace No TRACE32 instr.
mmc_start_request	6.950us	8.108us (+1.158us)	36.875us
mmc_request_done	0.770us	1.364us (+0.594us)	63.031us

(*) measuring points are the part of functions where the instrumentation is added.

The functions average duration analysis of eMMC accesses highlights the greater weight required by ftrace. The tests were performed under the following conditions.

Test scenario: R/W access to mmc0 with command:

```
stressapptest -s 20 -f /mnt/mmc0/file1 -f /mnt/mmc0/file2   ; duration = 20s
```

Results in /mnt/mmc0 (16MB)

```
-rw-r--r-- 1 root root 8388608 Dec  3 16:30 file1
-rw-r--r-- 1 root root 8388608 Dec  3 16:30 file2
```

Setup for ftrace

```
echo 1 > /sys/kernel/debug/tracing/tracing_on
echo 1 > /sys/kernel/debug/tracing/events/mmc/enable
echo 20000 > /sys/kernel/debug/tracing/buffer_size_kb     ; 20MB buffer size
echo > /sys/kernel/debug/tracing/trace
cat /sys/kernel/debug/tracing/trace_pipe > /home/root/prove/ftrace.txt
```

Please note that the ftrace pipe is saved to a file on a different memory device (mmc1).

Additional, more detailed charts are provided in Appendix 2, which show that using ftrace also involves a greater dispersion of the runtime durations compared to both the kernel without ftrace and the kernel instrumented only with the code for TRACE32. In particular, the functions **`mmc_start_request()`** and **`mmc_request_done()`** have a few uS constant execution time without ftrace, and show a very variable execution time with ftrace, with a maximum time up to 279uS and 285uS respectively.

Conclusion

The HW method based on TRACE32 provides the same log data as recorded by ftrace but with minimal changes to the kernel (a few lines in a file) and a tiny time penalty. It also does not use any additional memory (ram and file system) and allows for extremely long measurement times.
The following Table 3 summarizes the advantages and disadvantages of the two considered solutions: TRACE32 vs ftrace.

Table 3: PROS and CONS

	PROS	CONS
TRACE32	- light kernel instrumentation; - no additional memory required; - long-term analysis (few hours up to over 100 days); - can be ported to other OS / eMMC device drivers.	- HW-based solution: requires a debug and trace tool + offchip-trace capable processor and target
ftrace	- SW-based solution	- available for Linux kernel only; - heavy kernel instrumentation; - time intrusion in eMMC operation; - kernel program and data size increase; - 10—100 MB of ram required for ring buffer; - additional storage device to save the ring buffer; - for each eMMC operation ftrace saves roughly 876 byte of log information.

Please contact your eMMC vendor to obtain more information on how TRACE32 logs can be used to calculate your application lifespan. This is a very important milestone to improve the storage performance stability of your platform and for making sure the expected reliability requirements are met.

References

Design Considerations for Embedded Products, Western Digital Corporation, 2018
https://link.westerndigital.com/content/dam/customer-portal/en_us/external/public/cps/p/White_Paper_Design_Considerations_v1.0.pdf

Automotive Workload Analysis, Western Digital Corporartion, September 2021
https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/white-paper/white-paper-automotive-workload-analysis.pdf

