Vol 53 No 4, December 2019

RAQ Issue 172: Manipulating MCU SPI Interface to Access a Nonstandard SPI ADC

Steven Xie, Product Applications Engineer

Question:

Can I access a nonstandard SPI interface with my MCU?

Answer:

Yes, but it might take a little extra effort.

Introduction

Many current precision analog-to-digital converters (ADCs) have a serial peripheral interface (SPI) or some serial interface to communicate with controllers including a microcontroller unit (MCU), a DSP, or an FPGA. The controllers write or read ADC internal registers and read conversion codes. SPI is becoming more and more popular due to its simple printed circuit board (PCB) routing and a faster clock rate compared to parallel interface. And, it is easy to connect an ADC to the controller with a standard SPI.

Some new ADCs have an SPI, but others have a nonstandard 3-wire or 4-wire SPI as a slave because they want to achieve a faster throughput rate. For example, the AD7616, AD7606, and AD7606B family has two or four SDO lines for faster throughput rate in serial mode. The AD7768, AD7779, and AD7134 families have multiple SDO lines and they work as SPI masters. Users tend to encounter difficulties in designing microcontroller SPIs for ADC configuration and code reading.

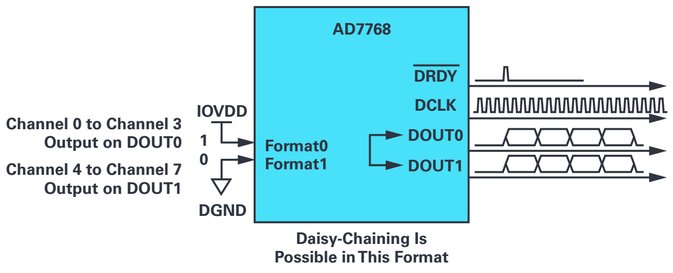

AD7768
DRDY
DCLK
DOUT0
DOUT1
Format0
Format1
IOVDD
1
0
DGND
Channel 0 to Channel 3 Output on DOUT0
Channel 4 to Channel 7 Output on DOUT1
Daisy-Chaining Is Possible in This Format

Figure 1. AD7768 as a serial master with two data output pins (14001-193).

Standard MCU SPI Connection to an ADC

SPI is a synchronous, full-duplex, master/slave-based interface. The data from the master or the slave is synchronized on the rising or falling clock edge. Both master and slave can transmit data at the same time. Figure 2 shows a typical 4-wire MCU SPI interface connection.

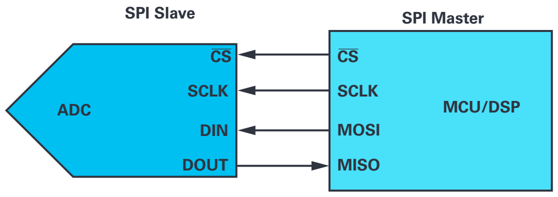


Figure 2. Standard MCU SPI connection to an ADC slave.

To begin SPI communication, the controller must send the clock signal and select the ADC by enabling the $\overline{CS}$ signal, which is usually an active low signal. Since SPI is a full-duplex interface, both the controller and ADC can output data at the same time via the MOSI/DIN and MISO/DOUT lines, respectively. The controller SPI interface provides the user with flexibility to select the rising or falling edge of the clock to sample and/or shift the data. For reliable communication between the master and the slave, users must follow the digital interface timing specifications of both the microcontroller and the ADC chip.

If the microcontroller SPI and ADC serial interface have the standard SPI timing mode, it is not a problem for users to design the PCB routing and develop the drive firmware. But there are some new ADCs with a serial interface port that is not a typical SPI timing pattern. It does not seem possible for the MCU or the DSP to read data through the AD7768 serial port, a nonstandard timing SPI port, as shown in Figure 4.

This article will introduce approaches to manipulating the standard microcontroller SPI to interface with ADCs that have nonstandard SPI ports.

