Vol 57, No 1—March 2023

ANALOG DEVICES
AHEAD OF WHAT'S POSSIBLE™

AnalogDialogue

How to Optimize the MCU SPI Driver to Reach High Speed ADC Throughput

Denny Wang, Applications Engineer, and
Sally Tseng, Applications Engineer

Abstract

With the advancement of technology, the transmission of more precise data is needed in low power Internet of Things (IoT) and edge/cloud computing. In Figure 1, the wireless sensing system is a high precision data acquisition system with a 24-bit analog-to-digital converter (ADC). In this case, whether the microcontroller unit (MCU) can afford the serial high speed interface of the data converter is our problem.

This article describes the process of designing a high speed Serial Peripheral Interface (SPI) data transaction driver between the MCU and ADC. The following sections will show a brief description of different ways to optimize the SPI driver and the required configuration on the ADC and MCU. A detailed description of the example code of the SPI and direct memory access (DMA) data transaction will be provided after the brief illustration. Finally, a demonstration of the throughput of ADC with the same driver in a different MCU (ADuCM4050, MAX32660) will be included.

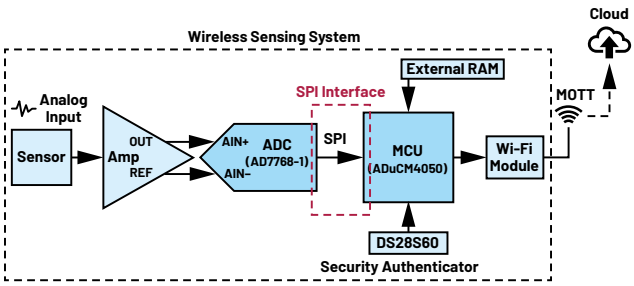

Figure 1. Condition-based monitoring.

Introduction

Introduction to General SPI Driver

As the name suggests, MCU vendors provide general SPI driver/API in example code to each MCU. The general SPI driver/API usually can cover most users' applications that may conclude many configurations or determine statements. In some specific scenarios, such as ADC data acquisition, the general SPI driver cannot satisfy the full speed throughput of the ADC data because too many different configurations in the general driver are implemented. The configurations that are not used in the application create additional overhead and cause a time delay.

```
void adi_spi_MasterSubmitBuffer (struct hDevice, struct *buffer)
{
    if(…)
        *p = xxx;
    else if(…)
        *p = xxx << xxx;
        for (…)

    :

    start_spi_transaction();
}
```

Config.

Figure 2. Configuration of a general API.

Framework of the Thoughts and Practice

To extract the output data of an ADC through SPI, we would choose an MCU as the main device for its low power consumption and high speed performance. However, when a data transaction is based on the ADI SPI driver, the speed can degrade due to commands that are not functional in the ADC-to-MCU application. To fully unleash the potential speed of the ADC, we experiment with the ADuCM4050 and AD7768-1 and try out possible solutions. Despite the maximum output data rate of 256 kHz (under the default filter), the ADuCM4050 is currently limited to 8 kHz. Potential solutions for the acceleration of the output rate include removing unnecessary commands and activation of the DMA controller. These ideas are realized in the following sections.

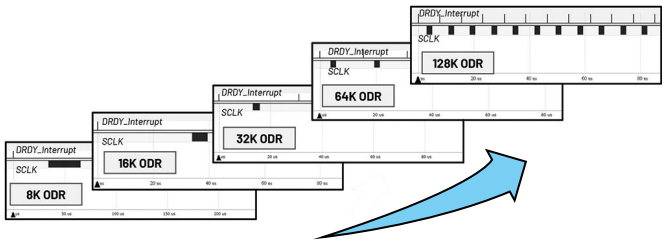

Figure 3. Different ODRs and the relationship between DRDY and SCLK.

MCU as Main

The ADuCM4050 MCU is an ultra low power microcontroller system with a 26 MHz main clock rate. The system is driven by the Arm® Cortex®-M4F processor. The ADuCM4050 is equipped with three SPIs, and each SPI incorporates two DMA channels, receive and transmit, that interface with the DMA controller. The DMA controller and DMA channels provide a means to transfer data between memory and peripherals. This is an efficient way to distribute data freeing up the core to handle other tasks.

