PROJECT 2: µSD CARD READERS

OVERVIEW

For this project you will modify a program to recover CPU idle time in two ways. The program reads data from and writes data to a µSD card using the SPI interface. The CPU is fast enough that at times it waits for a SPI byte transfer to complete, and at other times it waits for the µSD card's controller to complete an operation. You will modify the base program to create two different versions: in Part A you will use a non-preemptive, non-prioritized scheduler (a while(1) loop which calls task functions) while in Part B you will use a preemptive, prioritized scheduler (the RTX5 real-time kernel).

The base program uses the open source SD card interface code **ulibSD** (available on github) by Nelson Lombardo based on work by Chan. The program does the following:

- Initializes card controller
- Repeats these steps, starting with the first sector and then advancing.
  - Reads the next 100 sectors (blocks) of data (each 512 bytes long) from the µSD card.
  - Writes test data to the next sector and reads it back to verify correct operation.
- Also does make-work (approximating π) which represents other processing/threads in the program.

You will need a working µSD flash memory card to complete this project. The code will erase and rewrite sectors, so be sure to copy any critical information off the card before using it for this project. The card can be of any size; 1 GB is more than enough.

You will use your logic analyzer **extensively** to examine the system's behavior by probing the SPI communication signals as well as debug signals (twiddle bits) which you'll add to the code.

The µSD card communication approach is described in detail in the SD Specifications listed in "Further Information" below. Here is a quick summary. An SD or µSD card contains a flash memory array (for data storage), a controller and multiple communication interfaces (one generic single-bit SPI interface and faster, proprietary multi-bit interfaces). The MCU will use the SPI interface to talk with the SD card controller. The MCU operates as the master and initiates all communications; the card controller is the slave and can only respond. After power-up (or card insertion), the MCU must initialize the card controller and select SPI communications.

SPI communications are bit-serial, byte-oriented and full-duplex (there are two data lines, MOSI and MISO). A SPI data transfer simultaneously transfers a byte from the master to the slave (via MOSI) and another from the slave to the master (via MISO). Refer to Chapter 8 of **Embedded Systems Fundamentals** and the SPI chapter of the **KL25Z Subfamily Reference Manual** for SPI details.

The master must ensure the card controller is ready before sending a command. The master does this by polling the card controller for its status, repeating until the status is correct. The master can then send a command. The master must poll the controller to determine if the command has been processed and completed. SD commands available with the SPI interface are listed in Chapter 7 of the **SD Specifications: Physical Layer Simplified Specification.**

PERIPHERALS USED

- The SPI interface communicates with the µSD controller.
- The supplied code uses a timer (LPTMR0) to measure time for delays.
- You will use GPIO peripherals for outputs to indicate current system activity by asserting debug signals on Port B.

FURTHER INFORMATION

- **Embedded Systems Fundamentals**, Chapters 3 and 8 (especially introduction and SPI).
- **FRDM-KL25Z Subfamily Reference Manual**.
- **SD Specifications: Part 1 Physical Layer Simplified Specification**, Version 6.00, April 10, 2017, SD Card Association Technical Committee. Located in project's Documents folder.
- How to use MMC/SDC, http://elm-chan.org/docs/mmc/mmc_e.html
- CMSIS-RTOS2: https://www.keil.com/pack/doc/CMSIS/RTOS2/html/index.html
- RTX v5: https://www.keil.com/pack/doc/CMSIS/RTOS2/html/rtx5_impl.html
- Source code for uLibSD, https://github.com/1nv1/ulibSD. Do not use this code for this project, as it lacks necessary modifications. Instead use the code from the class NCSU github repository.

PART A: FSM

PROGRAM STRUCTURE

This version of the SD card program uses round-robin, non-preemptive task scheduling, provided by the function Scheduler in main.c. It calls three tasks repeatedly:

- Task_Makework is an arbitrary task representing **other work** in the system which could continue executing during idle time in the SD card accesses. Recovering idle time from the SD and SPI operations will let this task run more often and finish sooner. This task slowly approximates π using double-precision floating point math.
- Task_Test_SD initializes the SD card and then repeatedly scans through the card with this sequence: reading 100 sectors and then writing one sector (verifying the data was written correctly). This task does not access the SD card directly, but instead sends commands to Task_SD_Server which processes them. See Figure 1. This is a common design architecture for non-preemptively-scheduled systems which must be responsive despite slow or potentially blocking operations.
- Task_SD_Server carries out commands to initialize the card, read a sector or write a sector. You are encouraged to examine the inter-task (Task_Test_SD <-> Task_SD_Server) communication mechanism, but it is not a requirement for this project.