This article will cover four different solutions to read the ADC codes by serial interface:

- Solution 1: MCU as SPI slave interfacing to ADC as SPI master by one DOUT line.
- Solution 2: MCU as SPI slave interfacing to ADC as SPI master by two DOUT lines.
- Solution 3: MCU as SPI slave interfacing to ADC as SPI master through DMA.
- Solution 4: MCU as SPI master and SPI slave to read data on two DOUT lines.

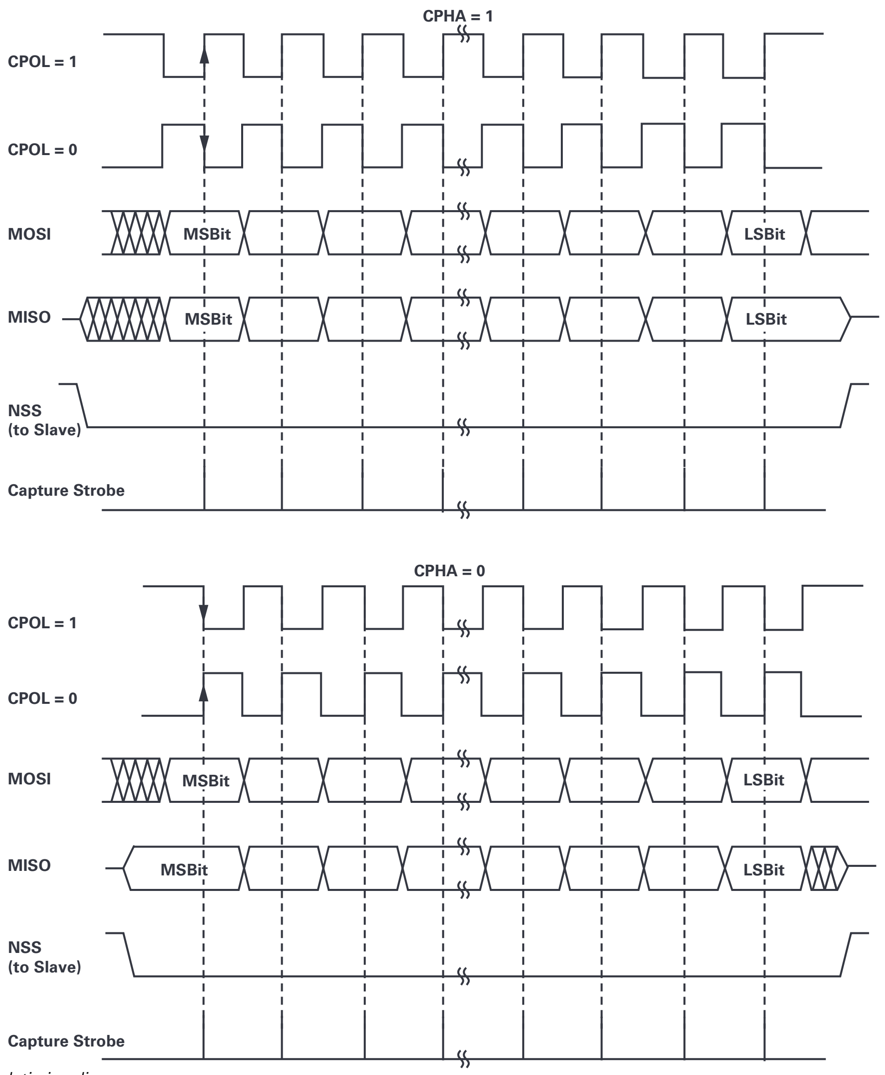


Figure 3. Example SPI data clock timing diagram.