Appendix 1: source code example

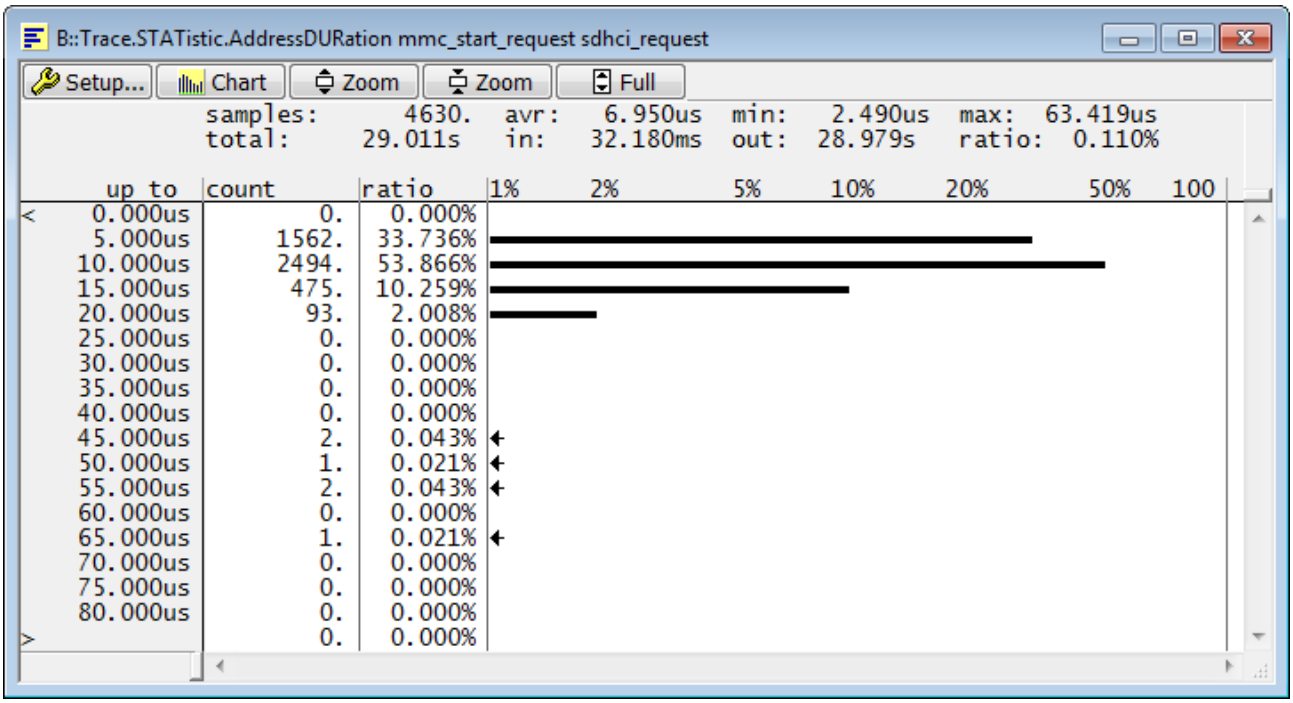
```
static struct T32_mmc_struct {
  unsigned int   enable;
  unsigned int   infoBit;
  unsigned int   dev;
  unsigned int * pHost;
  unsigned int   cmd;
  unsigned int   arg;
  unsigned int   flags;
  unsigned int   blksz;
  unsigned int   blocks;
  unsigned int   err;
  unsigned int   resp0;
  unsigned int   resp1;
  unsigned int   resp2;
  unsigned int   resp3;
} T32_mmc;

int mmc_start_request(struct mmc_host *host, struct mmc_request *mrq)
{
  int err;
  mmc_retune_hold(host);
  if (mmc_card_removed(host->card))
     return -ENOMEDIUM;
  mmc_mrq_pr_debug(host, mrq, false);
  WARN_ON(!host->claimed);
  if (T32_mmc.enable) {
     T32_mmc.pHost = (unsigned int *)mmc_hostname(host);
     if ((*T32_mmc.pHost)==T32_mmc.dev) {
        if (mrq->cmd) {
           write_sysreg((*T32_mmc.pHost)|T32_mmc.infoBit,
                   contextidr_el1);
           isb();
           T32_mmc.cmd = (mrq->cmd->opcode)|T32_mmc.infoBit;
           write_sysreg(T32_mmc.cmd, contextidr_el1);
           isb();
           T32_mmc.arg = (mrq->cmd->arg)|T32_mmc.infoBit;
           write_sysreg(T32_mmc.arg, contextidr_el1);
           isb();
           T32_mmc.flags = (mrq->cmd->flags)|T32_mmc.infoBit;
           write_sysreg(T32_mmc.flags, contextidr_el1);
           isb();
        }

        if (mrq->data) {
           T32_mmc.blksz = (mrq->data->blksz)|T32_mmc.infoBit;
           write_sysreg(T32_mmc.blksz, contextidr_el1);
           isb();
           T32_mmc.blocks = (mrq->data->blocks)|T32_mmc.infoBit;
           write_sysreg(T32_mmc.blocks, contextidr_el1);
           isb();
        }
     }
  }
  err = mmc_mrq_prep(host, mrq);
  if (err)
     return err;
...
```

```
void mmc_request_done(struct mmc_host *host, struct mmc_request *mrq)
{
  struct mmc_command *cmd = mrq->cmd;
  int err = cmd->error;