ADC as Node

The AD7768-1 is a 24-bit, low power, high performance, sigma-delta (Σ-Δ) ADC. The output data rate (ODR) and power dissipation mode are adjustable to meet the users' requirements. The decimation factor and power mode together determine the ODR as shown in Table 1.

Table 1. Power Mode Configuration for Output Data Rate

Power Mode	Decimation	ODR
Fast (MCLK/2)	×32	256 kHz
Fast (MCLK/2)	×64	128 kHz
Median (MCLK/4)	×32	128 kHz
Median (MCLK/4)	×64	64 kHz
Low Power (MCLK/16)	×32	32 kHz
Low Power (MCLK/16)	×64	16 kHz

The AD7768-1's continuous read mode is also a significant feature. The output data of the ADC is stored in the register 0x6C. In general, the data in ADC register requires address specification before each read/write operation. Continuous read mode enables direct derivation of data from the 0x6C register after each data ready signal. The ADC output data is a 24-bit digital signal that converts to volts as shown in Table 2.

Table 2. Output Codes and Ideal Input Voltage

Description	Analog Input	Digital Output Code
+Full Swing −1 LSB	+4.095999512 V	0x7FFFFF
Midscale +1 LSB	+488 nV	0x000001
Midscale	0 V	0x000000
Midscale −1 LSB	−488 nV	0xFFFFFF
−Full Swing −1 LSB	−4.095999512 V	0x800001
−Full Swing	+4.096 V	0x800000

Connection Diagram of Data Flow

The ADuCM4050 and AD7768-1 are used as the data transaction example models. The pin connection is shown in Figure 4.

Their reset signal is sent from the MCU GPIO28 to the ADC RST_1 pin, and the data ready signal is sent from the ADC DRDY_1 to the MCU GPIO27. The rest of the pins are connected as a general SPI configuration, where the MCU is the main and ADC is the node. SDI_1 receives the ADC register read/write commands from the MCU, and DOUT_1 sends the output data to the MCU.

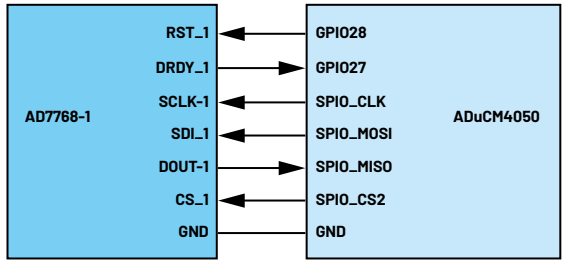

Figure 4. An interface pin connection for the AD7768-1 and the ADuCM4050.

Realization of Data Transaction

Interrupt Data Transaction

To realize the continuous data transaction, we use the GPIO27 (connected to DRDY) as the interrupt trigger. When the ADC sends a data ready signal to the GPIO27, the MCU runs the callback function, where data transaction commands are included. As shown in Figure 5, data acquisition must be handled in the interval between Interrupt A and Interrupt B.

With the ADI SPI driver, we can easily realize the data transaction between the ADC and MCU. However, the ADC ODR would be limited to 8 kHz due to redundant commands in the driver. To speed up the process, we trim the codes to the slimmest. We introduce two methods for DMA data transactions: basic mode DMA transaction and ping-pong mode DMA transaction.

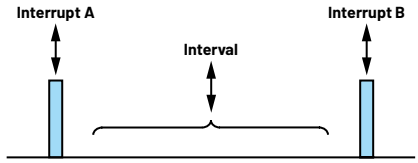

Figure 5. An interval between two interrupts.

Basic Mode DMA Transaction

Before each DMA transaction, some SPI and DMA settings are required (see the example code in Figure 6). `SPI_CTL` is the SPI configuration. `SPI_CTL=0x280f` is derived from the value set for the ADI SPI driver. `SPI_CNT` is the transfer byte count. Since each DMA transaction can only transmit a fixed bit number of 16, `SPI_CNT` must be a multiple of 2. In our case, `SPI_CNT=4` is chosen to cover the 24-bit output data of ADC. `SPI_DMA` register is the SPI DMA enable. `SPI_DMA=0x5` enables the receive DMA request. `pADI_DMA0->EN_SET=(1<<5)` enables the DMA of the fifth DMA channel - SPI0 RX.