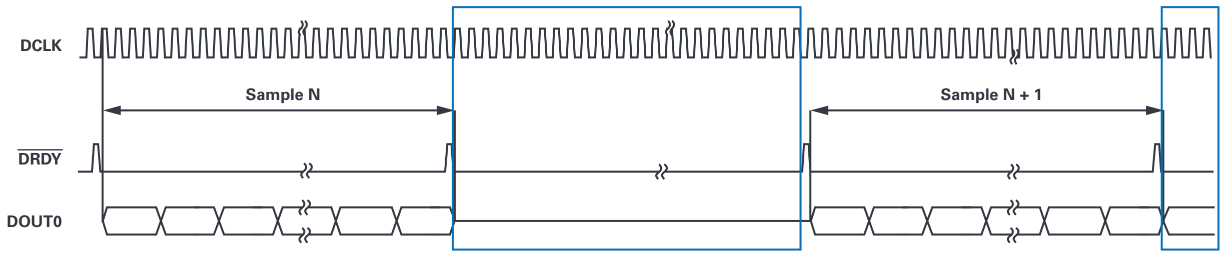


Figure 4. AD7768 FORMATx = 1× timing diagram output on DOUT0 only.

AD7768 Code Reading with STM32F429 Microcontroller SPI by One DOUT Line

As shown in Figure 4, when FORMATx = 11 or 10, Channel 0 to Channel 7 output data on DOUT0 only. In standard mode operation, the AD7768/AD7768-4 operates as the master and stream data to the MCU, DSP, or FPGA. The AD7768/AD7768-4 supplies the data, the data clock (DCLK), and a falling edge framing signal (DRDY) to the slave device.

The STM32Fxxx family of microcontrollers are widely used in many different applications. The MCUs have several SPI ports, which can be configured as SPI master or slave with typical SPI timing modes. The methods introduced in the following session can also be applied on other microcontrollers with an 8-bit, a 16-bit, or a 32-bit frame.

The AD7768/AD7768-4 have 8-channels and 4-channels, simultaneous sampling sigma-delta (Σ-Δ) ADCs, respectively, with a sigma-delta modulator and digital filter per channel, enabling synchronized sampling of ac and dc signals. They achieve 108 dB dynamic range at a maximum input bandwidth of 110.8 kHz, combined with typical performance of ±2 ppm INL, ±50 µV offset error, and ±30 ppm gain error. The AD7768/AD7768-4 user can trade off input bandwidth, output data rate, and power dissipation, and select one of three power modes to optimize for noise targets and power consumption. The flexibility of the AD7768/AD7768-4 allows them to become reusable platforms for low power dc and high performance ac measurement modules. Unfortunately, AD7768's serial interface is not a typical SPI timing mode, and AD7768 works as the serial interface master. Generally, users must use FPGA/CPLD as its controller.

For example, 32F429IDISCOVERY and AD7768 eval boards are used. The workaround SPI wires are connected as shown in Figure 5. In this setup, all eight AD7768 channel data outputs on DOUT0 only.

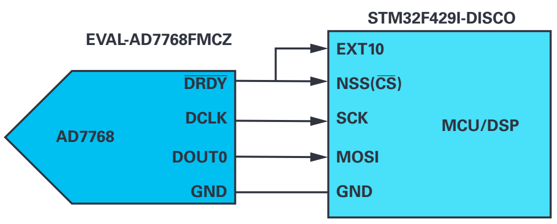


Figure 5. AD7768 outputs data on DOUT0 to an STM32F429 MCU SPI connection.

Problems to be solved:

- AD7768 works as the SPI master, so the STM32F429I SPI must be configured as SPI slave.
- $\overline{\text{DRDY}}$ high pulse is just one cycle of DCLK duration that is not a typical $\overline{\text{CS}}$.
- DCLK continuously outputs and $\overline{\text{DRDY}}$ is low when all the channel data bit output is finished.

