LAUTERBACH
DEVELOPMENT TOOLS

Application Note for eMMC Analysis

MANUAL

Application Note for eMMC Analysis

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents .....

Trace Application Notes .....

Trace Analysis .....

Application Note for eMMC Analysis 1

History 3

Introduction 4

TRACE32-based eMMC Access Log Solution 5

Implementation Example for Linux OS 8

Comparison with the Software Method ftrace 11

Conclusion 13

References 14

Appendix A: Source Code Example 15

Appendix B: Time Details 18

Application Note for eMMC Analysis

Version 06-Jun-2024

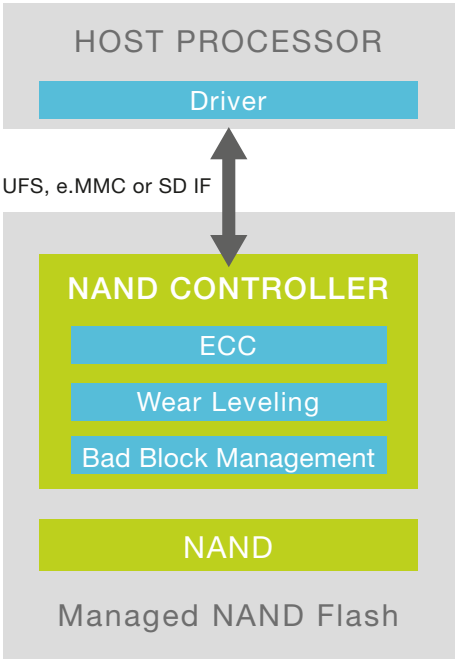
History

17-Jan-2022 New manual.

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 analyzed to improve stability and reliability over the device's expected lifespan. From this kind of analysis, it's necessary 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 document proposes a different method of capturing and saving such information through the use of a TRACE32 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 a SoC through a dedicated trace port, and records it to its own dedicated memory.

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 code points where eMMC accesses start and finish can be provided by a program trace.

In order to provide the eMMC details, a tiny amount of instrumentation to the source code is required.

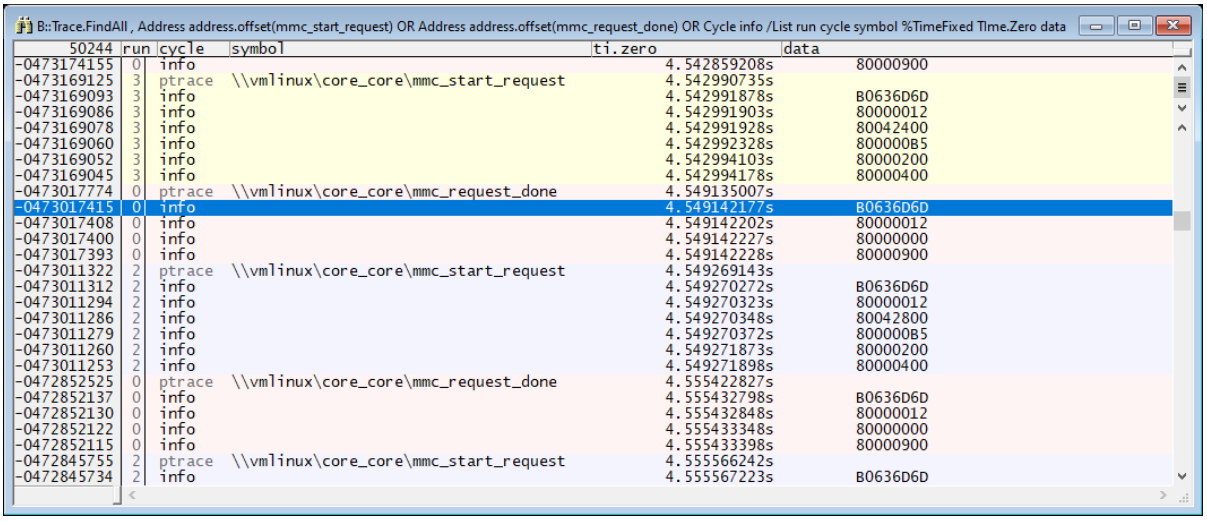
- If data trace is available, the eMMC details can be written to a static data structure.
- If no data trace is available, the eMMC details can be written to a register. This register must have the special property that a write to the register generates a trace message containing the register contents. An example for such a register is the ContextID register of Arm CoreSight.