...

...

  if (!err || !cmd->retries || mmc_card_removed(host->card)) {
     mmc_should_fail_request(host, mrq);

     if (!host->ongoing_mrq)
        led_trigger_event(host->led, LED_OFF);

     if (mrq->sbc) {
        pr_debug("%s: req done <CMD%u>: %d: %08x %08x %08x %08x\n",
           mmc_hostname(host), mrq->sbc->opcode,
           mrq->sbc->error,
           mrq->sbc->resp[0], mrq->sbc->resp[1],
           mrq->sbc->resp[2], mrq->sbc->resp[3]);
     }

     pr_debug("%s: req done (CMD%u): %d: %08x %08x %08x %08x\n",
        mmc_hostname(host), cmd->opcode, err,
        cmd->resp[0], cmd->resp[1],
        cmd->resp[2], cmd->resp[3]);

     if (mrq->data) {
        pr_debug("%s:     %d bytes transferred: %d\n",
           mmc_hostname(host),
           mrq->data->bytes_xfered, mrq->data->error);
     }

     if (mrq->stop) {
        pr_debug("%s:     (CMD%u): %d: %08x %08x %08x %08x\n",
           mmc_hostname(host), mrq->stop->opcode,
           mrq->stop->error,
           mrq->stop->resp[0], mrq->stop->resp[1],
           mrq->stop->resp[2], mrq->stop->resp[3]);
     }

     if (T32_mmc.enable) {
        T32_mmc.pHost = (unsigned int *)mmc_hostname(host);
        if ((*T32_mmc.pHost)==T32_mmc.dev) {
           write_sysreg((*T32_mmc.pHost)|T32_mmc.infoBit,
                   contextidr_el1);
           isb();
           T32_mmc.cmd = (cmd->opcode)|T32_mmc.infoBit;
           write_sysreg(T32_mmc.cmd, contextidr_el1);
           isb();
           T32_mmc.err = (err)|T32_mmc.infoBit;
           write_sysreg(T32_mmc.err, contextidr_el1);
           isb();
           T32_mmc.resp0 = (cmd->resp[0])|T32_mmc.infoBit;
           write_sysreg(T32_mmc.resp0, contextidr_el1);
           isb();
        }
     }
  }
  /*
   * Request starter must handle retries - see
   * mmc_wait_for_req_done().
   */
  if (mrq->done)
     mrq->done(mrq);
}
```