Solution 1: MCU SPI as Slave Interfacing to SPI Master ADC by One DOUT Line

- Configure one of STM32F429 is SPI ports, like SPI4, as a slave to receive data bits on MOSI at DCLK.
- Connect AD7768 $\overline{\text{DRDY}}$ to the STM32F429 external interrupt input pin EXTI0 and NSS (SPI $\overline{\text{CS}}$) pin. The rising edge of $\overline{\text{DRDY}}$ will trigger EXTI0 handler routine to enable the SPI slave to start to receive data bits from the first DCLK falling edge after $\overline{\text{DRDY}}$ goes to low. Timing design is critical here.
- After all the data from Channel 0 to Channel 7 are received, the SPI should be disabled to prevent reading in extra invalid data, since the $\overline{\text{DRDY}}$ makes SPI slave $\overline{\text{CS}}$ low and DCLK keeps toggling.

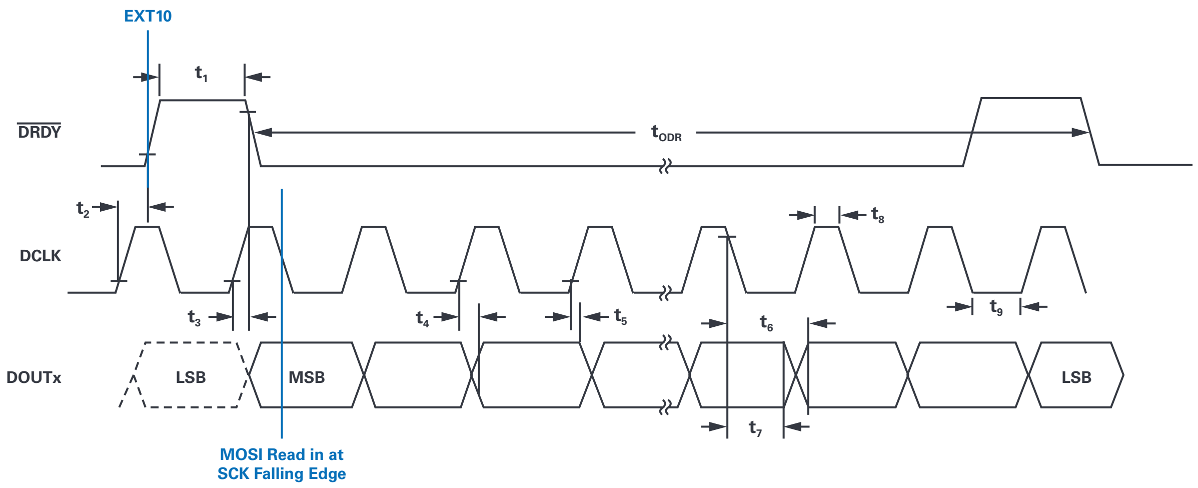


Figure 6. AD7768 data bits read in timing solution.