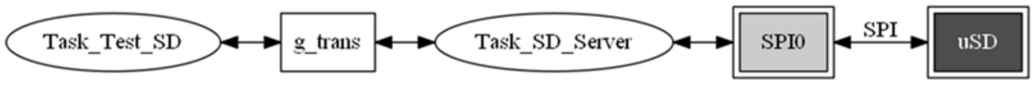


Figure 1. Communication between SD tasks and µSD card

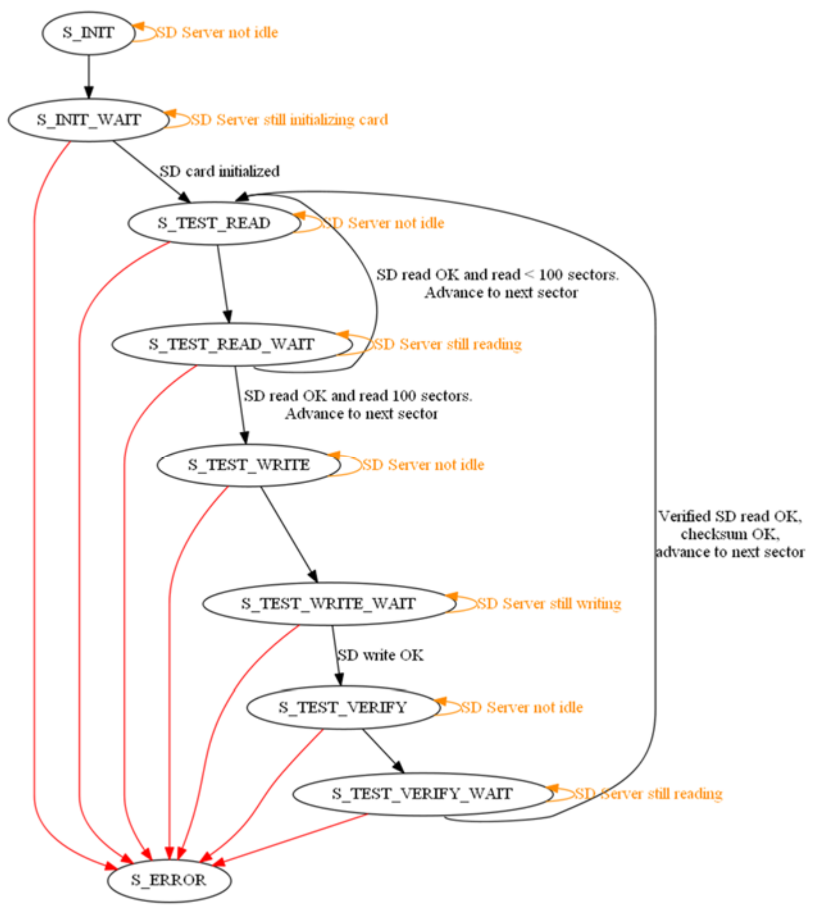


Figure 2. Finite state machine for Task_Test_SD

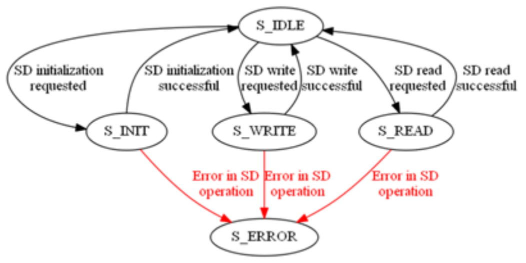


Figure 3. Finite state machine for Task_SD_Server

All three of these tasks are implemented as finite state machines in order to improve responsiveness. The FSMs for the two latter tasks are shown in Figure 2 and Figure 3. Task_Makework operates iteratively and the function body is one state.

MODIFICATION: ADD DEBUG SIGNALS

Add code to create and control the software-generated debug signals as shown in the table below. These will be used in timing analysis, debugging, generating screenshots for your report, and grading. Refer to the appendix for an example of the signals in action.