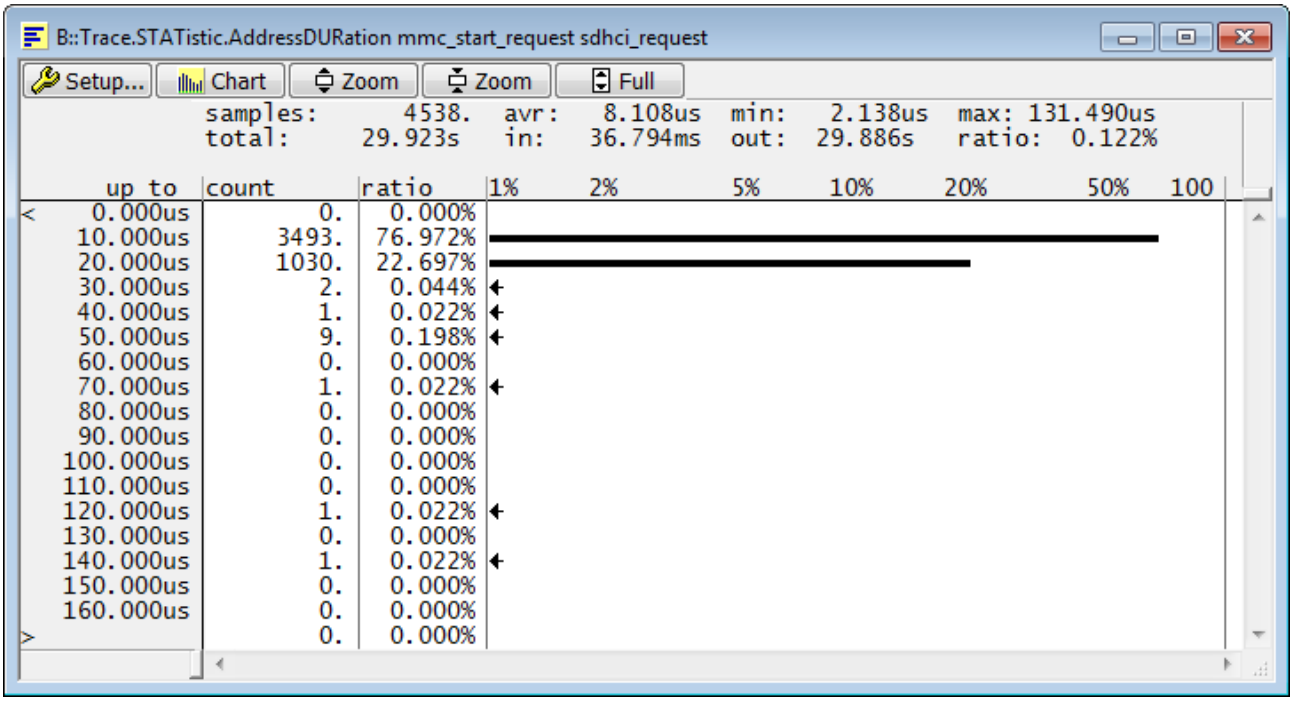
Appendix 2: time details

1) time duration analysis: mmc_start_request

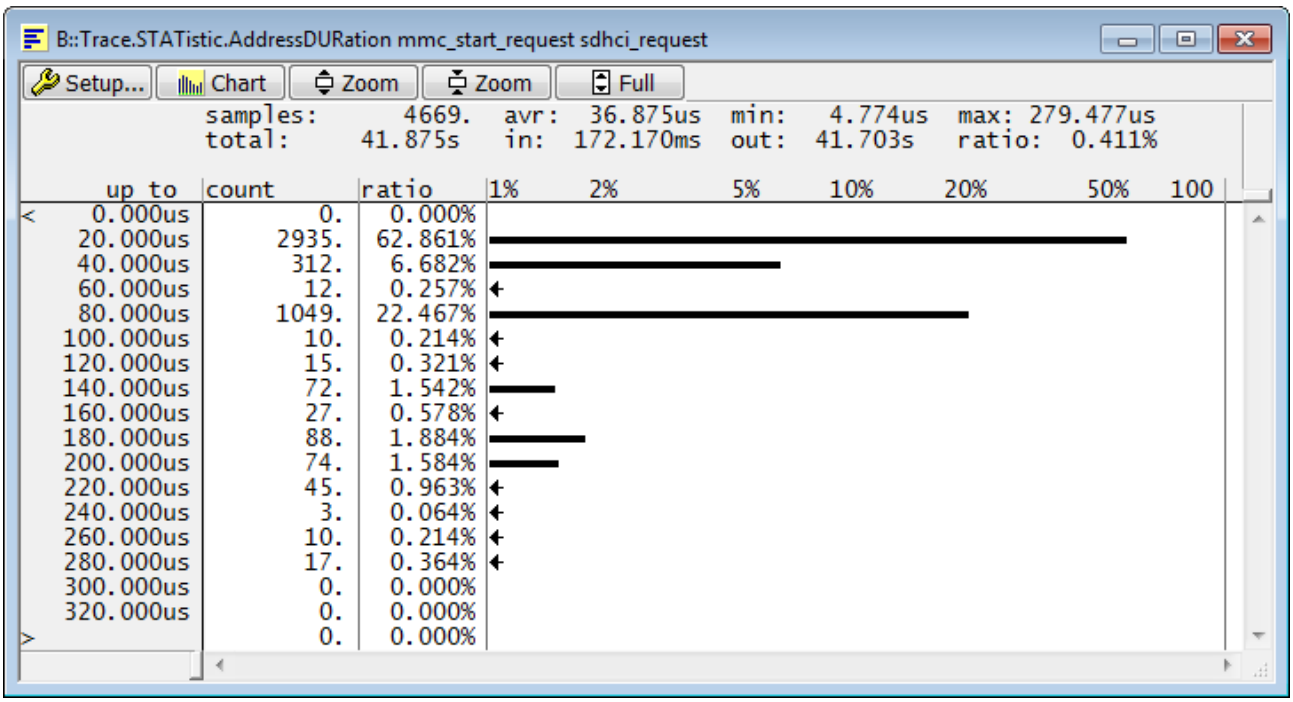
1.1) No ftrace, no TRACE32 instrumentation



