Application Note

Implementing Host Controller for TMP1826 and TMP1827 Single-Wire Temperature Sensor

Amit Ashara *Temperature and Humidity Sensing*

ABSTRACT

The TMP1826 and TMP1827 devices are part of the TI portfolio of high-accuracy, single-wire compatible digital output temperature sensors. The TMP182x devices simplify applications by providing a true single-wire bus-powered mode of operation, with multi-drop capability, thereby reducing the requirement for bulky cables and complex routing on space-constrained PCBs. With the dual-bus speed mode and seamless switching between the modes, the single-wire is designed for both short- and long-distance applications, enabling customers to scale their applications with minimal software updates. Because the interface is different from traditional inter-integrated circuit (I^2C), universal asynchronous receiver and transmitter (UART), and serial-peripheral interfaces (SPI), this application note provides both design and software references, so that customers can implement the bus protocol on their choice of MCUs using UART, SPI, and general-purpose input/output (GPIO) peripherals. The examples provided show the TMP1826 being interfaced via different interfaces, but the same is applicable to TMP1827 devices as well.

Table of Contents

List of Figures

List of Tables

Trademarks

1 Introduction

The single-wire interface for the TMP1826, does not have a reference clock. Therefore all communication is performed asynchronously with variable pulse widths to indicate different operations. Figure 1-1 shows that the bus consists of a single pullup resistor for all devices on the bus. The devices can be powered by the supply, where the V_{DD} pin is connected to the same supply as the host MCU and pullup resistor or bus powered, where the V_{DD} pin is connected to GND and the device derives power from the pullup resistor.

After power up, the external pullup resistor holds the line high which is referred to as the idle state. Almost all communication is initiated by the host by driving the data line low to generate a falling edge. Based on the duration of the low period, the device interprets the data bit as a reset request, logic '0' or logic '1'.

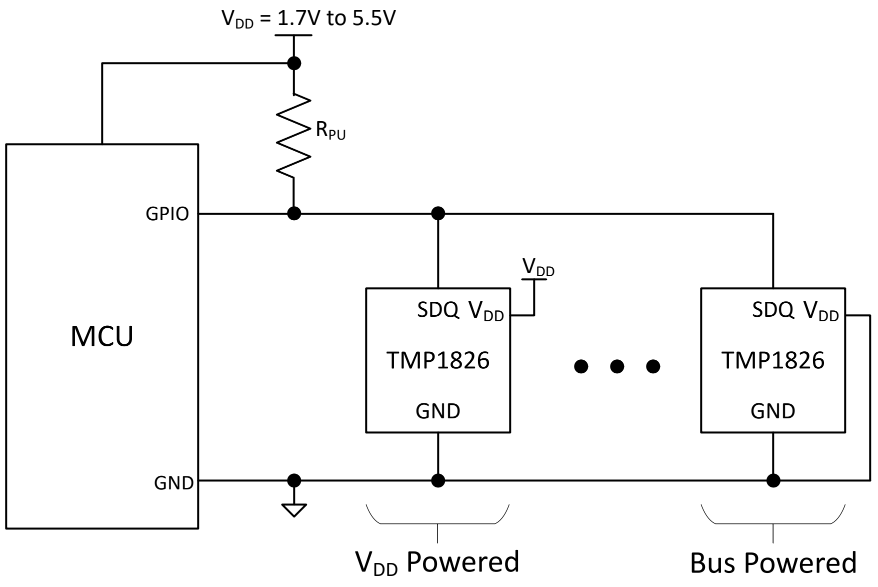

Figure 1-1. Simplified System Block Diagram

1.1 Bus Reset and Response

All communication to the TMP1826 on the single-wire begins with the bus reset and response phase. The phase is initiated by the host by holding the single-wire data line low for a period of t_{RSTL}. All devices on the bus, irrespective of their current state respond to the bus reset, by reinitializing their internal state and responding to the host-initiated bus reset. The devices respond after a minimum of t_{PDH}, by holding the single-wire low for a time period of t_{RSTH} as shown in Figure 1-2.

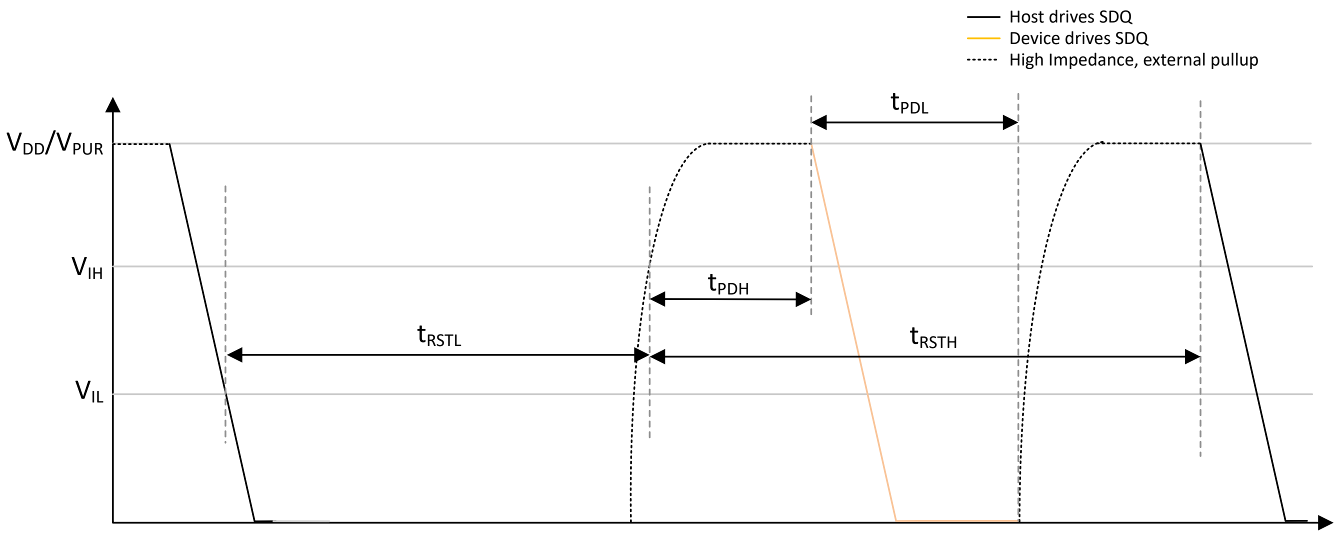

Figure 1-2. Bus Reset and Response

1.2 Host Write, Device Read

A host write is the means by which the host sends the command, function, and data to the devices. A host write starts by the host driving the data line low as shown in Figure 1-3. If the host intends to transmit a logic '1', the line is released after t_{WR1L} time. If the host intends to transmit a logic '0', the line is released after t_{WR0L}. After releasing the data, the pullup resistor causes the line to become high till the beginning of the next time slot. The device samples the line after t_{RDV} has elapsed from the falling edge, for a time frame indicated by t_{DSW}. The host must factor the rise time due to the pullup resistor and bus capacitance to determine the release of the data line before the line is sampled by the device and the host drives the next write bit time slot.

