MicroSD with SAMD51 SDHC

– August 2, 2018 RA

To begin, I wrote a SDHC controller, got it working and have since dropped it.

If you go through the SAMD5x/E5x family datasheet on the SDHC, you'll see that, for as complex as the SDHC controller is, it is minimally documented. Rather they rely on external documents: SD Host Controller Simplified Specification V3.00, SDIO Simplified Specification V3.00, Physical Layer Simplified Specification V3.01 from sdcard.org and Embedded MultiMedia Card (e.MMC) Electrical Standard 4.51 from jedec.org. Even with all these documents, there are large gaps of information.

Microchip has two examples which were helpful, FATFS example and SD/MMC raw example. I followed the ASF4 driver removing the mmc sections since it was intended for MicroSD. It still does the card detection process in a very similar manner. I kept and reused the sd_mmc_protocol.h along with a cleaned up version of the hri sdhc.h include. The files sd.c and sd.h are the new driver.

The FatFS is the CHaN FatFS and the file diskio.c implements the functions that FatFS requires from the SDHC driver. It will reference an include diskio.h which provides prototype for the FatFS. FatFS also requires a real time clock, We used the I2C Abracon AB1805 RTC rather than the onboard RTC.

The SDHC has four data pins SDDAT0-SDDAT3, SDCK clock, and SDCMD command. I also had SDCD card detect use the EIC controller as a debounced interrupt. These are initialized through the initmaker with

```
[GPIO]
pin=SDCK
function=SDHC0
[GPIO]
pin=SDCMD
function=SDHC0
[GPIO]
pin=SDDAT0
function=SDHC0
[GPIO]
pin=SDDAT1
function=SDHC0
[GPIO]
pin=SDDAT2
function=SDHC0
[GPIO]
pin=SDDAT3
function=SDHC0
[EIC]
ref_source=GCLK3
[GPIO]
eic=SDCD
sense=both
debounce=1
interrupt=1
```

The SDHC controller has a small chunk of startup code in the initmaker/extended directory that is inserted into the SystemInit() routine. called sdhc_init.c.

```
mclk_set_AHBMASK(MCLK_AHBMASK_SDHC0);
gclk_write_PCHCTRL(SDHC0_GCLK_ID, GCLK_PCHCTRL_GEN_GCLK4 |
GCLK_PCHCTRL_CHEN);
gclk_write_PCHCTRL(SDHC0_GCLK_ID_SLOW, GCLK_PCHCTRL_GEN_GCLK5 |
GCLK_PCHCTRL_CHEN);
sdhc_set_SRR(SDHC0, SDHC_SRR_SWRSTALL); // reset all
while(sdhc_get_SRR(SDHC0, SDHC_SRR_SWRSTALL)) {};
sdhc_write_TCR(SDHC0, 14); // max timeout is 14 or about 1sec
sdhc_write_PCR(SDHC0, SDHC_PCR_SDBPWR | SDHC_PCR_SDBVSEL_3V3);
sdhc_set_NISTER(SDHC0, SDHC_NISTER_MASK); // clear all normal
interrupt bits
sdhc_set_EISTER(SDHC0, SDHC_EISTER_MASK); // clear all error
interrupt bits
```

I used GLCK4 for high speed clock (100MHz from DPLL1) and GCLK5 for slow clock (12MHz from DPLL0/10). The SDHC seemed fussy about the clock rate and the datasheet only gave maximum. It will work at 12MHz but only if synchronized with the CPU, It would hang for > 12MHz and < 32MHz.

I was able to pull together code which would read, write, create files, create directories, change directories, get and set the label which all worked reliably.

I can get the status of the MicroSD that was plugged in and it reported the same information that an SD Card utility would. That is, whether to use one or four bit, clock rate and so on. Note that this SDHC doesn't do the low voltage 1.8V modes. This is based on compatibility registers in the SDHC which are reset to values which represent what the controller can and cannot do.

I did see on the oscilloscope that the signals matched what the status stated as the card capability. Usually four bit 25MHz clock.