1.2) No ftrace, with TRACE32 instrumentation

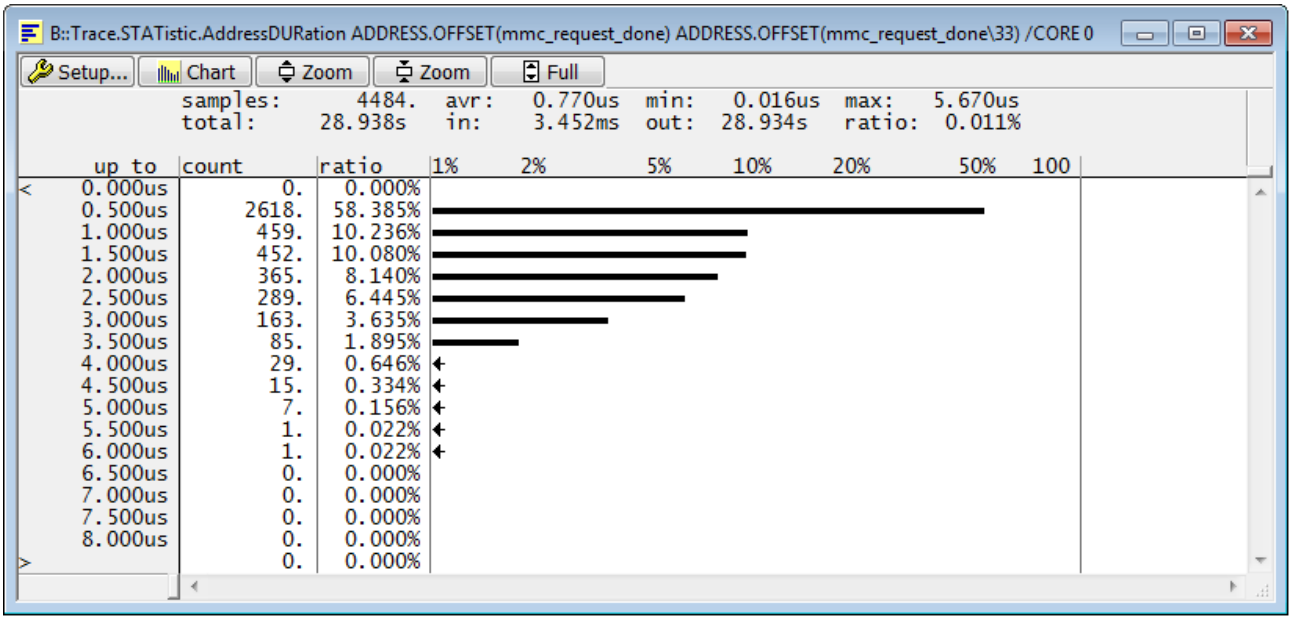


1.3) With ftrace, no TRACE32 instrumentation

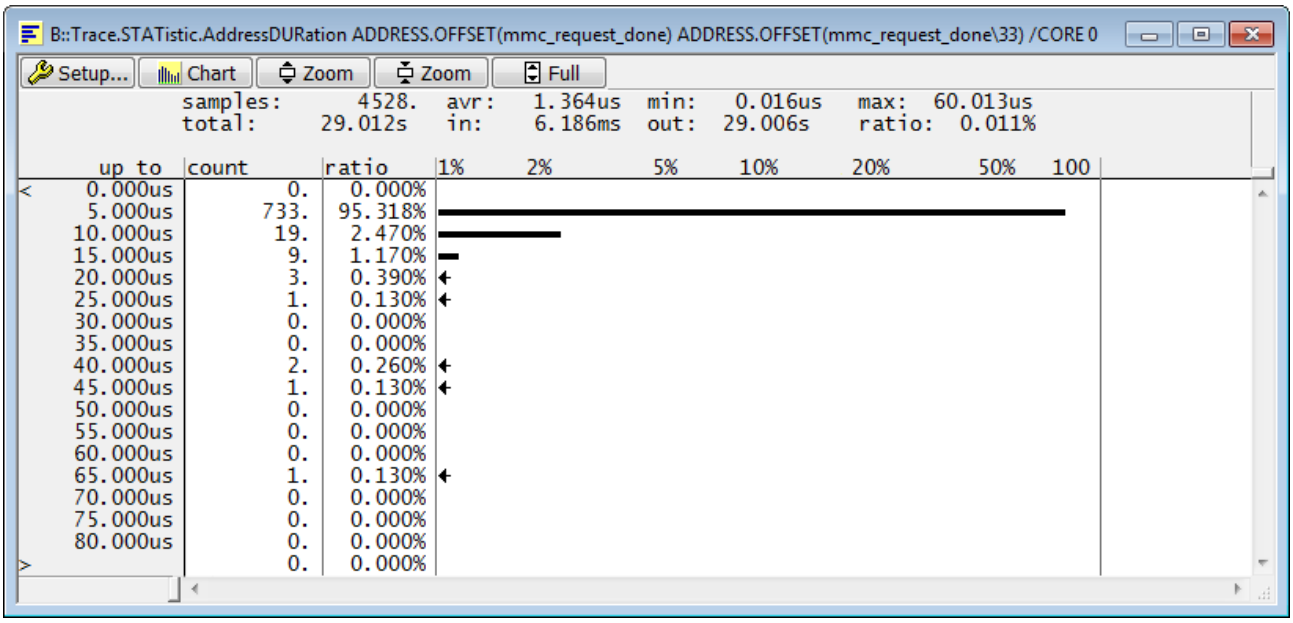


2) time duration analysis: mmc_request_done

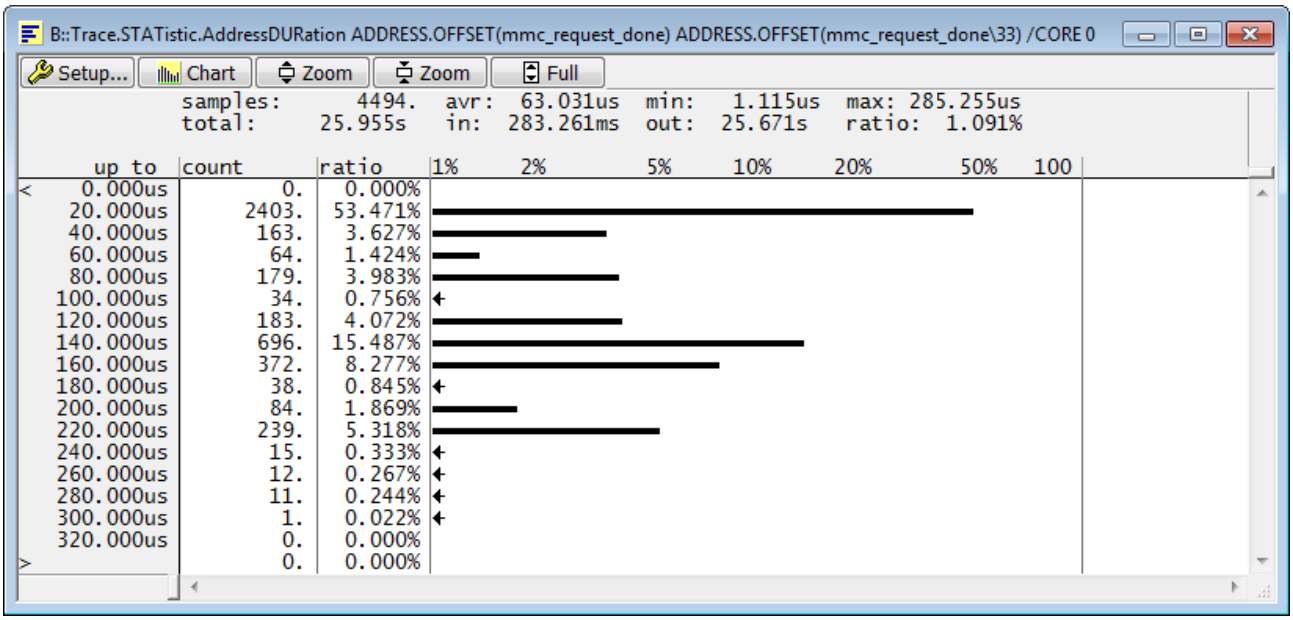
2.1) No ftrace, no TRACE32 instrumentation



2.2) No ftrace, with TRACE32 instrumentation



2.3) With ftrace, no TRACE32 instrumentation



Appendix 3: TRACE32 tools configuration for Arm Cortex-A/R architectures

This appendix describes the TRACE32 Arm trace tools needed to perform the eMMC access analysis.