MCU Firmware Development Notes

With the software in interrupt mode, DCLK can run up to 4 MHz, and ODR 8 kSPS is achieved. The software should go into the interrupt handler to start SPI within one and a half DCLK period time (375 ns). To more easily enable the software to go into the interrupt routine, the MCU can read the data at the DCLK rising edge, which can give an additional half DCLK period time. But, since the t5 DCLK rise to the DOUTx invalid minimum is –3 ns (–4 ns for IOVDD = 1.8 V), a propagation delay (>|t5| + MCU hold time) on DOUTx should be added by PCB routing or buffer.

```
/*## Configure the SPI4 peripheral ###*/
Spi4Handle.Instance                          = SPI4;//use STM32F429 SPI4
Spi4Handle.Init.Direction                    = SPI_DIRECTION_2LINES_RXONLY;
Spi4Handle.Init.CLKPhase                     = SPI_PHASE_1EDGE;//read at DCLK falling edge
Spi4Handle.Init.CLKPolarity                  = SPI_POLARITY_HIGH;//read at DCLK falling edge
Spi4Handle.Init.DataSize                     = SPI_DATASIZE_8BIT;//or 16BIT
Spi4Handle.Init.NSS                          = SPI_NSS_HARD_INPUT;//make /CS low active
Spi4Handle.Init.Mode                         = SPI_MODE_SLAVE;//MCU SPI4 as SPI Slave
/*## Enable EXTI0 and SPI4 to Receive AD7768 Data bits ###*/
// clear EXTI0 IT flag prior to enable external interrupt 0 !!!
__HAL_GPIO_EXTI_CLEAR_IT(KEY_BUTTON_PIN);
HAL_NVIC_EnableIRQ(EXTI0_IRQn);
// wait for EXTI0 interrupt (/DRDY rising edge) to prepare for reading last conversion data
if (EXTI0_Flag == SET)
{
          EXTI0_Flag = RESET;//clear /DRDY rising edge flag variable
          // throw out the last byte/word captured in the previous ODR cycle !!!
          Rx_temp = *(__IO uint8_t *)&Spi4Handle.Instance->DR;
          __HAL_SPI_ENABLE(&Spi4Handle);
          // SPI4_CNVByteNum is the total data byte number to read in one conversion cycle
          while (SPI4_ByteCount < SPI4_CNVByteNum)
          {
                    // Check the RXNE flag
                    if (__HAL_SPI_GET_FLAG(&Spi4Handle, SPI_FLAG_RXNE))//
                    {
                              // transfer the received data from DR register to memory
                              SPI_RxBuffer[RxBuf_Idn] = *(__IO uint8_t *)&Spi4Handle.Instance->DR;
                              RxBuf_Idn++;
                              SPI4_ByteCount++;
                    }
          }
          // disable SPI4 to prevent read in extra data after all channel codes finished due to /DRDY
          is low active and DCLK continuously pulses
          __HAL_SPI_DISABLE(&Spi4Handle);
          SPI4_CNVCount++;
          RxBuf_Idn = SPI4_CNVCount * SPI4_CNVByteNum;
          SPI4_ByteCount = 0;
}//end of if (EXTI0_Flag == SET)
else
{//*** other software jobs ***//}
/*## handles External 0 interrupt request ###*/
// EXTI0 rising edge triggered to leave more response time for going into EXTI0_IRQHandler !!!
void EXTI0_IRQHandler(void)
{
          if(__HAL_GPIO_EXTI_GET_IT(EXTI0) != RESET)
          {
                    // enable SPI4 as soon as possible, and make sure before the first DCLK falling edge
                    after /DRDY falling !!!
                    __HAL_SPI_ENABLE(&Spi4Handle);
                    __HAL_GPIO_EXTI_CLEAR_IT(EXTI0);
                    EXTI0_Flag = SET;
          }
}
```

Figure 7. Configure the SPI4 peripheral.

Solution 2: MCU SPI as Slave Interfacing to SPI Master ADC by Two DOUT Lines

In the first solution, only DOUT0 is used to output all the 8-channel data. So, the data reading limits the ADC throughput rate to 8 kSPS. As shown in Figure 1, Channel 0 to Channel 3 output on DOUT0 and Channel 4 to Channel 7 output on DOUT1 can reduce the data transfer time. The serial wires are connected as shown in Figure 7. With such improvement, the ODR can easily go up to 16 kSPS at DCLK 4 MHz.

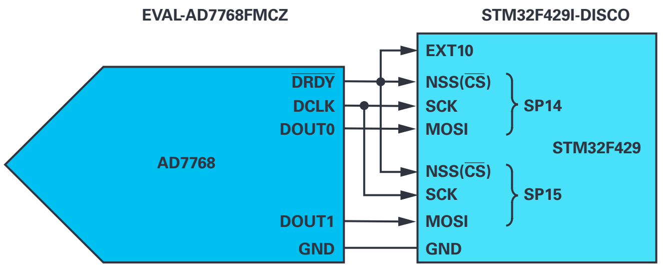


Figure 8. AD7768 output data on DOUT0 and DOUT1 to STM32F429 MCU SPI connection.