I tested with an assortment of low end Gskill to the higher end Samsung EVO. I also tested these cards with Crystal Disk Mark on a PC. The performance seemed off though. I wasn't expecting speeds that a PC might see but it was noticeably slow.

I used the TC0 timer clocked from 1MHz with the following configuration

```
[TC0]
ref_source=gclk7
mode=32
wavegen=NFRQ
event=TS_EVENT
swgen=TS_EVENT
evact=STAMP
count=0
capten0=1
```

Used this for timestamp

```
#define TIMESTAMP() evsys_write_SWEVT(1 << TS_EVENT);
```

and this to get time between two timestamps.

```
uint32_t get_timestamp(void)
{
   uint32_t start_ts;
   tc_wait_for_sync(TC0, TC_SYNCBUSY_CTRLB);
   tc_set_CTRLB(TC0, TC_CTRLBSET_CMD_UPDATE);
   while (tc_get_CTRLB(TC0, TC_CTRLBSET_CMD_Msk)) {};
   start_ts = tccount32_read_CC(TC0, 0);
   tc_wait_for_sync(TC0, TC_SYNCBUSY_CTRLB);
   tc_set_CTRLB(TC0, TC_CTRLBSET_CMD_UPDATE);
   while (tc_get_CTRLB(TC0, TC_CTRLBSET_CMD_Msk)) {};
   return (tccount32_read_CC(TC0, 0) - start_ts);
}
```

It uses the difference between two captures to find the time elapsed in 0.1 microseconds. Setting up and sending the command was 13.1us for doing a block read of 512 bytes. The time from when the command to read is sent and when the buffer indicates it is full was 784us. The time to move memory from the buffer to SRAM was 14us. That is, 811.1us or 631KB/s and the bulk of the time is waiting on the SDHC to return from the command. In contrast, if I used SPI in single bit mode at 25Mb/s, I'd still expect around 2MB/s data rate.

I did this on the E54 Xplained board with the FatFS example modified to add a timer for timestamps. This was to provide Microchip with a test case. The results were worse as they clocking everything at 12MHz and the ASF4 implementation wasn't very efficient.

I asked Microchip in May about the performance and am still awaiting MCHP internal feedback. The code is there, If someone wants to take a crack at it. I exceeded the alotted time to work on it and have to move on.

Addendum

I received a response from Microchip:

> *When checking with SD/MMC Raw example read throughput comes around 900+ KB/s since command/response based protocol. This can be checked using default START example.*
> — Microchip Support 9/11/2018

I got less with the E54 dev board and START code but still closed out the case. The 900KB was close to what I could get with my code.

I was contacted by a hardware engineer from IRCAM (Paris France), Emmanuel Fléty as he was working on a SAMD51 project using the SDHC. He had taken the performance experiments further.

To begin, we expect a 25MHz SD card with 4 bit access to take 41µs (12.5MB/s).

He found that the SD card itself delays response for the first block and then contiguous blocks access at our expected speed. Say the setup for the first block is 400 µs plus the 41µs transfer. If we are doing single block transfers only using CMD17, we will get a maximum of 1.1MB/s or worse. The setup time varies from SD card to SD card; some cards had delays upwards to 1ms.

He tried CMD18 which accesses contiguous blocks. You still get the first block setup time but the subsequent sectors do not require that setup time. There is a slight turnaround penalty of a few microseconds though. The speed becomes 400 µs + (N*41µs) where N is the number of contiguous blocks accessed. That is, if we read 16 contiguous blocks our data rate increases to 3.9MB/s.

This issue is similar to the hard drive seek time, that is the time for the hard drive head to move to a cylinder before it could start reading contiguous sectors. Manufactures came up with strategies to deal with that delay and improve the access time. A few ideas are to access as large clusters of contiguous sectors or cache the accesses. I suspect the USB SD card readers along with the PC are doing that and the performance tools that check SD Card random access reflect that improvement.

Emmanuel also got DMA working. The SDHC has its own DMA controller and does not use the DMAC. It doesn't really save much time with respect to transfer speed. He read 8 blocks with his code and was seeing speeds of 2MB/s. So it doesn't look like there is a performance problem with the SDHC, rather the microSDs are not suited for random access.