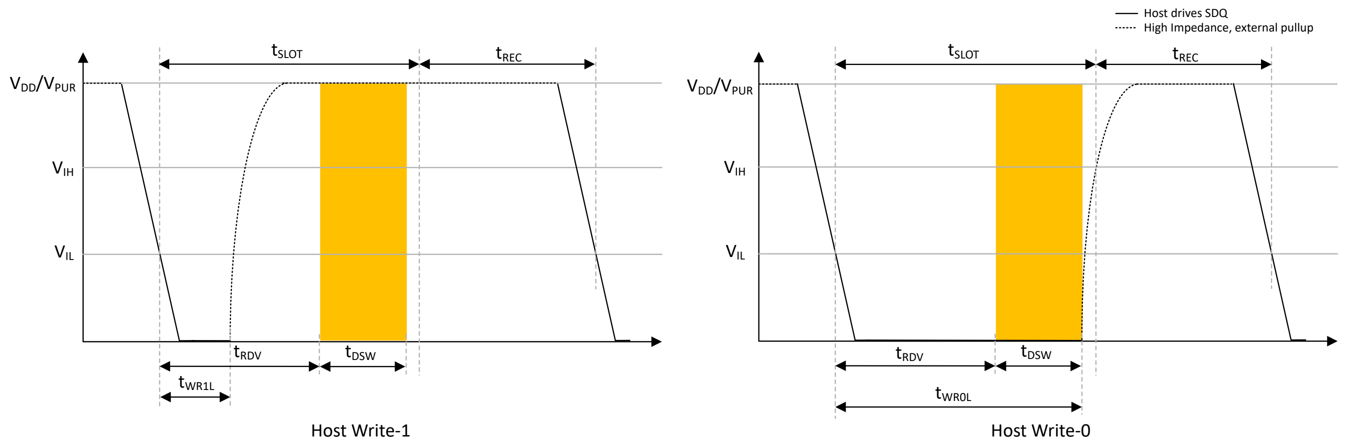

Figure 1-3. Host Write, Device Read

1.3 Host Read, Device Write

A host read is the means by which the hosts gets the data from the device or the CRC for data integrity check. A host read starts by the host driving the data line low as shown in Figure 1-4. When the device detects the falling edge, the device can drive the line low before the time t_{RL}. The host can release the bus from the side after the time $t_{RL(MIN)}$ elapses. If the device intends to transmit a logic '1', then the bus is released before $t_{RL(MAX)}$ elapses. If the device intends to transmit a logic '0', then the bus is released after $t_{SLOT(MIN)}$. The host must sample the line after the time t_{RWAIT}, for a time frame indicated by t_{MSW}. The host must factor the rise time due to the pullup resistor and bus capacitance to determine the sampling window for the host to sample the bit level sent by the device or to drive the next read bit time slot.

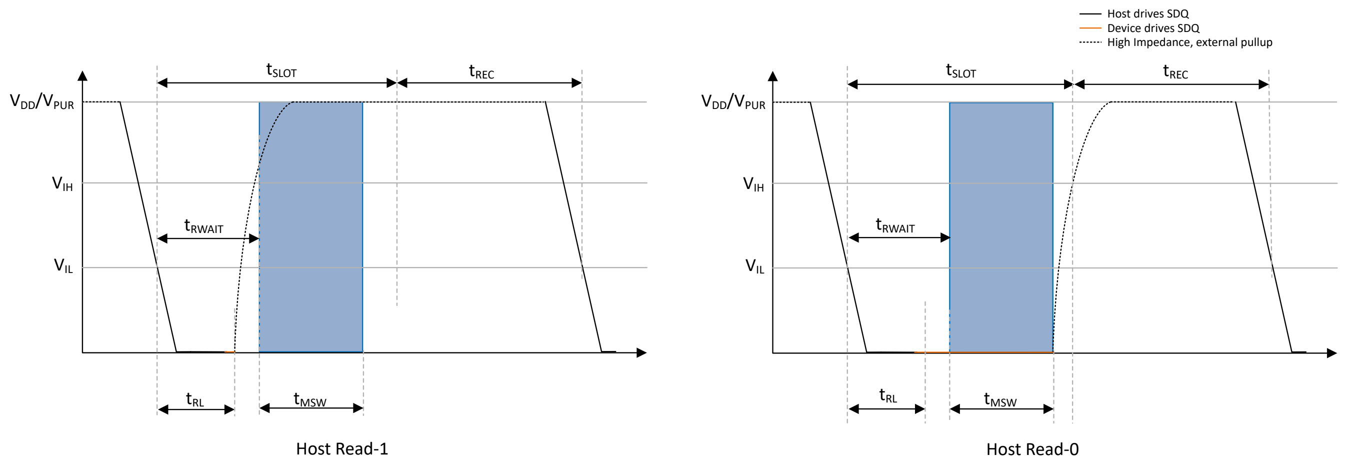

Figure 1-4. Host Read, Device Write

2 Interfacing TMP1826 With the Host MCU

The previous section provided a basic understanding of how the single-wire communication is performed using the different low times to represent reset, logic '0' and logic '1'. This section provides mechanisms by which different peripherals like general-purpose input and output (GPIO), UART, or SPI can be used by a host MCU to communicate with the TMP1826 for configuration, or getting temperature conversion results, or both.

The default overdrive speed (90 kbps) is used for communication with the devices to simplify the interface and software requirements,.

Note

All subsequent example waveforms were generated using respective interfaces on TI's TM4C123x and TM4C129x family of microcontrollers.

2.1 Using GPIO as Host Interface

All MCUs provide software configurable GPIO for input and output functions. Figure 2-1 shows an example of an MCU which supports a bidirectional open-drain pin. The example illustrated is perhaps the simplest method to interface the TMP1826 to an MCU since only a bus pullup resistor is required.

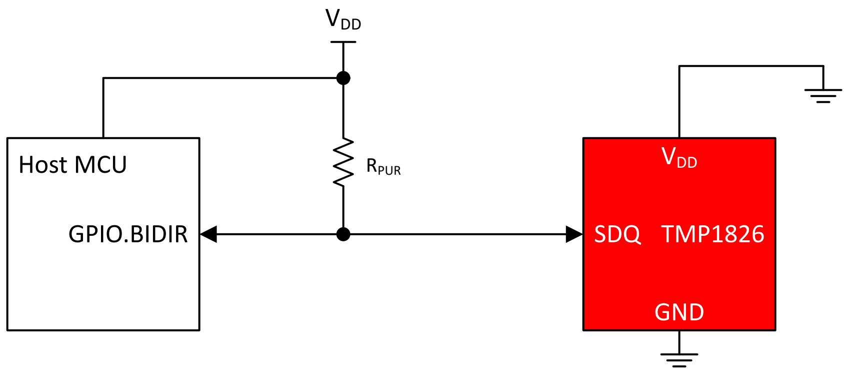

Figure 2-1. Bidirectional GPIO Interface for TMP1826

Figure 2-2 shows an example of an MCU which supports GPIO with only an input or output function. In such an implementation, a bus pulldown transistor or a push pull to open-drain driver like the SN74LVC1G07 device is required along with the bus pullup resistor.