The firmware can use polling mode instead of the interrupt mode to reduce the time latency from the DRDY rising edge trigger to enable the SPI to receive the data. This can achieve ODR 32 kSPS at DCLK 8 MHz.

Solution 3: MCU SPI as Slave Interfacing to SPI Master ADC Through DMA

Direct memory access (DMA) is used in order to provide high speed data transfer between peripherals and memory, and between memory and memory. Data can be quickly moved by DMA without any MCU action. This keeps MCU resources free for other operations. Here are the design notes for an MCU SPI acting as slave to receive data through DMA.

Solution 4: MCU SPI as Master and Slave to Read Data on Two DOUT Lines

The high throughput or multichannel precision ADCs provide SPI ports with two, four, and even eight SDO lines for faster code reading time in serial mode. For microcontrollers with two or more SPI ports, they can concurrently run the SPI ports for faster code reading.

```
/*## EXTI0 in Polling Mode and SPI4 & SPI5 to Receive AD7768 Data bits on DOUT0 and DOUT1 ###*/
// Polling for EXTI0 (/DRDY) rising edge to start MCU SPI ports
while (__HAL_GPIO_EXTI_GET_IT(EXTI0) != SET);
{
          __HAL_SPI_ENABLE(&Spi4Handle);
          __HAL_SPI_ENABLE(&Spi5Handle);
          __HAL_GPIO_EXTI_CLEAR_IT(EXTI0);
}
// throw out the last byte/word captured in the previous ODR cycle !!!
Rx_temp = *(__IO uint8_t *)&Spi4Handle.Instance->DR;
Rx_temp = *(__IO uint8_t *)&Spi5Handle.Instance->DR;
while (SPI4_ByteCount < SPI4_CNVByteNum)// total data byte number to read in one conversion cycle
{
          if (__HAL_SPI_GET_FLAG(&Spi5Handle, SPI_FLAG_RXNE))//
          {
                    SPI_RxBuffer[RxBuf_Idn] = *(__IO uint8_t *)&Spi4Handle.Instance->DR;
                    SPI_RxBuffer[RxBuf_Idn+1] = *(__IO uint8_t *)&Spi5Handle.Instance->DR;
                    RxBuf_Idn++;
                    SPI4_ByteCount += 2;
          }
}
__HAL_SPI_DISABLE(&Spi4Handle);
__HAL_SPI_DISABLE(&Spi5Handle);
```

Figure 9. EXTI0 in polling mode and SPI4 and SPI5 to receive AD7768 data bits on DOUT0 and DOUT1.

```
/*## EXTI0 in Polling Mode and SPI4 DMA to Receive AD7768 Data bits on DOUT0 ###*/
// Polling for EXTI0 (/DRDY) rising edge to start MCU SPI ports
while (EXTI0_Flag != SET);// wait for EXTI0 interrupt (/DRDY rising edge)
EXTI0_Flag = RESET;// clear flag variable
// throw out the last byte/word captured in the previous ODR cycle !!!
Rx_temp = *(__IO uint8_t *)&Spi4Handle.Instance->DR;
Spi4Handle.hdmarx->Instance->NDTR = SPI4_CNVByteNum;// set data number to read
Spi4Handle.hdmarx->Instance->PAR = (uint32_t)&(Spi4Handle.Instance->DR);// source address
Spi4Handle.hdmarx->Instance->M0AR = (uint32_t)(SPI_RxBuffer+RxBuf_Idn); // target address
//*** clear event flags corresponding to the stream in DMA_LISR or DMA_HISR register ***//
((DMA_Base_Registers *)Spi4Handle.hdmarx->StreamBaseAddress)->IFCR = 0x3FU << Spi4Handle.hdmarx->StreamIndex;
__HAL_DMA_ENABLE(Spi4Handle.hdmarx);
while ((Spi4Handle.hdmarx->Instance->CR & DMA_SxCR_EN) == SET) // hardware cleared
{;} // ADC data received in the target memory buffer
SPI4_CNVCount++;
RxBuf_Idn = SPI4_CNVCount * SPI4_CNVByteNum;
```

Figure 10. EXTI0 in polling mode and SPI4 DMA to receive AD7768 data bits on DOUT0.