Table 3. DMA Structure Register

Name	Description
SRC_END_PTR	Source end pointer
DST_END_PTR	Destination end pointer
CHNL_CFG	Control data configuration

Each DMA channel has a DMA structure register shown in Table 3. Note that the end of the source address here (that is, SPI0 Rx) needs no increments during the entire operation since the Rx FIFO pushes the data out of the register automatically. On the other hand, the end of the destination address is calculated by the function (destination address + `SPI_CNT` -2) according to the ADI SPI driver.

The current address here is the address of an internal array buffer. DMA control data configuration includes the settings of source data size, source address increment, destination address increment, the number of transfers left, and DMA control mode. The value **0x4D000011** sets the configuration as described in Table 4.

Table 4. DMA Configuration for the Control Data Configuration 0x4D00011

Register	Description	Value
DST_INC	Destination address increment	2 bytes
SRC_INC	Source address increment	0
SRC_SIZE	Source address increment	2 bytes
N_minus_1	Total number of transfers in the current DMS cycle – 1	1 (N = 2)
Cycle_ctrl	The operating mode of the DMA cycle	Basic mode

With the dummy read command **SPI_SPIO -> RX,** SCLK clocking starts, and the output data is transmitted from ADC to MCU through the MISO line. There are also some negligible data transfers on the MOSI line. Once the Rx FIFO is filled, the DMA request is generated, thus activating the DMA controller to transfer the data from the DMA source (that is, SPIO Rx FIFO) to the DMA destination (that is, the internal array buffer). It is worthy of note that the Tc request is generated at **SPI_DMA=0x3**.

At last, we maintain the destination address for the next 4-byte transmission by adding 4 to the current destination address.

Please also be aware that both **pADI_DMA0->DSTADDR_CLR** and **pADI_DMA0->RMSK_CLR** for SPIO DMA channel must be set in the main function before the first interrupt occurs. The prior register is DMA Channel Destination Address Decrement Enable Clear, which sets the destination address shifting after each DMA transfer in increment mode (only in increment mode does the function for destination address calculation works). The latter register is DMA Channel Request Mask Clear, which clears the DMA request status for the channel.

A time diagram for the basic mode DMA transaction is depicted in Figure 7a. The time slots represent the DRDY signal, SPI/DMA settings, and DMA data transaction, respectively. To better utilize the idle time of the CPU, we wish to assign tasks for the CPU while the DMA controller is handling the data transfer.

```
//SPI settings
pADI_SPIO->CTL = 0x280f;
pADI_SPIO->CNT = (uint16_t)4;
pADI_SPIO->DMA = 0x5;

//DMA settings
pADI_DMAO->EN_SET = (1<<5);
pPrimaryCCD[5].DMASRCEND = (uint32_t)&pADI_SPIO->RX;
pPrimaryCCD[5].DMADSTEND(uint32_t)Current_address;
pPrimaryCCD[5].DMACDC = 0x4D000011;

//Dummy read
pADI_SPIO->RX;

//Data destination address maintenance
Current address = Current_address + 4;
```

Figure 6. A code of the basic DMA transaction mode.