Signal Name	Shield Signal	Logic Analyzer Channel	HW or SW Generated?	Notes
SPI CLK	CLK	0	HW	SPI Clock
SPI DO (MISO)	DO	1	HW	MISO (master in, slave out)
SPI DI (MOSI)	DI	2	HW	MOSI (master out, slave in)
SPI CS	CS	3	HW	SPI Chip Select (active low)
SD_Read	Debug 2	5	SW	Set to 1 immediately after entry to this function, clear to 0 immediately before returning from this function.
SD_Write	Debug 3	6	SW	
SD_Init	Debug 4	7	SW	
Task_SD_Server	Debug 5	8	SW	
Task_Test_SD	Debug 6	9	SW	
Task_Makework	Debug 7	10	SW	

MODIFICATION: CONVERSION TO FINITE STATE MACHINES

Task_SD_Server calls three functions (SD_Init, SD_Read and SD_Write) directly. These functions take a long time to complete, increasing the maximum state time for Task_SD_Server. You will convert the three functions SD_Init, SD_Read and SD_Write into finite state machines, where each state meets the timing budget shown below. Use the approach described in the lecture **Converting C Tasks to FSMs**.

Table 1. Timing budgets (maximum state times)

Function	**ECE 460**		**ECE 560**	
	Full Credit	**10% Extra Credit**	**Full Credit**	**10% Extra Credit**
SD_Init	1 ms	500 µs	250 µs	100 µs
SD_Read	1 ms	500 µs	200 µs	75 µs
SD_Write	1 ms	500 µs	200 µs	75 µs

You will also need to convert Task_SD_Server where it calls your new FSM versions of these three functions. Because the FSM function may return before completing, the calling function will need to determine if the FSM completed before it can test the result.

The general case of generating FSMs which cross function call boundaries (i.e. splitting an FSM state within a called function) is complex. However, this program's calling structure simplifies the FSM generation process.

- First, only one function (Task_SD_Server) calls SD_Init, SD_Read or SD_Write.
- Second, these three functions are never called in an interleaved way: a new function is never started until the previously called function has completed (returned an idle status, indicating completion).

With these two limitations we do not need to consider the possibility of multiple callers or interleaved accesses, so Task_SD_Server can safely call the FSM versions without significant modifications.

PROCEDURE

You are recommended to follow this procedure for each function (Task_SD_Server, SD_Read, SD_Write, SD_Init). Note that the report template identifies the diagrams and analysis required.

TIMING ANALYSIS

1. Use a logic analyzer to determine the SPI communication rate for the starter code.
2. Raise the SPI bit rate by modifying the function **SPI_Freq_High** in spi_io.c. The faster you can run the SPI link, the fewer states you'll need in your FSMs. Refer to Chapter 8 of the textbook and the SPI chapter of the KL25 Subfamily Reference Manual for details.
3. Add twiddle output bits to understand the timing of the three SD functions. Some debug outputs and macros have been defined for you already in debug.h, but you'll need to add code in your SD functions to set, clear or toggle the bits (see next item).
4. Use a logic analyzer to determine function timing information. Trigger the logic analyzer on the rising edge of the function's twiddle bit. You may need use single run (capture) mode to trigger only on the first rising edge. See the example timeline screenshot in the appendix. You can determine how long the function takes to reach a given

internal point by inserting a call to DEBUG_TOGGLE(channel number) there. Use these signals to identify when the function…

   a. starts (sets the bit)
   b. finishes (clears the bit)
   c. is blocking in a waiting loop (toggles the bit twice)

5. Mark the diagram to identify the types of time segments for each function:
   a. Compute: Depends only on computation time.
   b. I/O-SD: Delay mainly due to waiting for SD card controller. Making SPI infinitely fast would not reduce this segment significantly.
   c. I/O-SPI: Delay mainly due to waiting for SPI data transfer rate. Making the SD card controller infinitely fast would not reduce this segment significantly.
   d. Other/None of the above

CODE STRUCTURE ANALYSIS

6. Obtain or create a CFG for each of the three functions which will be turned into FSMs. The CFGs for SD_Init and SD_Write have been provided for you as PDFs in the project's Documents directory. They were created using a commercial code analysis tool (CrystalFLOW for C, by SGV Software Automation Research Corp.). You will need to draw a CFG for SD_Read.
7. Examine the CFG and mark the loops with unknown or long durations.