Chips based on Armv7/Armv8/Armv9 Cortex A/R cores typically include Arm Coresight debug & trace logic. In order to increase the trace recording time, the trace flow produced by the Arm ETM/PTM trace logic is sent off-chip via a dedicated trace port.
Typical trace ports are TPIU (parallel) and HSSTP (serial). Additionally, on some chips the trace flow can be transferred outside via PCIe bus, or even stored to an external DDR memory. This last method can be quite intrusive and limits the duration of the measurement because it depends on how much of the target RAM is free (not used by the application) and therefore available for trace storage.
Our recommendation is to use a trace port to transmit the trace off-chip in real time, without causing intrusion and without limiting the duration of the measurement.

Lauterbach offers various PowerTrace models, with different amounts of trace storage and different trace probes. Here an overview:
https://www.lauterbach.com/powertrace_overview.html

Chips based on Arm cores can have different trace ports, in particular:

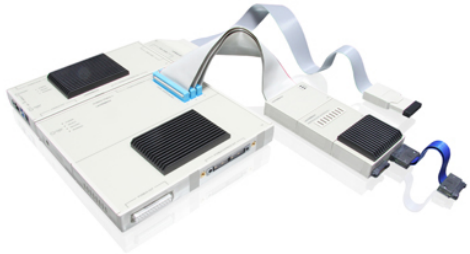
For chips with "parallel" trace port (TPIU), a PowerTrace-II or, better yet, PowerTrace-III system is recommended, plus an Arm AutoFocus II pre-processor with Mictor38 connector.