Ping-Pong Mode DMA Transaction

After the dummy read command is implemented, the DMA controller starts the data transaction, leaving the CPU of the MCU idle with no tasks. If we can have the CPU and DMA controller work simultaneously, the task handling is no longer serial but parallel. Therefore, we can conduct the DMA configuration (by the CPU) and the DMA data transaction (by the DMA controller) at the same time. To realize this idea, the so-called ping-pong mode is required for the DMA controller. Ping-pong mode incorporates two sets of DMA structures—primary and alternate. The DMA controller switches automatically between the two structures at each DMA request. Initially set to 0, variable p marks if the primary (p = 0) or alternate (p = 1) DMA structure is in charge. If p = 0, DMA primary data transaction starts on the dummy read command. At the same time, we assign values for the alternate DMA structure, which will be in charge in the next interrupt cycle. If p = 1, the primary and alternate structures switch their roles. The modification of the DMA structure during a DMA transaction can fail with only the primary one in basic DMA mode. The use of ping-pong mode enables the CPU to access and write the alternate DMA structure while the primary is read by the DMA controller and vice versa. As shown in Figure 7b, the DMA data transaction can be performed soon after the DRDY signal got transmitted from the ADC to the MCU since the DMA structure configuration is done in the last cycle. The CPU and DMA are now working simultaneously without one waiting for the other. We expect that there is now room for the ADC ODR up-tuning because the total operation time is significantly shortened.

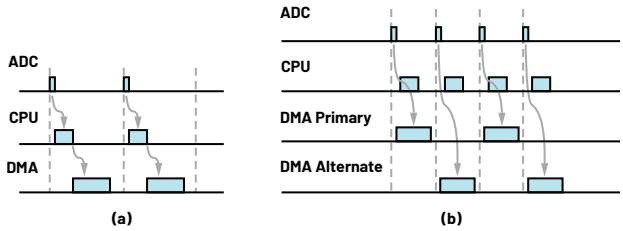

Figure 7. A time diagram for (a) basic DMA mode and (b) ping-pong mode.

Interrupt Handler Optimization

The time interval between the data ready signal does not only include the conduct time of the commands in the callback function but also the ones in the ADI GPIO interrupt handler.

At the activation of the MCU, the CPU runs the start-up file (that is, startup.s). All the event handlers are defined in the file, including the GPIO interrupt handler. Once the GPIO interrupt is triggered, the interrupt handler function (that is, GPIO_A_INT_HANDLER and GPIO_B_INT_HANDLER in the ADI GPIO driver) is performed. In general interrupt handler functions, the CPU searches through all GPIO pins for the one that triggers, clears its interrupt status, and runs the registered callback function. In the ADC-MCU application used, DRDY is the only interrupt signal. We would thus like to trim the function to speed up the process. Possible solutions include (1) retargeting in the start-up file and (2) the modification of the original interrupt handler. Retargeting means to self-define an interrupt handler and to replace the original one in the start-up file.

Modification, on the other hand, requires a self-defined GPIO driver. We take the latter option and modify the function as in Figure 8, where it clears only the interrupt status of the pin connected to the DRDY and goes directly to the callback function. Please note that the original GPIO driver needs to be blocked by unchecking the box of include in build target.

```
DCD   GPIO_A_Int_Handler_Denny   ; GPIO Int A          /* 9  */
DCD   GPIO_B_Int_Handler         ; GPIO Int B          /* 10 */
DCD   GPIO_Tmr0_Int_Handler      ; GP Timer 0          /* 11 */
DCD   GPIO_Tmr1_Int_Handler      ; GP Timer 1          /* 12 */
```

Figure 8. A nested vectored interrupt controller (NVIC).

Result

Speed Performance

Assume the user is now reading the 200 24-bit ADC output data. The SPI bit rate is set at 13 MHz. We connect the pin with the DRDY signal and SCLK to the oscilloscope. By observing the time interval between the DRDY signal and the start of the SPI data transaction (also DMA transaction), we can quantize the speed improvement in each method introduced in this article. For convenience, we refer to the time interval from the DRDY signal to the start of the SCLK signal as Δt. For the 13 MHz SPI bitrate, the measured Δt are:

- (a) Basic mode DMA Δt = 3.754 µs
- (b) Ping-pong mode DMA Δt = 2.8433 µs
- (c) Ping-pong mode DMA with optimized interrupt handler Δt = 1.694 µs

Methods (a) and (b) can support an ODR of 64 kHz while Method (c) can support an ODR of 128 kHz. This is because Δt for Method (c) is the shortest, enabling SCLK to end earlier. If the SCLK signal—that is, data transaction—is done before the T/2 (T is the current ADC output data period), multiplication of the ODR will be available. Compared to the 8 kHz ODR speed performance of the ADI SPI driver, this can be seen as tremendous progress.