CODE TRANSFORMATION

8. The CFG can be thought of as an FSM with all the code contained in a single state. Split this state into multiple states which meet the timing goals (e.g. no more than X ms in any state).
9. Convert the function's code to implement the FSM you designed above. Modify the input and output data mechanisms (function parameters and return value) as described in the **C to FSM** lecture notes and as discussed in class. Note that you will need to modify the calling function to repeatedly call an FSM function until its return status is idle.
10. Verify the FSM works properly.

PART B: RTOS

PROGRAM STRUCTURE

The starter program for Part 2B is a bare-bones version which lacks the SD server task and instead directly accesses the SD card. Task_Test_SD does not use an FSM because it is not needed. Using a preemptive scheduler allows the program to be structured in a more natural form and simplifies development. Start with Project_2B_Base (derived from ulibSD) and modify it as needed to use RTX5 as described in the Procedure section below.

MODIFICATION: ADD DEBUG SIGNALS

Add code to create and control the software-generated debug signals as shown in the table below. These will be used in timing analysis, debugging, generating screenshots for your report, and grading. **Note that these signals are slightly different from the Part A debug signals.**

Signal Name	Shield Signal	Logic Analyzer Channel	HW or SW generated?	Notes
SPI CLK	CLK	0	HW	SPI Clock
SPI DO (MISO)	DO	1	HW	MISO (master in, slave out)
SPI DI (MOSI)	DI	2	HW	MOSI (master out, slave in)
SPI CS	CS	3	HW	SPI Chip Select (active low)
SPI_RW	Debug 1	4	SW	Set to 1 when function starts executing, toggle once per busy-wait loop, set to 1 after exiting busy-wait loop, clear to 0 at end of function
SD_Read	Debug 2	5	SW	
SD_Write	Debug 3	6	SW	
SD_Send_Cmd	Debug 4	7	SW	
SD_Init	Debug 5	8	SW	
SPI ISR	Debug 6	9	SW	Set to 1 when SPI1 ISR starts executing, clear to 0 at end of ISR
Idle	Debug 7	10	SW	Toggle once per iteration of osRtxIdleThread loop

PROCEDURE

Use the following procedure to develop your code. Where specified, take logic analyzer screenshots for your report.

TIMING ANALYSIS

1. Start with the starter code for Project 2B.
2. Add debug signals (twiddle bits) to understand the timing of the three SD functions. Some debug outputs have been defined for you already in debug.h, but you'll need to add code in the SD and SPI functions to set, clear or toggle the bits.

PORT TO CMSIS-RTOS2

You will change the code to execute on an RTOS.

3. Add CMSIS-RTOS2 support as described in the lecture slides on **Developing RTOS-Based Applications**.
   a. Set the system tick frequency to 2 kHz (change from the default of 1 kHz).
   b. Disable round-robin thread switching.
   c. Turn the functions Task_Makework and Task_Test_SD into threads.
   d. Verify proper code operation and debug signal behavior.

IDLE TIME ANALYSIS

4. Add code to the function osRtxIdleThread (in RTX_Config.c):
   a. Toggle the idle debug signal once per loop iteration. Use this to view when the idle loop is executing.
   b. Create a global uint32_t variable called idle_counter, initialized to 0. Increment idle_counter each time through the idle loop.
5. Add code to determine the idle_counter value for one second of idle time. This code must read the idle_counter before and after a one-second call to osDelay() with no other threads active. The difference in readings indicates the "100% idle for one second" reading of idle_counter. You will use this value later to determine how much idle time is available in various functions.
   a. What is the "100% idle for one second" value for idle_counter?
6. Idle loop analysis
   a. How long is an average idle loop iteration (based on 100 iterations)? Include a screenshot.
   b. How often does the scheduler tick run, and how long does it take each time? Determine this using the gaps in the idle debug signal. What fraction of the processor's time does the timer tick use? Include a screenshot.
7. How long does each of the following functions take, and how long is each segment (computation, SPI comm. for polling, other SPI comm.)? Include a screenshot marked to indicate the busy waiting time.
   a. SD_Init
   b. SD_Read
   c. SD_Write

SD_INIT MODIFICATION

SD_Init has a mix of operations, including a time delay, SPI communications with time-outs and polling the SD card.