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

- at the beginning of eMMC access (ptrace):
  - 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 (ptrace):
  - eMMC device id
  - command executed
  - result code and other return codes

Since all trace entries receive a timestamp, the access duration can also be analyzed.

A possible example of access monitoring is shown below:



50244	run	cycle	symbol	ti.zero	data
-0473174155	0	info		4.542859208s	80000900
-0473169125	3	ptrace	\\vmlinux\core_core\mmc_start_request	4.542990735s	
-0473169093	3	info		4.542991878s	B0636D6D
-0473169086	3	info		4.542991903s	80000012
-0473169078	3	info		4.542991928s	80042400
-0473169060	3	info		4.542992328s	800000B5
-0473169052	3	info		4.542994103s	80000200
-0473169045	3	info		4.542994178s	80000400
-0473017774	0	ptrace	\\vmlinux\core_core\mmc_request_done	4.549135007s	
-0473017415	0	info		4.549142177s	B0636D6D
-0473017408	0	info		4.549142202s	80000012
-0473017400	0	info		4.549142227s	80000000
-0473017393	0	info		4.549142228s	80000900
-0473011322	2	ptrace	\\vmlinux\core_core\mmc_start_request	4.549269143s	
-0473011312	2	info		4.549270272s	B0636D6D
-0473011294	2	info		4.549270323s	80000012
-0473011286	2	info		4.549270348s	80042800
-0473011279	2	info		4.549270372s	800000B5
-0473011260	2	info		4.549271873s	80000200
-0473011253	2	info		4.549271898s	80000400
-0472852525	0	ptrace	\\vmlinux\core_core\mmc_request_done	4.555422827s	
-0472852137	0	info		4.555432798s	B0636D6D
-0472852130	0	info		4.555432848s	80000012
-0472852122	0	info		4.555433348s	80000000
-0472852115	0	info		4.555433398s	80000900
-0472845755	2	ptrace	\\vmlinux\core_core\mmc_start_request	4.555566242s	
-0472845734	2	info		4.555567223s	B0636D6D

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 recorded trace data can be filtered and saved to a file, and then converted into a more suitable format for analysis using a PRACTICE or Python script, or an external conversion program.

The trace information for a single eMMC access can, for example, 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 re-programs 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 table below:

Item	**Value**
Device Capacity	8GB
Write Endurance	2K Program/Erase Cycles
Data Written Per Day to Device	2GB
Expected Life w/ WA=1 =(8x2000)/(2*1)	8,000 days
Expected Life w/ WA=5 =(8x2000)/(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 analyzing 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 CoreSight 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 A: Source Code Example"**, page 15 or in the `~~/demo/etc/trace/emmc/` folder. 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 via the **Var.set** command:

```
Var.set T32_mmc.enable = 1
Var.set ((char*)&T32_mmc.dev) = "mmc0"
```

The `infoBit` field can be written as follows:

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

In order to distinguish between data written in the Context ID register by the instrumentation code from those written by Linux for task switches, the range of values used by the instrumentation code must be reserved so that they are not interpreted as task switch identifiers. The command **ETM.ReserveContextID** can be used for this:

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

The cycle type `task` is assigned to Linux task switches, the cycle type `info` is assigned to the instrumented code.

It's important to note that the Linux kernel must be compiled for debug (see **"Training Linux Debugging"** (training_rtos_linux.pdf)).