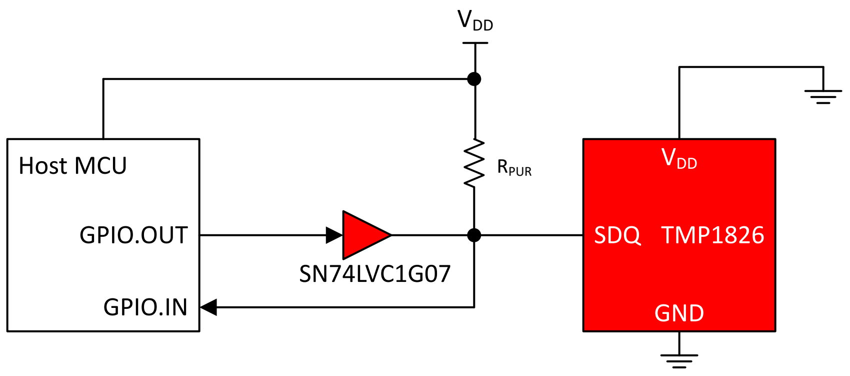

Figure 2-2. Push-Pull GPIO Interface for TMP1826

2.2 Software Driver for GPIO

The application example demonstrates a simple operation of the host sending a command to read the temperature from the TMP1826 device. As the application example shows, the basic units of bus communication are *mcu_txOneWireReset*, *mcu_txOneWireByte*, and *mcu_rxOneWireByte*.

```
// Send Bus Reset
mcu_txOneWireReset();
// Send SKIP ADDRESS command
mcu_txOneWireByte(0xCC);
// Send TEMP CONVERT function
mcu_txOneWireByte(0x44);
// Wait for tCONV = 5.5 ms
DelayMS(6);
// Send Bus Reset
mcu_txOneWireReset();
// Send SKIP ADDRESS command
mcu_txOneWireByte(0xCC);
// Send READ SCRATCHPAD-1 function
mcu_txOneWireByte(0xBE);
// Read two bytes of temperature data
TempLSB = mcu_rxOneWireByte();
TempMSB = mcu_rxOneWireByte();
```

When using GPIO for communicating with the TMP1826, the host MCU requires a delay counter to implement the communication. The following pseudocode shows how to implement bus reset and response using GPIO with the input or output function and a delay counter.

```
void mcu_txOneWireReset()
{
    // Send Bus Reset
    GPIOWrite(GPIO.OUT,0);
    // Wait for tRSTL = 48 us
    DelayUS(48);
    // Release Bus Reset
    GPIOWrite(GPIO.OUT,1);
    // Wait for tPDH = 8 us
    DelayUS(8);
    // Wait for Bus Reset Response
    while(GPIORead(GPIO.IN) != 0);
    while(GPIORead(GPIO.IN) != 1);
}
```

Figure 2-3 shows how the host drives the GPIO.OUT pin low for t_{RSTL} (indicated by marker pair 0) and then drives the pin high afterwards. The TMP1826 then waits for t_{PDH} and drives the SDQ pin low for t_{PDL} (indicated by marker pair 1). The host checks on the pin GPIO.IN for the response from TMP1826.

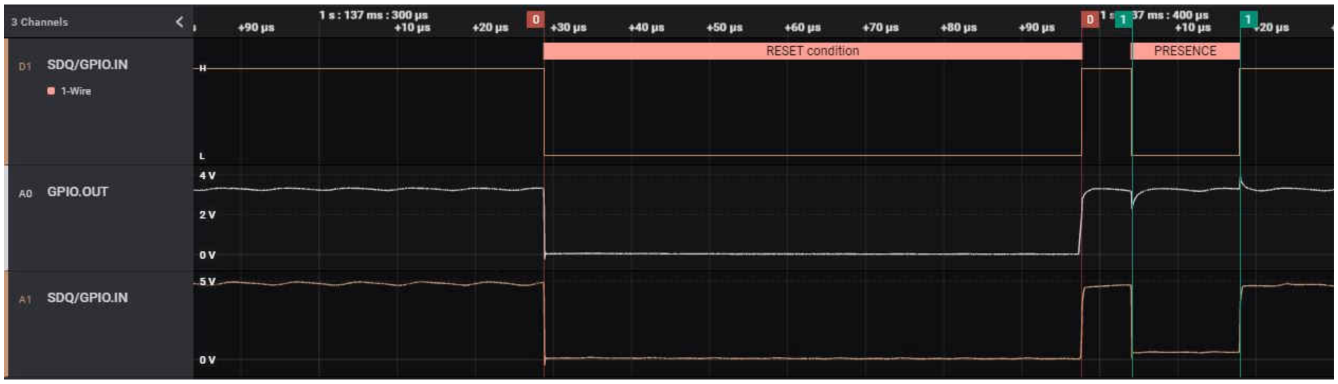

Figure 2-3. Scope Capture of Bus Reset and Response

Once the bus reset is sent and the response received, the host then sends the address command and function command to the device. As all commands and data transfers are at the byte boundary, the following pseudocode illustrates how to send the bytes and bits to the device.

```
void mcu_txOneWireByte(uint8_t bytevalue)
{
    uint8_t count;
    // Shift bits LSB to MSB
    for(count=0; count<8; count++)
    {
        mcu_txOneWireBit((bytevalue >> count) & 0x01);
    }
}

void mcu_txOneWireBit(uint8_t bitvalue)
{
    // If logic '0' then set GPIO to 0 and wait for tWR0L
    if(bitvalue == 0x00)
    {
        GPIOWrite(GPIO.OUT,0);
        // Wait for tWR0L = 10 us
        DelayUS(10);
    }
    // If logic '1' then set GPIO to 0 and wait for tWR1L
    else if(bitvalue == 0x01)
    {
        GPIOWrite(GPIO.OUT,0);
        // Wait for tWR1L = 2 us
        DelayUS(2);
    }
    // Release the GPIO
    GPIOWrite(GPIO.OUT,1);
    // Wait for tREC = 2 us
    DelayUS(2);
}
```

Figure 2-4 shows the write byte transfer of address command 0hCC (SKIPADDR) and function command 0h44 (CONVERTTEMP) for temperature conversion. During the write operation, the GPIO.OUT is set low in accordance with t_{WR0L} and t_{WR1L} to indicate logic '0' and logic '1', respectively. As the device is always in receive state, the device does not respond on the SDQ pin and hence the GPIO.IN pin mirrors the GPIO.OUT.