8. Replace the 500 ms time delay operation in SD_Init with an osDelay call. If you wish to make your delays consistent regardless of tick frequency, you need to use the function osKernelGetTickFreq(), which returns the tick frequency in Hz.
9. Add code to read idle_counter before and after the SD_Init operation to determine available idle time.
10. Analysis
    a. How long does SD_Init take to execute? How long is each segment now? Include a screenshot.
    b. How much idle time is available now, based on idle_counter?
    c. Do these values differ from the previous implementation? Why or why not?

SD_READ MODIFICATION

The SD_Read function sends a "read block" command (CMD17) and uses a while loop to poll the SD card. The card returns a token with a value of 0xFF until the read has completed.

11. Change this code to use an osDelay() call to reduce the polling frequency to once per timer tick.
12. Analysis
    a. How long does the SD_Read function take to complete? How long is each segment now? Include a screenshot.
    b. How much idle time is available now, based on idle_counter?
    c. Do the times differ from the previous implementation? Why or why not?

SD_WRITE MODIFICATION

The SD_Write function sends a “write block” command (CMD24) and the data to be written, and then uses a while loop to poll the SD card. The card returns a token with a value of 0xFF until the write has completed.

13. Change this code to use an osDelay() call to reduce the polling frequency to once per timer tick. Include a screenshot from the logic analyzer.
14. Analysis
    a. How long does the SD_Read function take to complete? How long is each segment now? Include a screenshot.
    b. How much idle time is available now, based on idle_counter?
    c. Do the times differ from the previous implementation? Why or why not?

SPI_RW MODIFICATION

SPI_RW has two busy-waiting loops. The first waits until the transmit buffer is empty, and the second waits until the receive buffer is full. At low SPI bit rates (in SD_Init) these loops waste large amounts of time, but high rates they waste much less time. You will modify the code to use the SPI interrupt, its ISR and an RTOS message queue to eliminate the second busy-waiting loop. You will change SPI_RW’s receive buffer wait to use either the original busy-waiting loop or an OS wait (specifically, a blocking call to osMessageQueueGet()).

15. Create a SPI message queue which can hold up to eight bytes (char) using osMessageQueueNew().
16. Add support for the two different waiting modes.
    a. Create a variable in spi_io.c which determines which waiting mode to use.
    b. The second loop in SPI_RW() waits until the receive buffer is full. Add another version which calls osMessageQueueGet() in order to get the latest data byte received from SPI1. Select between the two versions based on the waiting mode.
17. Add SPI1 interrupt support.
    a. Create an interrupt service routine SPI1_IRQHandler() in SPI_IO.c to read the received data byte and put in in the SPI message queue. This ISR must indicate its start (and end) of execution activity by setting (and clearing) the ISR debug output signal.
    b. Modify SPI_Freq_High() to disable the SPI1 interrupt and select the busy-wait mode.
    c. Modify SPI_Freq_Low() to enable the SPI1 interrupt and select the OS wait mode.
18. SPI Processing Analysis. Consider the operations occurring when executing SPI_RW() once (assuming SPI_Freq_Low() has been called). Mark these events and times on a screenshot.
    a. How long does SPI_RW() take to execute (DBG_SPI_RW rising edge to falling edge)?
    b. How much of this time is used for SPI communications (observe SPI Clock)?
    c. How much idle time is available during an execution of SPI_RW()?
    d. What is the delay from SPI communication completion to the start of SPI1_IRQHandler()? How much of this is the Cortex-M0+ CPU’s delay in responding to an interrupt (see ESF chapter 4 and KL25 Reference Manual)?
    e. How long does SPI1_IRQHandler() take to execute?
    f. What is the delay from the SPI1_IRQHandler() completing to SPI_RW() completing? How much of this is the Cortex-M0+ CPU’s delay in responding to an interrupt (see ESF chapter 4 and KL25 Reference Manual)? What else causes this delay?