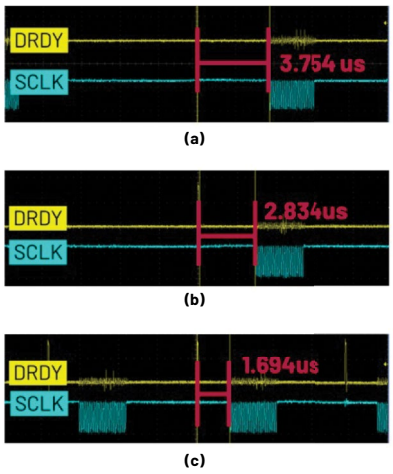

Figure 9. Δt for (a) basic mode DMA, (b) ping-pong mode, and (c) ping-pong mode with an optimized interrupt handler.

MAX32660 with the AD7768-1

What is the result when using the MAX32660, which is an MCU with 96 MHz of main clock rate? In this case, we used an interrupt data transaction with interrupt handler optimization. With this interrupt setup, a 256 kHz output data rate can be achieved without the DMA function. See Figure 10.

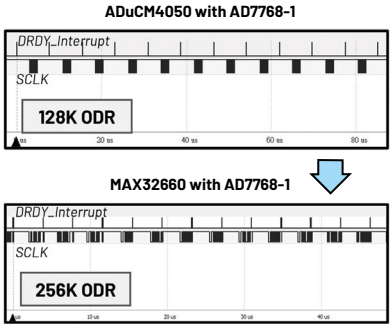

Figure 10. ODR print at MAX32660 without DMA.

Conclusion

With the selected ADC (AD7768-1) and MCU (ADuCM4050 or MAX32660), we can realize a high speed data transaction through SPI. To achieve the goal of speed optimization, we conduct the data transaction according to the ADI SPI driver but deduct the redundant commands. Also, the activation of the DMA controller frees the core and speeds up the continuous data transaction. With DMA ping-pong mode, DMA configuration time is saved with a proper schedule. On top of acceleration through DMA, the interrupt handler can also be optimized by direct specification of the interrupted pin. The best performance reaches a 128 kSPS ADC ODR at the 13 MHz SPI bit rate.

Table 5. High Speed SPI Connection Realized with the ADuCM405 and the MAX32660

	ADuCM4050 (MCU)				MAX32660 (MCU)
Data Transaction	Interrupt Without Optimization	Basic Mode DMA	Ping-Pong Mode DMA	Interrupt with Optimization	Interrupt with Optimization
Bus Type	SPI	SPI	SPI	SPI	SPI
Main Clock Rate	26 MHz	26 MHz	26 MHz	26 MHz	96 MHz
SPI Clock Rate	13 MHz	13 MHz	13 MHz	13 MHz	20 MHz
Interval Between DRDY and SCLK	6.34 µs	3.754 µs	2.834 µs	1.694 µs	1.464 µs
Output Data Rate	8 kSPS	32 kSPS	64 kSPS	128 kSPS	256 kSPS

Acknowledgments

Throughout the writing of this dissertation, we have received a great deal of support and assistance.

We would like to thank Charles Lee, whose expertise was invaluable in hardware experience, software support, and even debugging tips.

We would also like to thank our mentor, William Chen, for his guidance in technical support.

Finally, we want to thank Frank Chang, who shared with us much of the technical experience from his career.

About the Author

Denny Wang is an applications engineer who has spent 1.5 years under the NCG program in Analog Devices Taiwan. He holds a master's degree in integrated circuit engineering from Peking University. During graduate school, he engaged in semiconductor processing, BLDC/PMSM motor control, and analog schematic migration. He joined ADI in 2021 and is currently supporting system solutions, data security and authenticator, and MQTT wireless transmission.

About the Author

Sally Tseng is an applications engineer intern at ADI Taiwan. She's a senior student at National Taiwan University, majoring in electrical engineering. Her projects at NTU are mainly about analog circuit research. She is also involved in embedded systems during the duration of her internship at ADI.