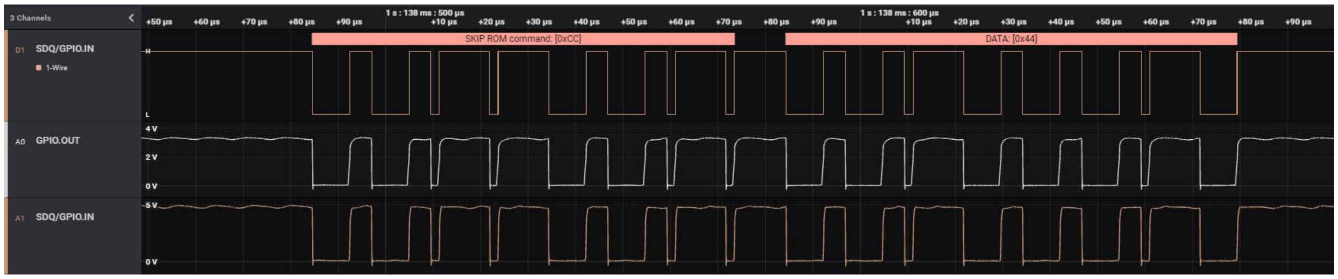

Figure 2-4. Scope Capture of Write Transfer

The final set of pseudocode (following) shows how to implement a read from the device after sending an address or function command which the device can respond to. As with data transfer to the device, whenever data is read back from the device, the read is done at byte boundary. The following function shows how to implement both the byte and bit read functions.

```
uint8_t mcu_rxOneWireByte(void)
{
    uint8_t count;
    uint8_t rxbyte = 0x00;

    // Shift bits LSB to MSB
    for(count=0; count<8; count++)
    {
        rxbyte |= (mcu_rxOneWireBit() << count);
    }
}

uint8_t mcu_rxOneWireBit(void)
```

```
{
    uint8_t rxbit;
    // Send the falling edge to begin the read
    GPIOWrite(GPIO.OUT,0);
    // Wait for tRL = 2 us and release the bus
    DelayUS(2);
    GPIOWrite(GPIO.OUT,1);
    // Wait for tRC and sample the state of the bus
    DelayUS(1);
    rxbit = GPIORead(GPIO.IN);
    // Wait for remainder of the slot time (tSLOT-tMSW)
    DelayUS(8);

    return(rxbit);
}
```

Figure 2-5 shows the write byte of the function command 0hBE (READ SCRATCHPAD-1) and read byte that follows the write byte. The host during the read operation drives GPIO.OUT low for t_{RL}. As indicated by the marker pair 2, if the device has a logic '1' to send to the host, the device does not drive the SDQ pin. When the device has a logic '0' to send, the device drives the SDQ pin low as shown by the marker pair 3. The host must sample the GPIO.IN at t_{MSW} to check if the device is sending a logic '1' or logic '0'.

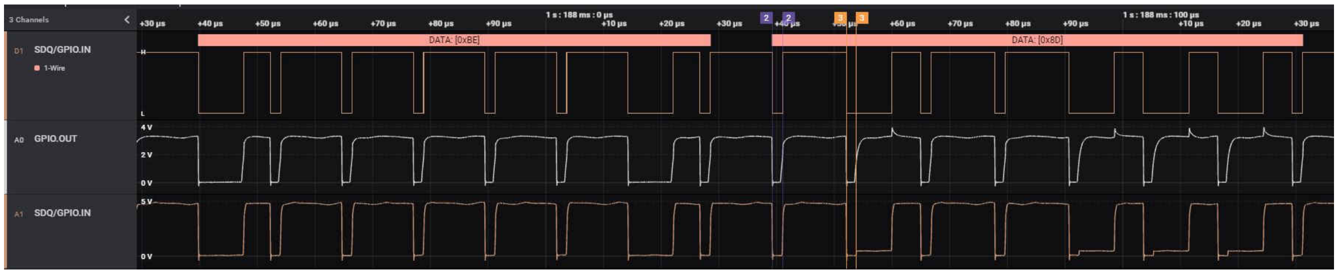

Figure 2-5. Scope Capture of Read Transfer

2.3 Using UART as Host Interface

Universal Asynchronous Receiver Transmitter (UART) is also a common peripheral found on most MCUs. UART is a full-duplex asynchronous bus protocol with a transmit and receive function that are independent. The transmit function is a push-pull pin. Figure 2-6 shows that by using a push pull to open-drain converter and an external pullup resistor, the transmit function can be connected to the single-wire interface. The receive pin can then be tied to the same interface for reading bus reset response and data back from the single-wire devices.

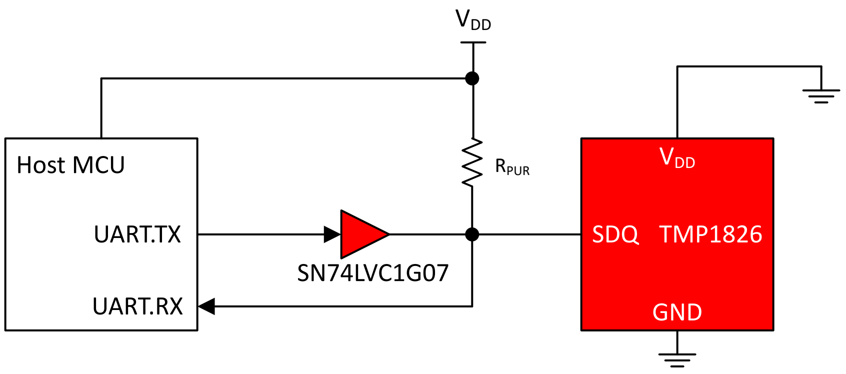

Figure 2-6. UART Interface for TMP1826

The UART bus has a well-defined frame format. The frame itself consists of a mandatory start bit, variable number of data bits, an optional parity, and one or two stop bits. For interfacing the UART peripheral from a host MCU to the single wire bus, use the 8-N-1 format, where there is a start bit, followed by 8 data bits, no parity and 1 stop bit as shown in Figure 2-7.

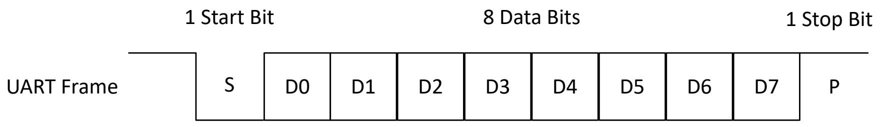

Figure 2-7. UART Frame Structure

Figure 2-8 shows that the UART baud rate is set to 187.5kbps and the host transmits 0h00 for bus reset and receives 0hFC or 0hFE for a bus reset response.