In the following use case, 32F429IDISCOVERY uses SPI4 as SPI master and SPI5 as SPI slave to receive EVAL-AD7606B-FMCZ data on DOUTA and DOUTB as shown in Figure 8.

The AD7606B is a 16-bit, simultaneous sampling, analog-to-digital data acquisition system (DAS) with eight channels, each channel containing analog input clamp protection, a programmable gain amplifier (PGA), a low-pass filter, and a 16-bit successive approximation register (SAR) ADC. The AD7606B also contains a flexible digital filter, low drift, 2.5 V precision reference and reference buffer to drive the ADC and flexible parallel and serial interfaces. The AD7606B operates from a single 5 V supply and accommodates ±10 V, ±5 V, and ±2.5 V true bipolar input ranges when sampling at throughput rates of 800 kSPS for all channels.

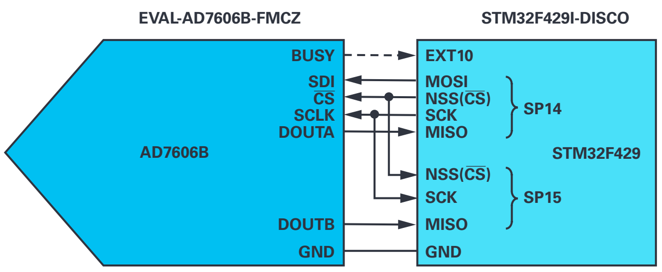
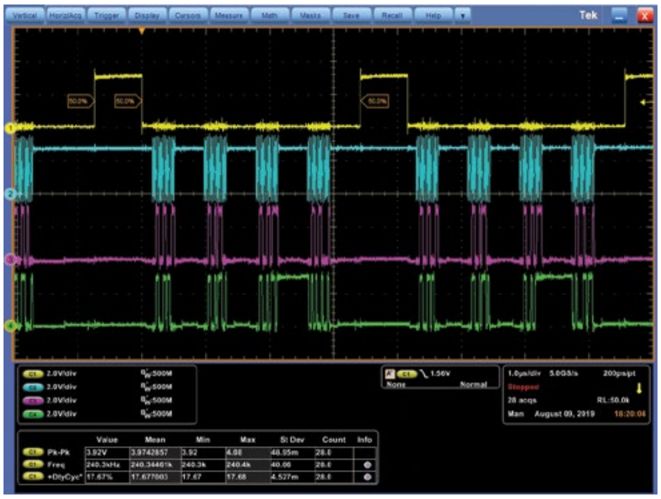