19. **560-Only: Alternatives**
    a. Why are we using a message queue instead of simply inserting osDelay(1) calls in the SPI_RW() blocking loops?
    b. Why didn't we change the first blocking loop in SPI_RW?
    c. Estimate how long it would take for SPI_RW() to execute if we raised the SPI bit rate to 1 MHz.
20. Thread Analysis
    a. How long does SD_Init take to execute? How long is each segment now? Include a screenshot.
    b. How much idle time is available now, based on idle_counter?
    c. Do these values differ from the previous implementation? Why or why not?

ECE 560-ONLY: ELIMINATING THE USE OF THE SPI TIMER

The original code uses a hardware timer (LPTMR0) to provide time-outs on certain SD operations. The RTOS has similar functionality using the SysTick timer and software. The osKernelGetTickCount() function returns the current kernel tick count. The osKernelGetSysTimerCount() function returns the current value of the hardware timer (SysTick Timer), but you probably don't need such fine-grain timing resolution.

21. Find all of the loops with time-outs in the functions (SD_Read, SD_Write, SD_Init) and explain what each loop does and what the timeout period is.
22. Change the code to free up LPTMR0 by using the RTOS timer tick support functions (described above) to provide equivalent time-outs instead.

DELIVERABLES

- Zipped archives of project directories for Part A and Part B submitted via Moodle. Reduce the archive sizes by cleaning the project targets in MDK-ARM before zipping the directory (Project -> Clean Targets, or Shift-F7).
- Project report. Use provided Project 2 template.

APPENDIX: EXAMPLE OF DEBUG SIGNALS FOR PART A

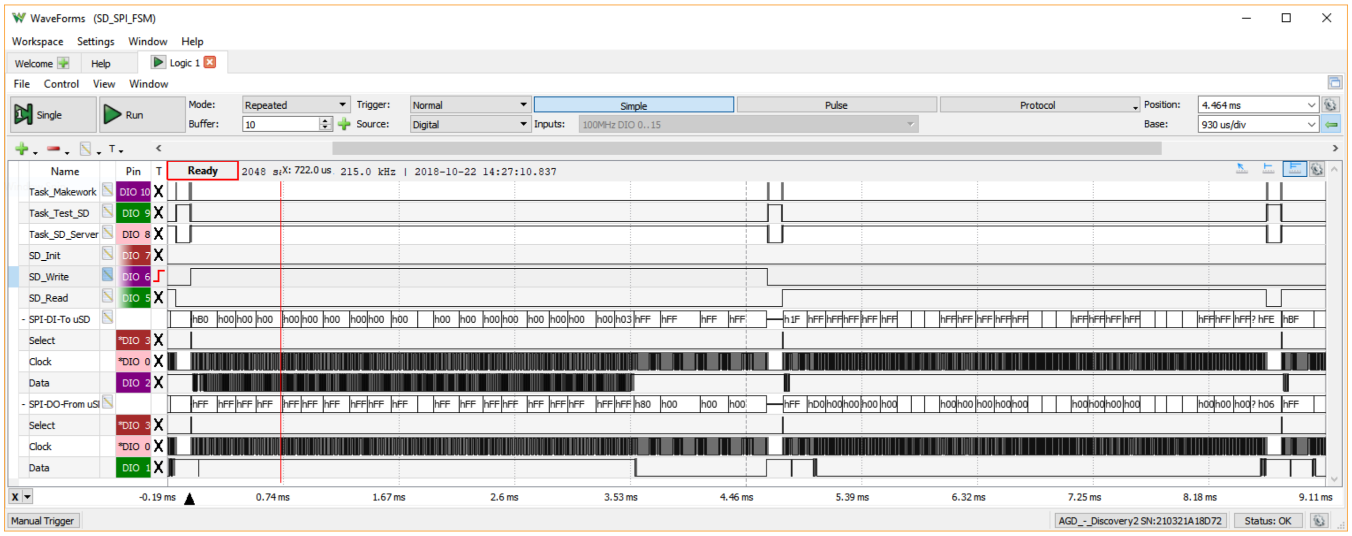


Figure 4. Example of logic analyzer output showing typical task and function debug signals when triggered by start of SD_Write.