To reduce the amount of trace information generated by the target and to allow long-term trace via TRACE32 streaming (**Trace.Mode STREAM**), filters can be applied to isolate the eMMC code and its writes to the Context ID register. The **Break.Set** command can be used for this purpose:

```
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.

Tracing task switch information is not required for the eMMC analysis, but if you want that 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:

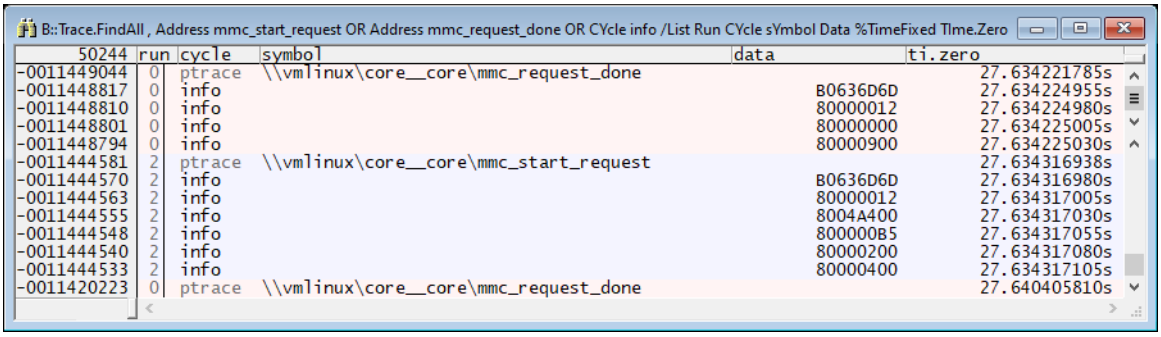
```
Break.Set   __switch_to+0x74 /Program /TraceON
Break.Set   __switch_to+0x80 /Program /TraceOFF
```

The recorded trace data can be filtered and saved to a file, and then converted into a more suitable format for analysis using a PRACTICE or Python script, or an external conversion program.

Use the command **Trace.FindALL** to filter and format trace data required for the eMMC analysis.

```
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.



50244	run	cycle	symbol	data	ti.zero
-0011449044	0	ptrace	\\vmlinux\core__core\mmc_request_done		27.634221785s
-0011448817	0	info		B0636D6D	27.634224955s
-0011448810	0	info		80000012	27.634224980s
-0011448801	0	info		80000000	27.634225005s
-0011448794	0	info		80000900	27.634225030s
-0011444581	2	ptrace	\\vmlinux\core__core\mmc_start_request		27.634316938s
-0011444570	2	info		B0636D6D	27.634316980s
-0011444563	2	info		80000012	27.634317005s
-0011444555	2	info		8004A400	27.634317030s
-0011444548	2	info		800000B5	27.634317055s
-0011444540	2	info		80000200	27.634317080s
-0011444533	2	info		80000400	27.634317105s
-0011420223	0	ptrace	\\vmlinux\core__core\mmc_request_done		27.640405810s

If the trace data are available as required, they can be saved in a file using the **PRinTer.File** command and the command prefix **WinPrint**.

```
PRinTer.FILE mmclog.txt ASCIIE

WinPrint.Trace.FindAll , Address mmc_start_request OR \
Address mmc_request_done OR CYcle info \
/List Run CYcle sYmbol Data %TimeFixed TIme.Zero
```

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.

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** (41source 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.

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/test/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 B: Time Details”**, page 18, 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

TRACE32 hardware-based trace tools provide 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 summarizes the advantages and disadvantages of the two considered solutions: TRACE32 and ftrace.

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 and 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 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 Corporation, 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 A: 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);
      }
```

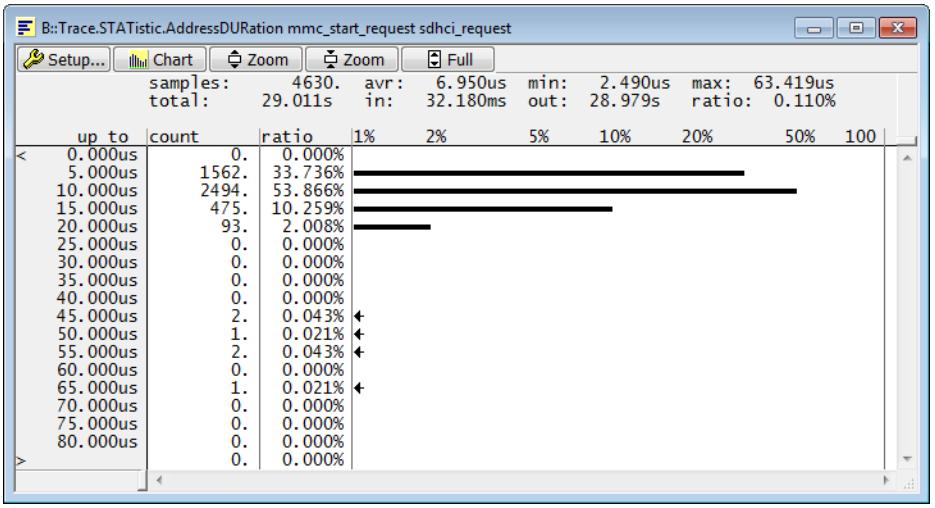
```
        if (mrq->stop) {
           pr_debug("%s:     (CMD%u): %d: %08x %08x %08x %08x\n",
              mmc_hostname(host), mrq->stop->opcode,
              mrq->stop->error,
              mrq->stop->resp[0], mrq->stop->resp[1],
              mrq->stop->resp[2], mrq->stop->resp[3]);
        }