Figure 11. MCU SPIs used in master and slave mode to receive data on DOUTA and DOUTB.

```
/*## Configure the SPI4 as Master and SPI5 as Slave ###*/
Spi4Handle.Init.Direction      = SPI_DIRECTION_2LINES;
Spi4Handle.Init.CLKPhase       = SPI_PHASE_1EDGE;//read at DCLK falling edge
Spi4Handle.Init.CLKPolarity    = SPI_POLARITY_HIGH;//read at DCLK falling edge
Spi4Handle.Init.DataSize       = SPI_DATASIZE_16BIT;
Spi4Handle.Init.NSS            = SPI_NSS_SOFT;// NSS pin is configured as GPIO output for /CS
Spi4Handle.Init.Mode           = SPI_MODE_MASTER;// SPI4 as SPI Master
Spi5Handle.Init.Direction      = SPI_DIRECTION_2LINES_RXONLY;// only receive data
Spi5Handle.Init.NSS            = SPI_NSS_HARD_INPUT;
Spi5Handle.Init.Mode           = SPI_MODE_SLAVE;// SPI5 as SPI Slave
/*## Enable SPI4 as Master and SPI5 as Slave to Receive AD7606B Codes ###*/
__HAL_SPI_ENABLE(&Spi4Handle);
__HAL_SPI_ENABLE(&Spi5Handle);
while (SPI4_CNVCount < SPI4_CNVNum)
{
          CLR_CNV();
          SET_CNV();//AD7606B conversion start
          // wait for conversion finish, BUSY goes from high to low. Polling or interrupt mode
          while (BUSY == SET) {;}
          while (SPI4_WordCount < SPI4_CNVWordNum)// code number to read per conversion cycle
          {
                    CLR_CS();
                    *(__IO uint8_t *)&Spi4Handle.Instance->DR = 0;
                    while (__HAL_SPI_GET_FLAG(&Spi4Handle, SPI_FLAG_RXNE) != SET);
                    Delay_xus(1);// need half SCLK cycle delay for slow SCLK rate < 10MHz
                    SET_CS();
                    SPI_RxBuffer[RxBuf_Idn] = *(__IO uint16_t *)&Spi4Handle.Instance->DR;
                    SPI_RxBuffer[RxBuf_Idn+ADCSDO1_WordIdn] = *(__IO uint16_t \\
                    *)&Spi5Handle.Instance->DR;
                    RxBuf_Idn++;
                    SPI4_WordCount += 2;
          }
          SPI4_CNVCount++;
          RxBuf_Idn = SPI4_CNVCount * SPI4_CNVWordNum;
          SPI4_WordCount = 0;
}//while (SPI4_CNVCount < SPI4_CNVNum)
__HAL_SPI_DISABLE(&Spi4Handle);
__HAL_SPI_DISABLE(&Spi5Handle);
```

Figure 12. Configure the SPI4 as master and SPI5 as slave.

Figure 13 shows the AD7606B digital interface capture of BUSY, SCLK, DOUTA, and DOUB running at 240 kSPS.



	Value	Mean	Min	Max	St Dev	Count	Info
C1 Pk-Pk	3.92V	3.9742857	3.92	4.08	48.95m	28.0	
C1 Freq	240.3kHz	240.34461k	240.3k	240.4k	40.06	28.0	●
C1 +DtyCyc*	17.67%	17.677003	17.67	17.68	4.527m	28.0	●

Figure 13. Scope capture of AD7606B BUSY, SCLK, and data on DOUTA and DOUTB.

Conclusion

This article discussed approaches to using a microcontroller SPI to access ADCs with nonstandard SPI interfaces. These approaches can be used directly or with slight adjustments to control the ADC SPI, which is working as an SPI master or with multiple DOUT lines for a faster throughput rate.

Acknowledgements

Many thanks to applications engineers Mika Jiang and Yao Zhao for advice on the quick startup on the STM32F429IDISCOVERY kit and firmware debug jobs.

References

Dhaker, Piyu. "Introduction to SPI Interface." Analog Dialogue, Vol 52. September 2018.

RM0090 Reference Manual: STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 Advanced ARM®-Based 32-Bit MCUs. STMicroelectronics, February 2019.

STM32F427xx Data Sheet. STMicroelectronics, January 2018.

UM1670 User Manual: Discovery Kit with STM32F429ZI MCU. STMicroelectronics, September 2017.

Usach, Miguel. AN-1248 Application Note: SPI Interface. Analog Devices, Inc., September 2015.

About the Author

Steven Xie has worked as a product applications engineer with the China Design Center in ADI Beijing since March 2011. He provides technical support for SAR ADC products across China. Prior to that, he worked as a hardware designer in wireless communication base stations for four years. In 2007, Steven graduated from Beihang University with a master's degree in communications and information systems. He can be reached at *steven.xie@analog.com*.