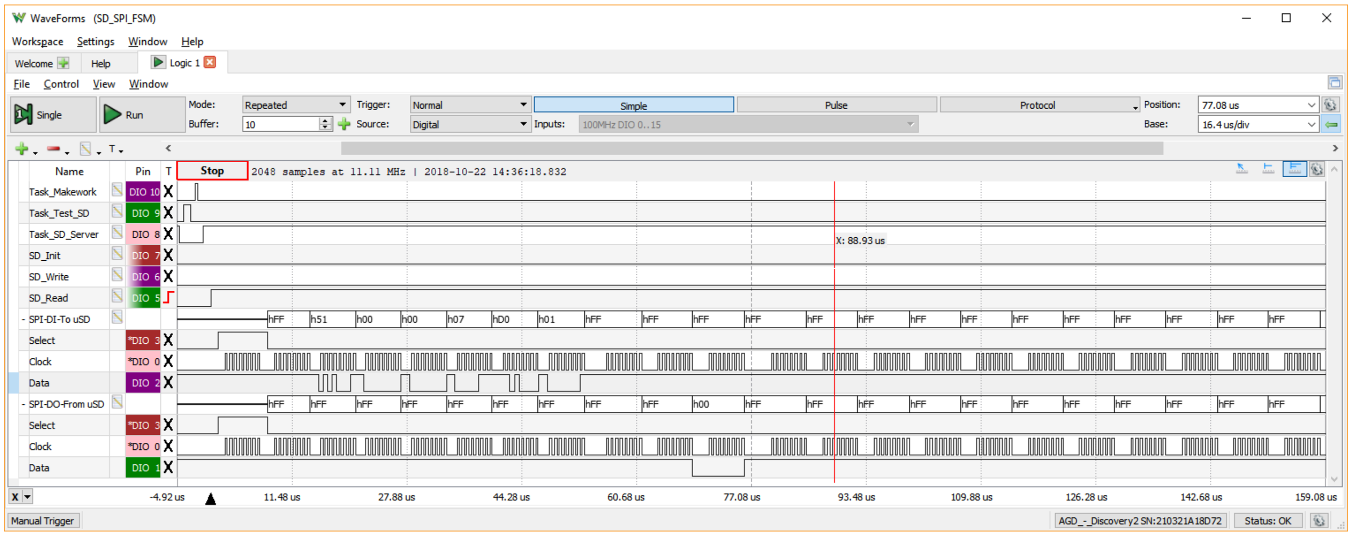
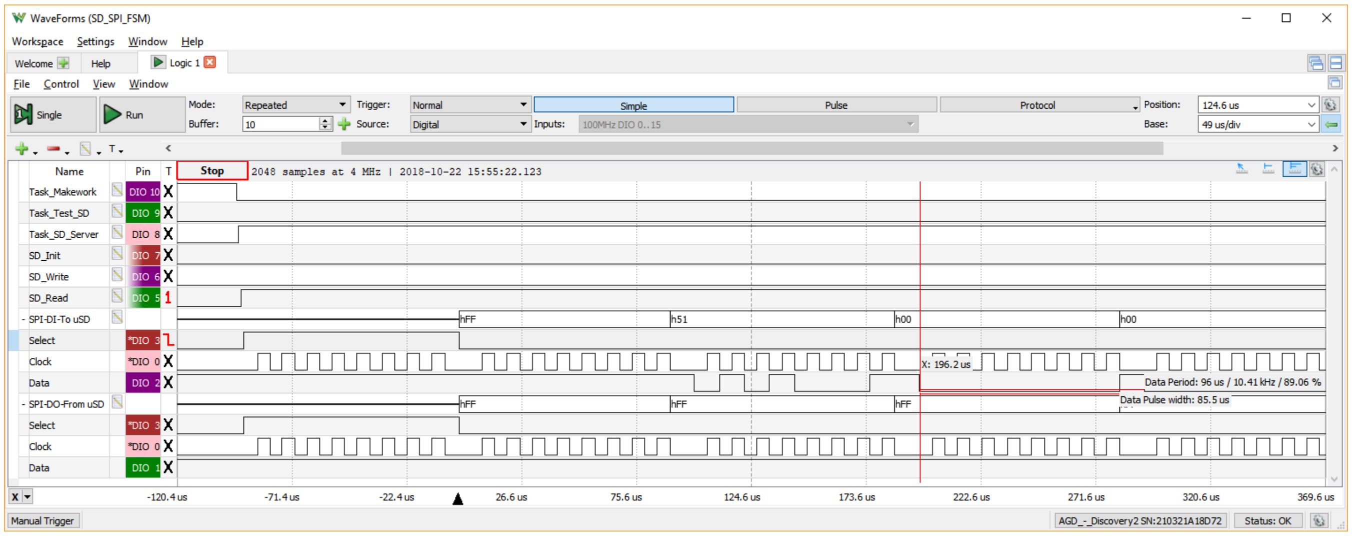


Figure 5. Example of logic analyzer output showing typical SPI signals when triggered by start of SD_Read.