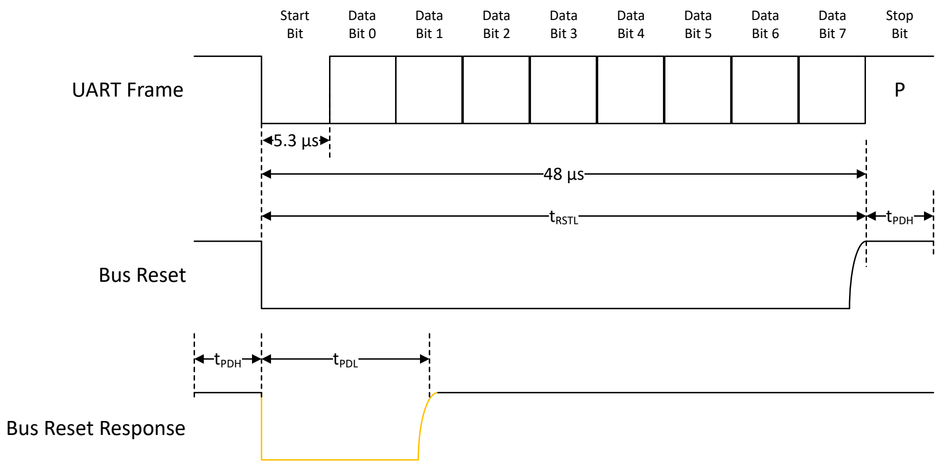

Figure 2-8. UART Frame Overlay for Overdrive Bus Reset and Response

When writing to the devices, the UART baud rate is set to 500kbps and transmits 0hC0 for a logic '0' and 0hFF for a logic '1'.

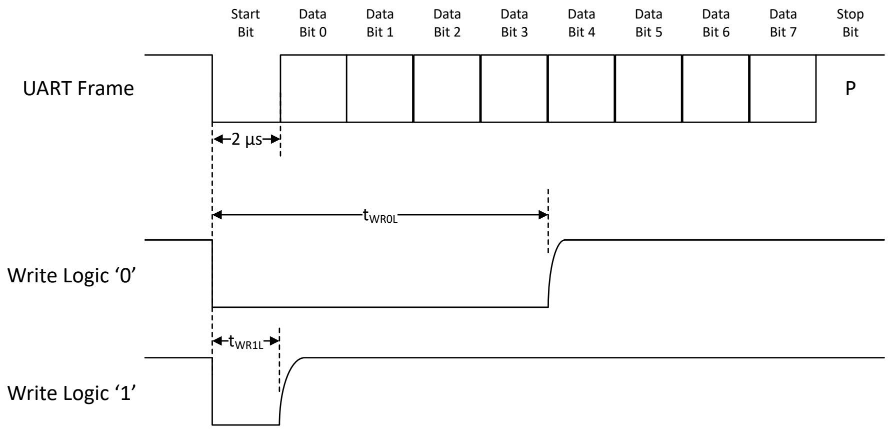

Figure 2-9. UART Frame Overlay for Overdrive Write

When reading from the devices, the UART baud rate is set to 500kbps and transmits 0hFF. If the device sends a logic '0', the host on the receive pin can read the data as 0hFC or 0hF8. However if the device sends a logic '1', the host reads the data as 0hFF or 0hFE.

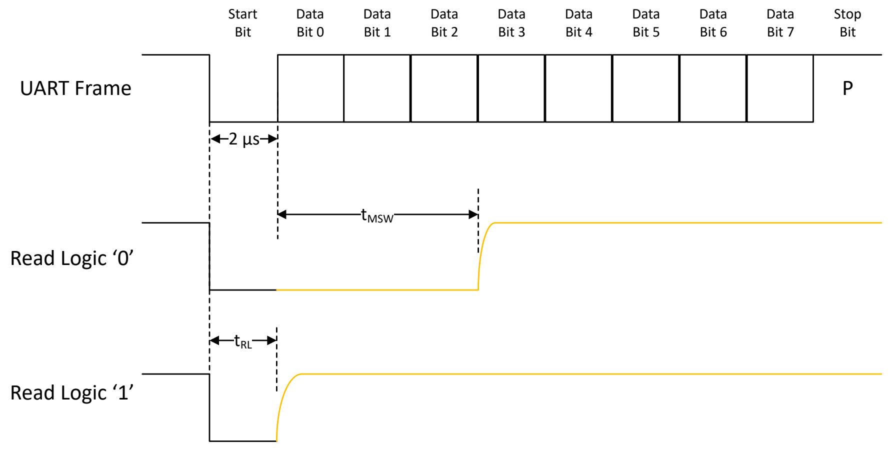

Figure 2-10. UART Frame Overlay for Overdrive Read

Table 2-1 provides the reference baud rates required by bus reset and data bits for standard and overdrive bus speed.

Table 2-1. UART Baud to Single-Wire Bus Speed

UART Baud for Single-Wire Standard Speed		UART Baud for Single-Wire Overdrive Speed	
Bus Reset	Data Bit	Bus Reset	Data Bit
18750bps	115200bps	187500bps	500000bps

2.4 Software Driver for UART

This section describes the UART pseudocode. The following code shows a generic UART configuration for 8-N-1 format with a baud rate of 187,500bps.

```
void ConfigUART()
{
    // Configure UART for Baud Rate of 8-N-1 format and 125,000 bps
    UARTConfig(DATA_8_BIT, PARITY_NONE, STOP_ONE_BIT)
    UARTBaudRate(187500);
}
```

The following pseudocode shows how to implement bus reset and response. As the previous state of the bus communication can be write or read of data over the single-wire interface, TI's recommendation is that the baud rate be explicitly configured before bus reset and response is executed.

```
void mcu_txOneWireReset()
{
    // Configure UART for Baud Rate if 125,000 bps
    UARTBaudRate(187500);
    // Send Bus Reset on Transmit pin
    UARTSendData(0x00);
    // Wait for Response on Receive pin
    while(UARTGetData() != 0xFC);
    // Configure UART for Baud Rate if 500,000 bps
    UARTBaudRate(500000);
}
```

For the previously-described application example code, Figure 2-11 illustrates the bus transactions along with a UART reference frame. The host sends 0h00 for bus reset (indicated by marker 0) with a baud rate of 187.5kbps and the device responds to the reset (indicated by marker 1) with a response of 0hFC.