        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 B: Time Details

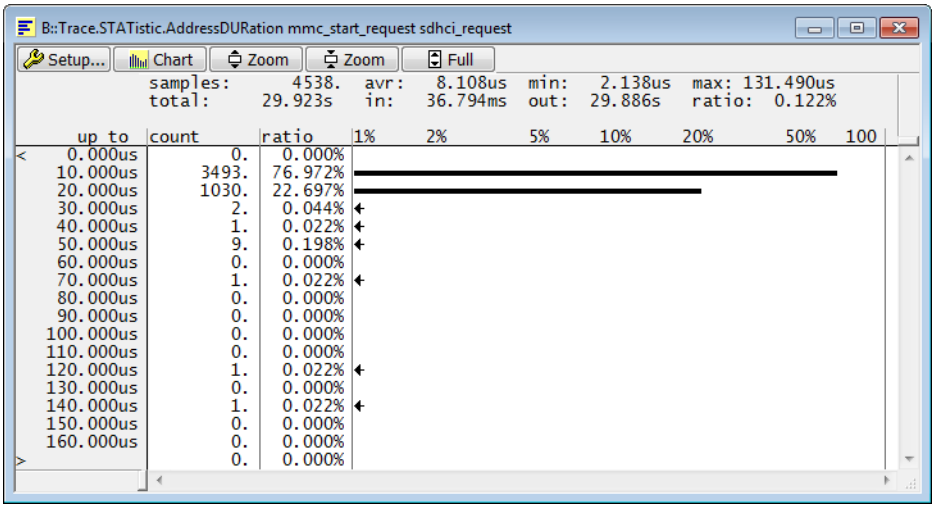
The **Trace.STATistic.AddressDURation** command was used for all time measurements.