PowerTrace-III system for Armv7 Cortex-A/R with “parallel” trace port, eg. i.MX6 Quad

LA-3505 PowerDebug PRO Ethernet
LA-2520 PowerTrace III 8 GigaByte
LA-7843 Debugger for Cortex-A/R (ARMv7 32-bit)
LA-7992 Preproc. for ARM-ETM/AUTOFOCUS II 600 Flex
https://www.lauterbach.com/pro/pro_imx6quad_alt03.php

Mictor38 connector p/n and pinout:
https://www.lauterbach.com/adetmmictor.html

PowerTrace-III system for Armv8 Cortex-A/R with “parallel” trace port, eg. some i.MX8M



LA-3505 PowerDebug PRO Ethernet
LA-2520 PowerTrace III 8 GigaByte
LA-3743 Debugger for Cortex-A/R (Armv8 and Armv9)
LA-7992 Preproc. for ARM-ETM/AUTOFOCUS II 600 Flex
https://www.lauterbach.com/pro/pro_cortex-a53_alt03.php

Mictor38 connector p/n and pinout:
https://www.lauterbach.com/adetmmictor.html

For chips with “serial” trace port (HSSTP), either a PowerTrace-II or PowerTrace-III with Serial Preprocessor, or a PowerTrace Serial can be used.

PowerTrace-III system for Armv8 Cortex-A/R with “serial” trace port, eg. i.MX8 Dual x Plus

LA-3505 PowerDebug PRO Ethernet
LA-7694 PowerTrace II 4 GigaByte
LA-3743 Debugger for Cortex-A/R (Armv8 and Armv9)
LA-7988 Preproc. for ARM-ETM/HSSTP HF-Flex
https://www.lauterbach.com/pro/pro_imx8dualxplus_alt03.php

Samtec40 connector p/n and pinout:
https://www.lauterbach.com/adetmhsstp.html

PowerTrace Serial system for Armv8 Cortex-A/R with “serial” trace port, eg. i.MX8 Dual x Plus

LA-3505 PowerDebug PRO Ethernet
LA-3743 Debugger for Cortex-A/R (Armv8 and Armv9)
LA-3520 PowerTrace Serial 4 GigaByte for ARM-ETM
LA-3521 Accessories for PTSerial for ARM-ETM 1-6Lanes
https://www.lauterbach.com/pro/pro_imx8dualxplus_alt04.php

Samtec40 connector p/n and pinout:
https://www.lauterbach.com/adetmhsstp.html

Some chips which provide sufficient PCIe lanes have the ability to transmit the trace out through the PCIe bus. In this case a PowerTrace Serial system with a particular adapter for PCIe is required. This method can be used as an alternative to the classical “parallel” or “serial” trace ports from Arm.

PowerTrace Serial system for Armv8 Cortex-A/R with PCIe trace, eg. i.MX8M Mini Quad

LA-3505 PowerDebug PRO Ethernet
LA-3743 Debugger for Cortex-A/R (Armv8 and Armv9)
LA-3520 PowerTrace Serial 4 GigaByte for ARM-ETM
LA-3550X License for PCI Express
LA-3522 Accessories for PTSerial for ARM-ETM 7-8Lanes
https://www.lauterbach.com/pro/pro_imx8mminiquad_alt03.php

PCIe connectors and adapters:
https://www.lauterbach.com/adpts_pcie.html

For further technical and commercial information, please contact the Lauterbach office for your area:
https://www.lauterbach.com/worldwide_rep.html
https://www.lauterbach.com