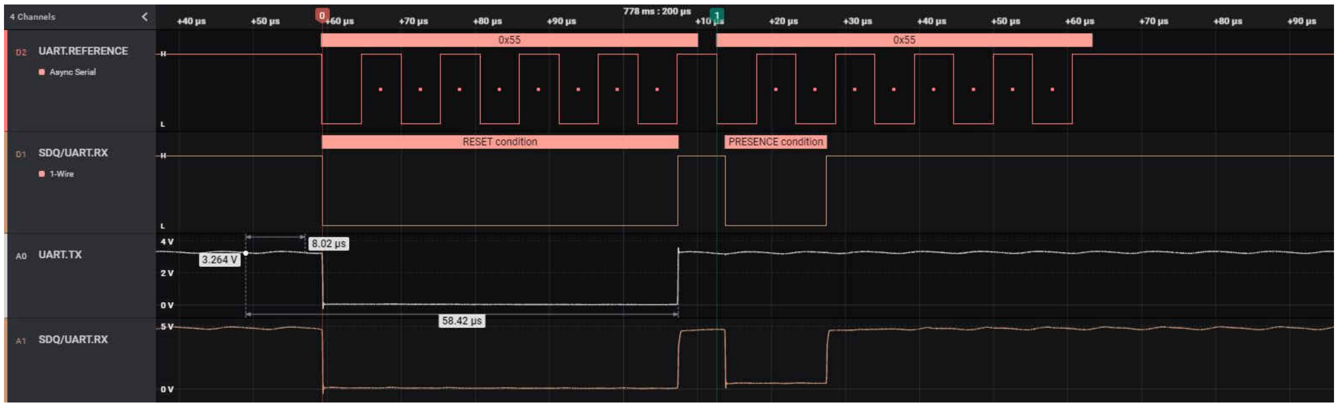

Figure 2-11. Scope Capture of Bus Reset and Response

Once the bus reset is sent and the response received, the host then sends the address command and function command to the device. As all commands and data transfers are at byte boundary, the pseudocode shows how to send the bytes and bits to the device.

```
void mcu_txOneWireByte(uint8_t bytevalue)
{
    uint8_t count;
    // Shift bits LSB to MSB
    for(count=0; count<8; count++)
    {
        mcu_txOneWireBit((bytevalue >> count) & 0x01);
    }
}

void mcu_txOneWireBit(uint8_t bitvalue)
{
    // If logic '0' then send 0xF0
    if(bitvalue == 0x00)
    {
        UARTSendData(0xC0);
    }
    // If logic '1' then send 0xFF
    else if(bitvalue == 0x01)
    {
        UARTSendData(0xFF);
    }
}
```

Figure 2-12 shows the write byte transfer of address command 0hCC (SKIPADDR) and function command 0h44 (CONVERTTEMP) for temperature conversion. During the write operation, the UART.TX is set as 0hF0 and 0hFF to indicated a logic '0' and logic '1', respectively. Because the device is always in receive state, the device does not respond on the SDQ pin and hence the UART.RX pin mirrors the UART.TX pin.

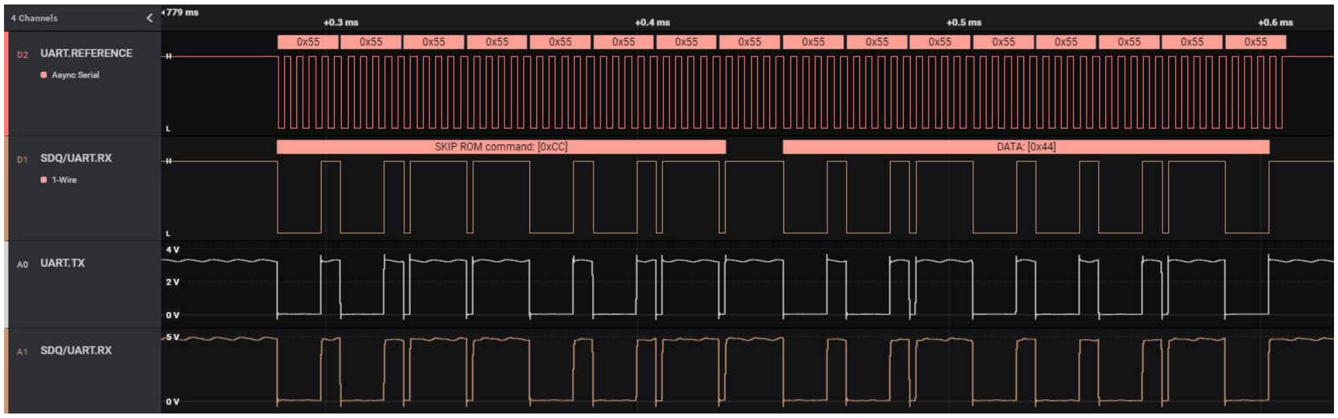

Figure 2-12. Scope Capture of Write Transfer

The following pseudocode shows how to implement read from the device after sending an address or function command to which the device responds. As with data transfer to the device, whenever data is read back from the device, the read is done at byte boundary. The following function shows how to implement both the byte and bit read functions.

```
uint8_t mcu_rxOneWireByte(void)
{
    uint8_t count;
    uint8_t rxbyte = 0x00;
    // Shift bits LSB to MSB
    for(count=0; count<8; count++)
    {
        rxbyte |= (mcu_rxOneWireBit() << count);
    }
}

uint8_t mcu_rxOneWireBit(void)
{
    uint8_t rcvbyte;

    // Send the Falling edge to begin the read
    UARTSendData(0xFF);
    // Get the data from Receive pin
    rcvbyte = UARTGetData();

    if(rcvbyte == 0xFF || rcvbyte == 0xFE)
    {
        return(1);
    }
    else
    {
        return(0);
    }
}
```

Figure 2-13 shows the write of the function command 0hBE (READ SCRATCHPAD-1) and the read byte that follows. The host during the read operation drives UART.TX as 0hFF. As indicated by marker 2, if the device has a logic '1' to send to the host, the device does not drive the SDQ pin and the byte received by UART.RX is 0hFF. When the device has a logic '0' to send to the host, the device drives the SDQ pin low as shown by marker pair 3. The UART.RX of the host pin, in this case receives a 0hF8 or 0hFC.

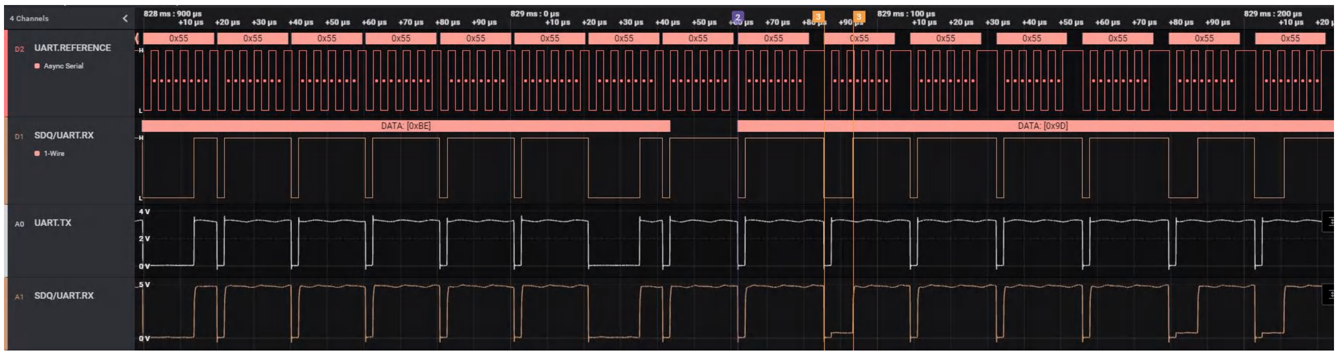

Figure 2-13. Scope Capture of Read Transfer

2.5 Using SPI as Host Interface

The Serial Peripheral Interface (SPI) is another common peripheral found on most MCUs. SPI is a full- or half-duplex synchronous interface generally reserved for high-speed data transfers. The transmit function is a push-pull pin while the receive function is an input function. Similar to the UART implementation, Figure 2-14 shows a method to reconfigure the interface to function as a single-wire interface.