1. Time duration analysis: mmc_start_request

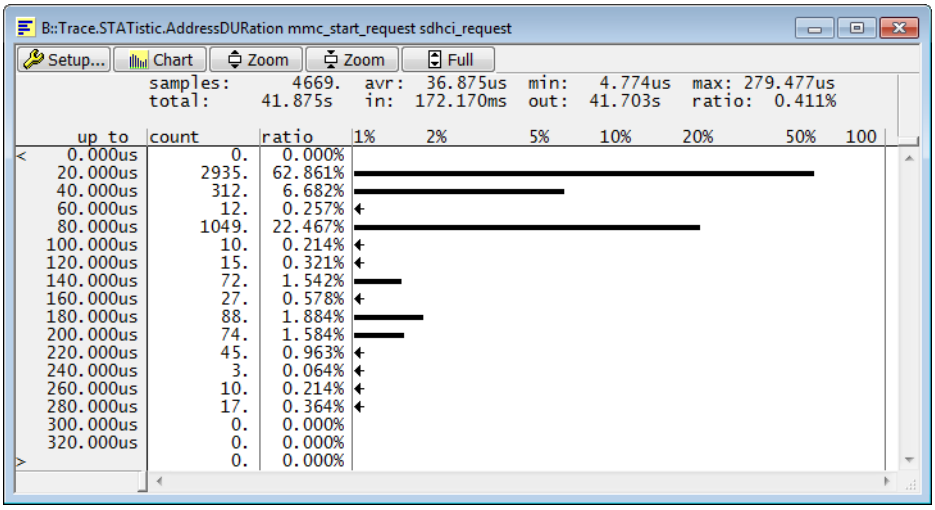
No ftrace, no TRACE32 instrumentation



No ftrace, with TRACE32 instrumentation

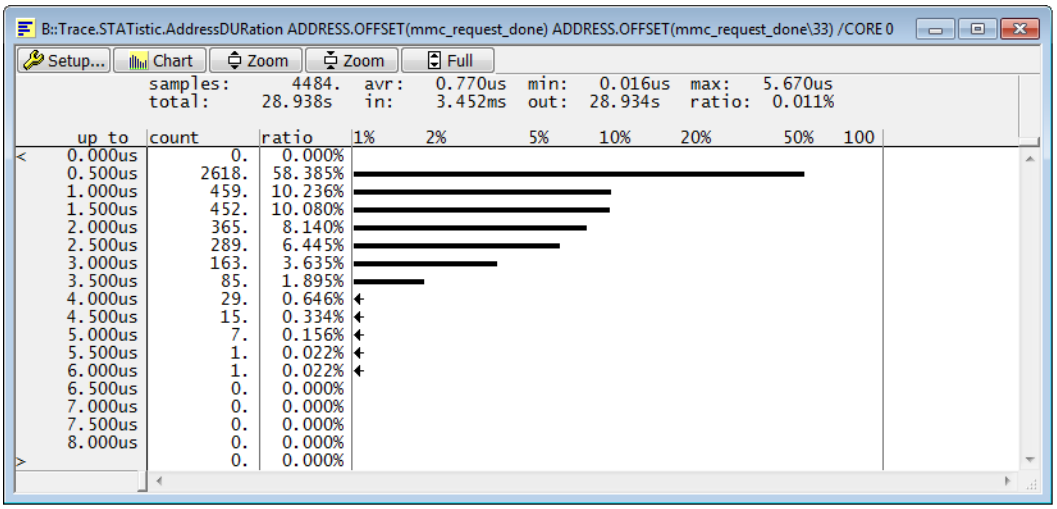


With ftrace, no TRACE32 instrumentation

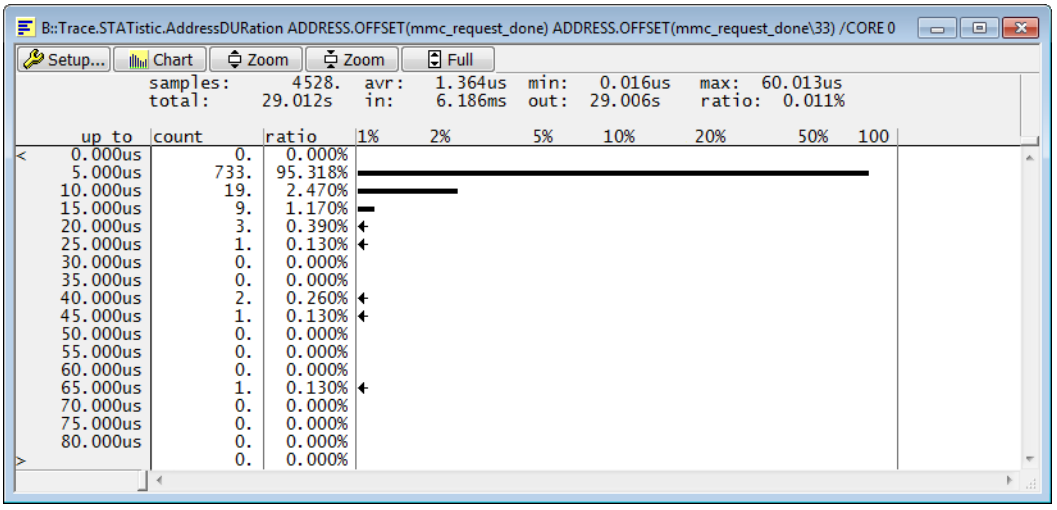


2. Time duration analysis: mmc_request_done

No ftrace, no TRACE32 instrumentation



No ftrace, with TRACE32 instrumentation



With ftrace, no TRACE32 instrumentation