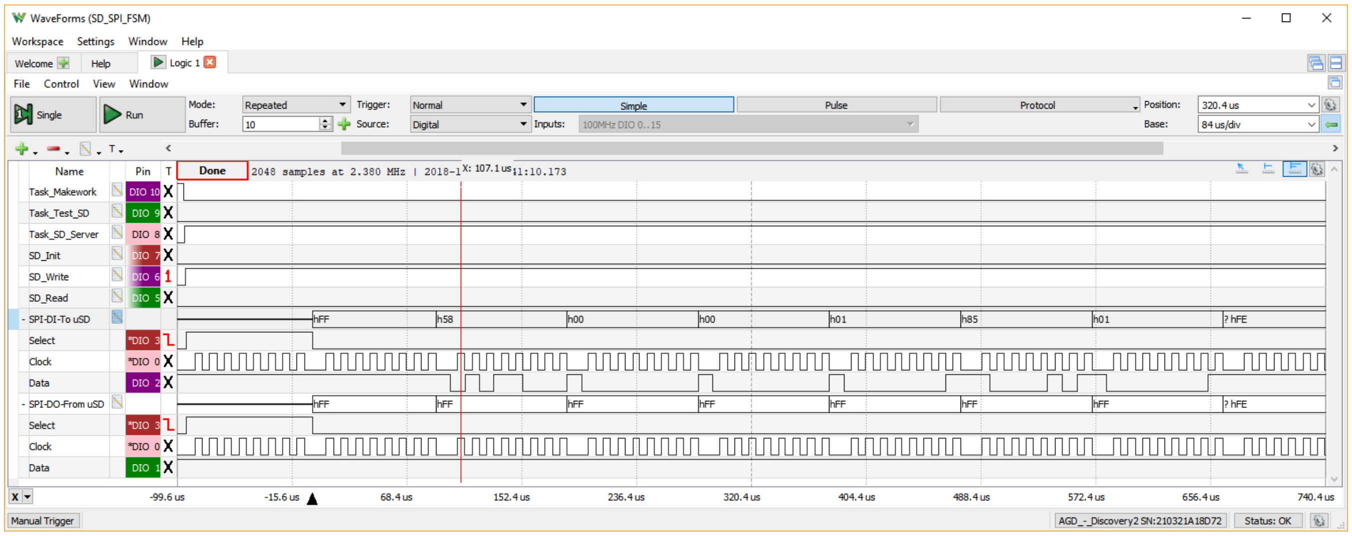
Mode: Repeated Trigger: Normal
Buffer: 10 Source: Digital Inputs: 100MHz DIO 0..15
Position: 124.6 us
Base: 49 us/div

2048 samples at 4 MHz | 2018-10-22 15:55:22.123

Data Period: 96 us / 10.41 kHz / 89.06 %
Data Pulse width: 85.5 us

AGD_-_Discovery2 SN:210321A18D72 Status: OK

Figure 6. Details of SPI communication at start of reading block. Command h51 is 0x51 = 0x40 + CMD17. Note that the CMD numbers are listed in decimal, so CMD17 = 17_{10} = 11_{16} = 0x11. See SD Spec. section 7.3 for details.



Mode: Repeated Trigger: Normal
Buffer: 10 Source: Digital Inputs: 100MHz DIO 0..15
Position: 320.4 us
Base: 84 us/div

2048 samples at 2.380 MHz | 2018-1 11:10.173

AGD_-_Discovery2 SN:210321A18D72 Status: OK

Figure 7. Details of SPI communication at start of writing block. Command h58 is 0x58 = 0x40 + CMD24. Note that the CMD numbers are listed in decimal, so CMD24 = 24_{10} = 18_{16} = 0x18.