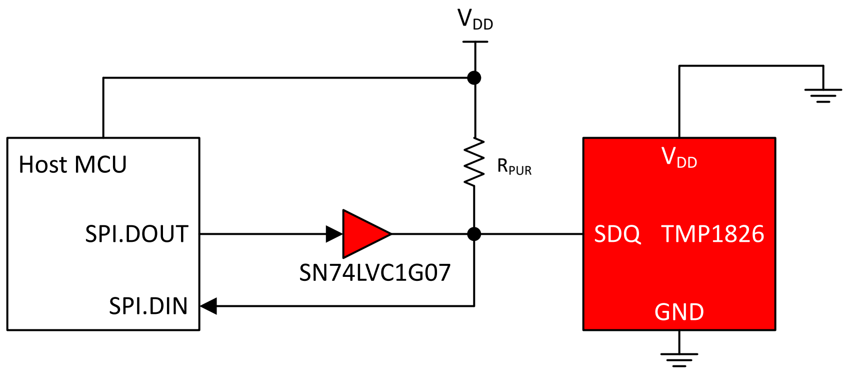

Figure 2-14. SPI for TMP1826

Depending on the implementation, the SPI bus can have variable data lengths from 4 bits to 32 bits along with a serial clock pin. Additionally, clock (CPOL) and data (CPHA) polarity are used to defined the mode of operation. For interfacing the SPI peripheral from a host MCU to the single-wire bus, use an 8-bit data length and mode-1 (CPOL = 0, CPHA = 1) of operation. Similar to UART bus, Table 2-2 provides the reference clock rates required by bus reset and data bits for standard and overdrive bus speed.

Table 2-2. SPI Clock Rate to Single-Wire Bus Speed

SPI Clock for Single-Wire Standard Speed		SPI Clock for Single-Wire Overdrive Speed	
Bus Reset	Data Bit	Bus Reset	Data Bit
16000 Hz	125000 Hz	100500 Hz	500000 Hz

2.6 Software Driver for SPI

The following code shows a generic SPU configuration for 8-bit data and mode-1 of operation with SPI clock rate of 100500 Hz.

```
void ConfigureSPI()
{
    // Configure SPI for 8-bit data, CPOL=0 and CPHA=1
    SPIConfig(DATA_8_BIT, CPOL_0, CPHA_1);
    SPIDataRate(100500);
}
```

The following pseudocode shows how to implement bus reset and response. As the previous state of the bus communication can be write or read of data over the single-wire interface, TI recommends that the SPI clock rate be explicitly configured before bus reset and response is executed.

```
void mcu_txOneWireReset()
{
    // Configure SPI clock speed 100,500 Hz
    SPIDataRate(100500);
    // Send Bus Reset on SPI.DOUT transmit pin
    SPISendData(0x07);
    // Wait for Response on SPI.DIN pin
    while(SPIGetData() != 0x05);
    // Configure SPI clock speed 500,000 Hz
    SPIDataRate(500000);
}
```

For the previously-described application example code, Figure 2-15 illustrates the bus transactions along with an SPI frame. The host sends 0h07 for bus reset (indicated by marker 0) and the device responds to the reset (indicated by marker 1) with a response of 0h05.

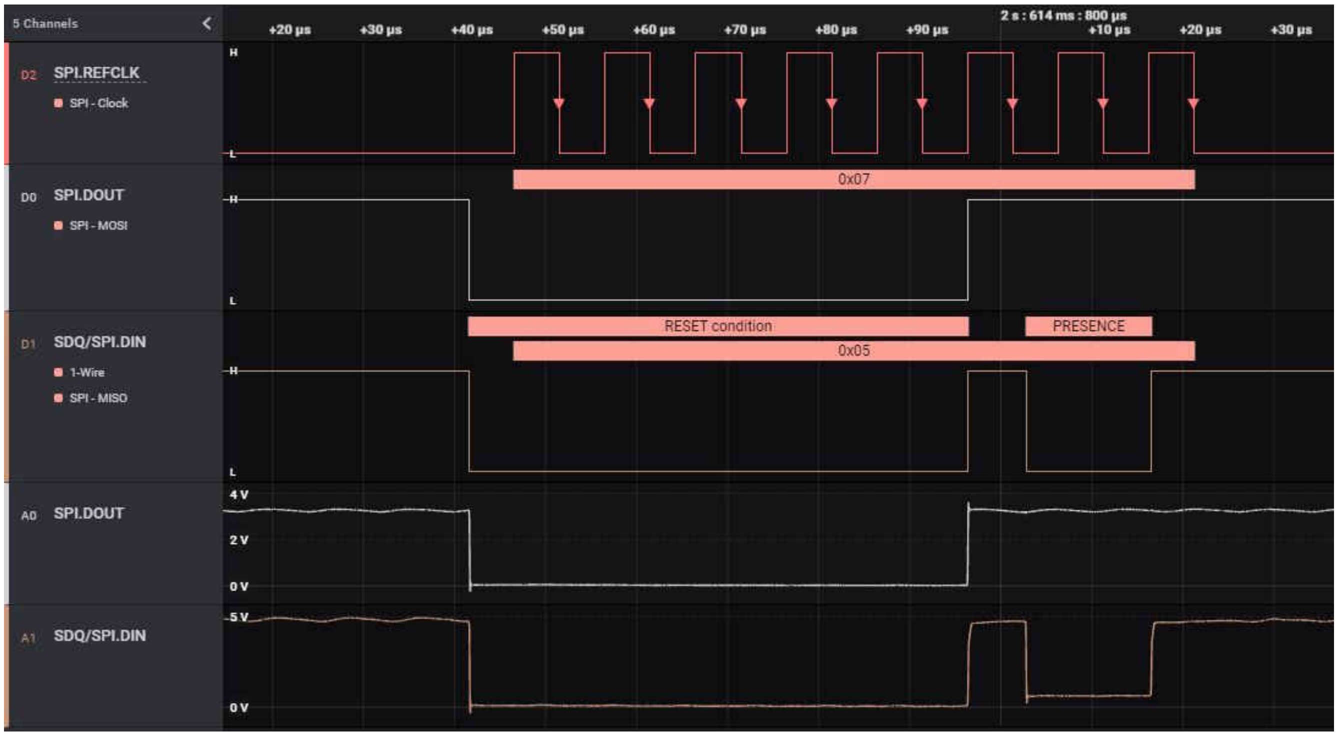

Figure 2-15. Scope Capture of Bus Reset and Response

Once the bus reset is sent and a response received, the host then sends the address command and function command to the device. As all commands and data transfers are at byte boundary, the pseudocode shows how to send the bytes and bits to the device.

```
void mcu_txOneWireByte(uint8_t bytevalue)
{
    uint8_t count;
    // Shift bits LSB to MSB
    for(count=0; count<8; count++)
    {
        mcu_txOneWireBit((bytevalue >> count) & 0x01);
    }
}

void mcu_txOneWireBit(uint8_t bitvalue)
{
    // If logic '0' then send 0xF0
    if(bitvalue == 0x00)
    {
        SPISendData(0x01);
    }
    // If logic '1' then send 0xFF
    else if(bitvalue == 0x01)
    {
        SPISendData(0x7F);
    }
}
```

Figure 2-16 shows the write byte transfer of address command 0hCC (SKIPADDR) and function command 0h44 (CONVERTTEMP) for temperature conversion. During the write operation the SPI.DOUT is set as 0h01 and 0h7F to indicated a logic '0' and logic '1', respectively. Since the device is always in receive state, the device does not respond on the SDQ pin and hence the SPI.DIN pin mirrors the SPI.DOUT pin.

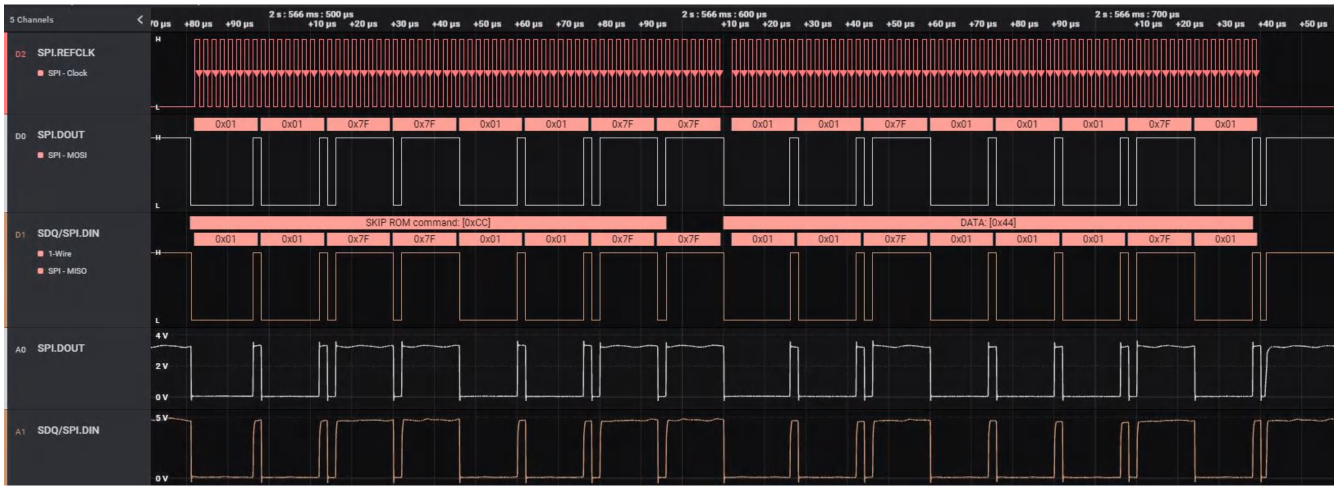

Figure 2-16. Scope Capture of Write Transfer

The following pseudocode shows how to implement read from the device after sending an address or function command to which the device can respond. As with data transfer to the device, whenever data is read back from the device, the read is done at byte boundary. The following function shows how to implement both the byte and bit read functions.

```
uint8_t mcu_rxOneWireByte(void)
{
    uint8_t count;
    uint8_t rxbyte = 0x00;
    // Shift bits LSB to MSB
    for(count=0; count<8; count++)
    {
        rxbyte |= (mcu_rxOneWireBit() << count);
    }
}

uint8_t mcu_rxOneWireBit(void)
{
    uint8_t rcvbyte;

    // Send the Falling edge to begin the read
    SPISendData(0x7F);
    // Get the data from Receive pin
    rcvbyte = SPIGetData();

    if(rcvbyte == 0x7F)
    {
        return(1);
    }
    else
    {
        return(0);
    }
}
```

Figure 2-17 shows the write of the function command 0hBE (READ SCRATCHPAD-1) and the read byte that follows. The host during the read operation drives SPI.DOUT as 0h7F. As indicated by marker 2, if the device has a logic '1' to send to the host, the device does not drive the SDQ pin and the byte received by SPI.DIN is 0h7F. When the device has a logic '0' to send to the host, the device drives the SDQ pin low as shown by marker pair 3. The SPI.DIN pin of the host in this case receives a 0h0F.

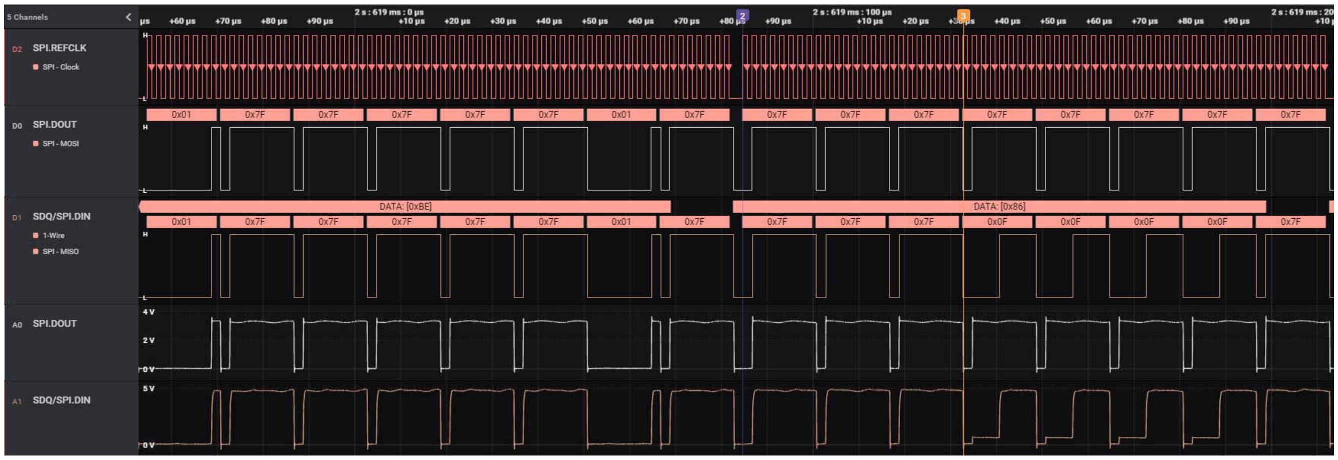

Figure 2-17. Scope Capture of Read Transfer

3 Summary

In conclusion, this application note explains how host MCUs ranging from slow 8- or 16-bit processors to fast 32-bit processors can be used to implement the host controller function for a single-wire TMP1826 and TMP1827 digital temperature sensors using bit-banged GPIO as well as higher throughput peripherals like UART and SPI. Based on the application and host MCU capabilities one or more of these methods can significantly reduce the development effort for software engineers. In addition, Texas Instruments provides an MCU independent application framework software which can rapidly get a design from concept to production.

With the capability of integrating power delivery and digital communication in a single conductor along with a ground reference, the TMP1826 and TMP1827 devices provide excellent scalability in applications like factory automation, power delivery, building automation, and medical equipment.

4 References

- Texas Instruments, *TMP1826 Single-Wire, ±0.3°C Accurate Temperature Sensor With 2-Kbit EEPROM* data sheet
- Texas Instruments, *TMP1827 Single-Wire, ±0.3°C Accurate Temperature Sensor With 2-Kbit EEPROM and SHA-256-HMAC Authentication Engine* data sheet
- Texas Instruments, *Analog Signal Chain Code Studio*